Universidad de Valladolid

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

DEPARTAMENTO DE INFORMÁTICA

TESIS DOCTORAL:

Parallel Approaches to Shortest-Path Problems for Multilevel Heterogeneous Computing

Presentada por

D. Héctor Ortega Arranz

para optar al grado de doctor por la Universidad de Valladolid

Dirigida por:

Dr. Diego R. Llanos Ferraris
Dr. Arturo González Escribano

Valladolid,
2015

Resumen

Desde hace mucho tiempo, diferentes algoritmos de grafos han solucionado problemas relacionados con la computación del camino más corto. Este tipo de problemas se considera como uno de los problemas más importantes dentro de la categoría de optimización combinatoria, debido a sus múltiples aplicaciones a problemas prácticos relacionados con la vida real. Entre algunos de estos ejemplos se encuentra la navegación de coches y/o robots en carreteras u otras superficies, las simulaciones de tráfico, optimización de recursos para compañías de envío de paquetes, rutado de paquetes en Internet, búsquedas en la web, o la explotación de beneficios con cambios de monedas. Sin embargo, durante las últimas décadas, el interés de la comunidad científica por este tipo de problemas ha crecido significativamente, no sólo por la amplia aplicabilidad de sus soluciones, sino también por el actual popular uso eficiente de la computación paralela. Además, la aparición de nuevos modelos de programación junto con los modernos aceleradores hardware, como los procesadores de gráficos (GPUs) y las tarjetas XeonPhi, ha enriquecido el rendimiento de los algoritmos paralelos anteriores, y ha propiciado la creación de nuevos algoritmos más eficientes. Por otra parte, el uso conjunto de estos aceleradores hardware junto con las clásicas CPUs conforman la herramienta perfecta para enfrentarse, dentro de un contexto de cómputo de altas prestaciones, a los problemas más costosos relacionados con el cálculo de caminos más cortos.

Sin embargo, la programación de estos aceleradores hardware, su optimización, y su coordinación con otros elementos de computación de diferente naturaleza, son tareas aún muy complicadas para los programadores no expertos. Una de las razones por las que estas tareas son demasiado complejas es la falta de estudios que puedan guiar al programador a utilizar valores apropiados para los parámetros de ejecución de las GPUs. Respecto a la coordinación de diferentes unidades computacionales, tampoco existen muchos modelos o herramientas que simplifiquen la programación de diferentes capas de cómputo paralelo. Por ejemplo, el uso conjunto de los diferentes cores de una CPU dentro de un sistema de memoria compartida, o incluso de otros sistemas externos.

Este trabajo de Tesis Doctoral aborda ambos contextos mencionados anteriormente mediante: el desarrollo de nuevos planteamientos orientados a plataformas GPU para la resolución de problemas de caminos cortos, junto con el estudio del ajuste óptimo de los parámetros de ejecución; y el diseño de soluciones donde algoritmos secuenciales y paralelos son desplegados, de manera concurrente, en entornos heterogéneos.

Palabras clave

Modelo paralelo abstracto, APSP, tuning automático de kernels, Boost Graph Library, configuración de la Cache L1, ejecución concurrente de kernels, CUDA, Dijkstra, GPGPU, sistemas heterogéneos, framework HPC, modelo de caracterización de kernels, proceso de caracterización de kernels, balanceo de carga, MPI, comparativa de plataformas de NVIDIA, OpenMP, técnicas de optimización, algoritmos paralelos, SSSP, geometría del bloque de hilos.

Abstract

Many graph algorithms have given solution to the problem of finding shortest paths between nodes in a graph. These problems are considered among the fundamental combinatorial optimization problems. They have many applications, such as car/robot navigation systems, traffic simulations, tramp steamer problem, courier-scheduling optimization, Internet route planners, web searching, or exploiting arbitrage opportunities in currency exchange, among others. During the last decades, the interest of the scientific community in these problems has significantly increased not only due to this wide-applicability, but also thanks to the currently popular and efficient parallel computing. Additionally, the advent of new parallel programming models together with modern powerful hardware accelerators, such as the Graphics Processing Units or the many-core XeonPhis boards, may highly improve the performance of previous parallel algorithms, and also has open the possibility to study new and more efficient parallel approaches to exploit these specific architectures. Furthermore, the emerging of heterogeneous parallel computing combining these powerful hardware accelerators with the classical and increasingly powerful CPUs, provides a perfect environment to face the most costly shortest-path problems in the context of High Performance Computing (HPC).

However, the programming of hardware accelerators, the optimization of their running times, and also, the coordination of these devices with other computational units of different nature, are still very complex tasks for non-expert programmers. One important indicator of the added complexity found in these environments is the lack of studies to guide the programmer into the correct use of proper values for GPU runtime configuration parameters. Regarding the coordination of different computational devices, there are also few models or frameworks that simplifies the programming when different parallel computing layers are used, to combine the use of many-cores and classical CPU cores present in a shared-memory system, or even from different systems.

This Ph.D. thesis addresses both mentioned problems, the algorithmic GPU programming and the heterogeneous parallel coordination in the context of: Developing new GPU-based approaches to the shortest path problem; the study of the tuning of the GPU configuration parameters; and also, designing solutions where both sequential and parallel algorithms are deployed concurrently in heterogeneous environments.

Keywords

Abstract parallel model, APSP, automatic kernel tuning, Boost Graph Library, cache L1 configuration, concurrent kernel execution, CUDA, Dijkstra, GPGPU, heterogeneous systems, HPC framework, kernel characterization model, kernel characterization process, load-Balancing, MPI, NVIDIA platform comparison, OpenMP, optimization techniques, parallel algorithms, SSSP, threadblock size.

“Donde no falta voluntad siempre hay un camino”

“If there is a will, there is a way”

J.R.R. Tolkien (1892 - 1973)

Acknowledgments

Aún recuerdo cuando de pequeño, y no tan pequeño, hablaba de recorrer esta ruta, que aún era desconocida para mí. Recuerdo cuando empecé a andarla, y en particular, el momento cuando descubrí, qué misterios se discernían tras estos caminos. Recuerdo sentir cómo ese crucero se transformaba en odisea, y el desafío se tornaba más difícil de lo que a veces uno desea. Muchas otras cosas seguro habré dejado en el pasado, pero de lo que no me he olvidado, es que no estaba yo sólo en esa senda. Gracias a los que me han acompañado, puedo recordar que he sido capaz de enfrentarme a todo por perseguir esa meta. Es a ellos a quienes van dedicados estos párrafos.

En primer lugar quiero agradecer a Diego y Arturo por aceptarme y darme la oportunidad de empezar este viaje, por su ayuda y apoyo durante el largo y duro recorrido, y su trabajo para que todo esto llegara a buen puerto. Quiero agradecer también, y resaltar la importancia de haber tenido unos inmejorables compañeros de travesía, y amigos, Yuri, Sergio, Javier, Álvaro y Ana. Sin su compañía, comprensión, empatía y ayuda esta andadura habría sido siempre muy oscura.

Igualmente quiero agradecer el insuperable trato que recibí durante mi pequeña y entrañable aventura en St. Andrews. No sólo de la gente de la universidad que me acogió, apoyó y confió en mí como si siempre hubiera trabajado con ellos, sino también a todas las personas que allí conocí, y con las que mucho más que anécdotas inolvidables compartí.

Por último, agradecer a mis amigos, a mis padres, y a mi hermano Alex, que tantas veces me han visto partir, siempre me han ayudado y siempre han confiado en mí, y nunca me han faltado cuando regresaba y les necesitaba. Y a quien siempre ha estado ahí, también, GRACIAS!

Héctor Ortega-Arranz
Valladolid, 2015

This research has been partially supported by the FPI-UVa 2011 and HiPEAC 2014 scholarships, the Ministerio de Economía y Competitividad (Spain) and ERDF program of the European Union: CAPAP-H5 network (TIN2014-53522-REDT), MOGECOPP project (TIN2011-25639); Junta de Castilla y León (Spain): ATLAS project (VA172A12-2); HomProg-HetSys project TIN2014-58876-P; and the COST Program Action IC1305: NESUS.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter R

Resumen de la tesis

Muchos problemas que surgen en las redes del mundo real requieren calcular los caminos más cortos, y sus distancias, entre uno o varios puntos de origen a uno o varios puntos de destino. Existen diferentes variantes del problema. El problema del SSSP, o Single-Source Shortest-Path, tiene como objetivo calcular los caminos más cortos y sus distancias entre un punto origen y el resto. Por otro lado, el problema del APSP, o All-Pair Shortest-Path, tiene como objetivo calcular los caminos más cortos y sus distancias entre cualquier par de puntos de la red. Algunos ejemplos de contextos donde este tipo de cómputo es necesario son los sistemas de navegación [1], simulaciones de tráfico [2], medios de transporte con horario fijo [3], control logístico [4], bases de datos espaciales [5, 6], planificación de rutas en Internet [7], o búsquedas en la web [8, 9]. A pesar de la importancia del problema de cálculo de caminos más cortos, los algoritmos actuales son aún muy costosos en términos computacionales, y en muchos casos los productos comerciales implementan abortamientos heurísticos para generar soluciones aproximadas en vez de soluciones óptimas.

El objetivo de paralelizar estos algoritmos implica no sólo la inmediata reducción de los tiempos de ejecución sino también su aplicación en otros planteamientos más complejos donde estos algoritmos representan una fase dentro del método global. De esta manera, estos métodos, que previamente eran sólo teóricos debido a prohibitivos tiempos de ejecución, pueden ser ahora factibles y viables gracias a esta reducción. Un ejemplo de este tipo de fases son los costosos preprocesos necesarios para ejecutar consultas en el cálculo de rutas de un mapa de carreteras. En este preproceso se obtienen valores que son almacenados para ser utilizados a posteriori en la consulta, donde se pide calcular la ruta más corta entre dos puntos del mapa. Normalmente, este preproceso tiene un coste muy alto en términos temporales, pero mientras sólo haya que ejecutarlo una vez cada mucho tiempo, podría considerarse como una tarea de coste asequible. Gracias a estos valores precomputados, que en ocasiones representan el cálculo de todas las distancias de todas las parejas de puntos de la red, la fase de consulta puede ser computada en el orden de nanosegundos [10, 11].

La configuración de los mapas de carreteras no cambia frecuentemente, por lo que podría ser plausible pagar el alto coste de la fase de preproceso una vez. Sin embargo, en otros contextos donde las redes tienden a cambiar con más frecuencia, o su topología es desconocida en un momento determinado, la reducción de tiempos de esta fase de pre-

proceso se convierte en algo crucial. Además, en el contexto de los mapas de carreteras, podemos encontrar también este comportamiento dinámico si queremos tener en cuenta otros factores en tiempo real como pueden ser el estado de las carreteras o el tráfico.

La aparición de los dispositivos móviles presenta un tercer desafío debido a su poca capacidad de almacenamiento que limita la cantidad de datos precomputados que podríamos almacenar. La mayoría de los métodos actuales son más rápidos cuanta más datos precomputados puedan almacenar. Sin embargo, los dispositivos móviles actuales tienen una gran capacidad de cómputo incorporando varios procesadores cuyo explotación en paralelo podría aliviar la mencionada falta de almacenamiento.

La computación paralela consiste en utilizar a la vez dos o más dispositivos para realizar cálculos, normalmente con el propósito de reducir los tiempos de ejecución. Aunque la paralelización de un algoritmo no reduce su tiempo asintótico, a veces es la única manera de conseguir tiempos de ejecución razonables y/o competitivos. La tendencia de los ordenadores actuales es incorporan un mayor número de procesadores en vez de un único procesador con mucha velocidad de cómputo. Esta evolución ha derivado en la creación de tipos diferentes de dispositivos de cómputo: los sistemas multi-core y los sistemas many-core. Los primeros incluyen dos o más núcleos de procesamiento, o cores, que son de propósito general dentro de un mismo chip. Sin embargo, un sistema es considerado many-core, cuando tiene más de una docena de estas unidades de procesamiento. En esta categoría podemos encontrar no sólo superordenadores con un gran número de CPUs con muchos cores, sino también los aceleradores hardware como los procesadores gráficos (GPUs) o los coprocesadores como los dispositivos Xeon-Phi.

Actualmente, los dispositivos con arquitectura many-core más representativos son las GPUs [12]. Estos están diseñados para ayudar a la CPU en el procesamiento de gráficos. Sin embargo, debido sus altas capacidades computacionales han hecho que se utilicen para otro tipo de propósitos de ámbito general. Con el fin de facilitar este tipo de programación, NVIDIA desarrolló un nuevo modelo de programación, llamado CUDA [13], en 2007. Desde entonces muchas soluciones sofisticadas y eficientes se han desarrollado para diferentes aplicaciones y problemas [14].

Gracias a este modelo de programación es posible desarrollar fácilmente soluciones básicas para GPUs que obtienen mejoras en el rendimiento. En contraposición, obtener una optimizar la explotación de los recursos computacionales es una tarea difícil. Se necesita conocer en profundidad muchos de los detalles relacionados con la arquitectura de estos dispositivos y su correspondiente gestión, para poder predecir su comportamiento. Esta responsabilidad recae de manera directa sobre los propios programadores, que han de proveer varios valores para los parámetros de ejecución de las GPUs, como por ejemplo el tamaño y forma de los bloques de hilos, o el estado/tamaño de la memoria cache de primer nivel, entre otros. Las guías de programación que ofrece CUDA sugieren el uso de determinados valores para obtener buenos rendimientos. Sin embargo, algunos estudios [15, 16] han demostrado que en algunos estas recomendaciones no siempre devuelven rendimientos óptimos, obligando a los programadores a realizar test de prueba-y-error para encontrar los valores que se ajustan a los mejores rendimientos.

Dentro de la computación paralela se encuentra la computación heterogénea [17, 18], que se refiere al uso de sistemas o entornos de computación que están compuestas por unidades de procesamiento de diferente naturaleza. Algunos ejemplos clásicos son los

convencionales sistemas de memoria compartida o distribuida que contienen multi-core CPUs junto con dispositivos many-core como las GPUs. Una de las principales ventajas de usar este paradigma de computación, a diferencia del modelo homogéneo, es la posibilidad de asignar un tipo particular de tareas/funciones a aquellos dispositivos computacionales que mejor se ajusten a los requisitos de procesamientos de esas tareas. Otra ventaja, es la posibilidad de aprovechar cualquier unidad computacional presente dentro del sistema heterogéneo, aunque sus características no se ajusten perfectamente para la realización óptima de un trabajo específico, siempre y cuándo podamos ahorrar tiempo de la ejecución global de tareas.

Sin embargo, la programación de este tipo de entornos heterogéneos resulta bastante difícil comparado con la programación de entornos homogéneos. El programador debe conocer los diferentes modelos y lenguajes de programación necesarios para aprovechar los diferentes recursos computacionales, sus limitaciones y particularidades, para proveer diferentes implementaciones dependiendo de la plataforma donde se vaya a ejecutar. Además el programador ha de conocer qué tipo de tareas se ajusta mejor a qué dispositivo para obtener un mejor rendimiento y cuánto trabajo ha de asignar a cada uno. Todos estos problemas hacen que la programación sobre sistemas heterogéneos sea casi el resultado de un trabajo artesanal en el que hay que invertir mucho tiempo y esfuerzo debido a la falta de modelos de programación de propósito general que puedan lidiar con este tipo de complejidad de una manera transparente.

R.1 Pregunta de Investigación

La identificación de los problemas expuestos en la anterior sección, nos conduce a la siguiente pregunta de investigación, que se resuelve en esta Tesis Doctoral:

> *¿Es posible desarrollar técnicas y herramientas para conseguir implementaciones más eficientes que resuelvan problemas relacionados con el cálculo de caminos más cortos utilizando: (1) los modernos dispositivos de procesamiento de gráficos (GPUs), y su optimización mediante técnicas de ajuste de los parámetros correspondientes, (2) entornos heterogéneos compuestos por este tipo de aceleradores hardware en conjunto con el uso de las tradicionales CPUs?*

R.1.1 Metodología de Investigación

La metodología de investigación llevada a cabo en esta tesis doctoral está basada en el método de la ingeniería de software que se compone de las cuatro siguientes fases: Observar las soluciones existentes; proponer mejores soluciones; desarrollar las soluciones propuestas; y medir y analizar los resultados [19]. Se trata de una metodología iterativa que puede ser repetida para refinar las soluciones propuestas. Esta metodología se parece al clásico método científico: Proponer una pregunta; formular una hipótesis; realizar una predicción; y validar la hipótesis.

1. *Observar las soluciones existentes*. Esta etapa de exploración tiene el propósito de encontrar problemas/limitaciones que serán abordados durante el proceso de inves-

tigación, y de detectar posibles mejoras y/o nuevas soluciones aún no contempladas. Esto conlleva un completo estudio del estado del arte con el objetivo de encontrar trabajos relacionados con nuestra investigación.

2. *Proponer mejores soluciones*. En esta etapa se realiza el análisis y diseño necesarios para abordar los límites o aprovechar las posibles mejoras encontradas en el paso anterior.

3. *Desarrollar las soluciones propuestas*. La metodología de esta fase consiste en el desarrollo o construcción de un prototipo de la solución que demuestre que el planteamiento propuesto es factible.

4. *Medir y analizar la nueva solución*. En esta última fase de la metodología, las implementaciones de los prototipos de las soluciones son evaluados a través de un estudio experimental. El objetivo de este estudio es corroborar si estas soluciones realmente resuelven los problemas descubiertos en la primera etapa.

R.2 Objetivos

Para poder responder a la pregunta de investigación propuesta, hemos realizado las tareas que se describen a continuación (ver Fig. R.1). Se han aplicado las diferentes etapas de la metodología descrita anteriormente para cada una de ellas.

Objetivo 1: Desarrollar un planteamiento nuevo para el problema del SSSP.

$(Observación)$ Se ha realizado un completo estudio del trabajo previo, visitando tanto las soluciones secuenciales como sus variantes paralelas para el problema del SSSP. Después de este análisis, el foco de la investigación se centrará sobre los algoritmos para un tipo de grafo en particular, grafos no densos.

$\binom{Propuesta\ y}{desarrollo}$ Combinando diferentes ideas de la literature, un nuevo algoritmo para GPUs será propuesto. Implementaremos este algoritmo utilizando CUDA para poder optimizarlo y ejecutarlo sobre GPUs de NVIDIA.

$(Medidas)$ La viabilidad de nuestra propuesta será evaluada comparándola con el último algoritmo del estado-del-arte para GPUs, y también con una versión secuencial optimizada de referencia, usando un conjunto de grafos sintéticos y de redes reales como benchmarks. Para ambos casos se espera que la nueva propuesta obtenga mejores rendimientos.

Objetivo 2: Optimizar nuestra implementación con apropiado ajuste de los parámetros de ejecución de las GPUs.

$(Observación)$ El despliegue en las GPUs de las implementaciones desarrolladas requiere la elección de muchos valores para ciertos parámetros de ejecución, como son por ejemplo, el tamaño de los bloques de hilos, o la activación de la memoria

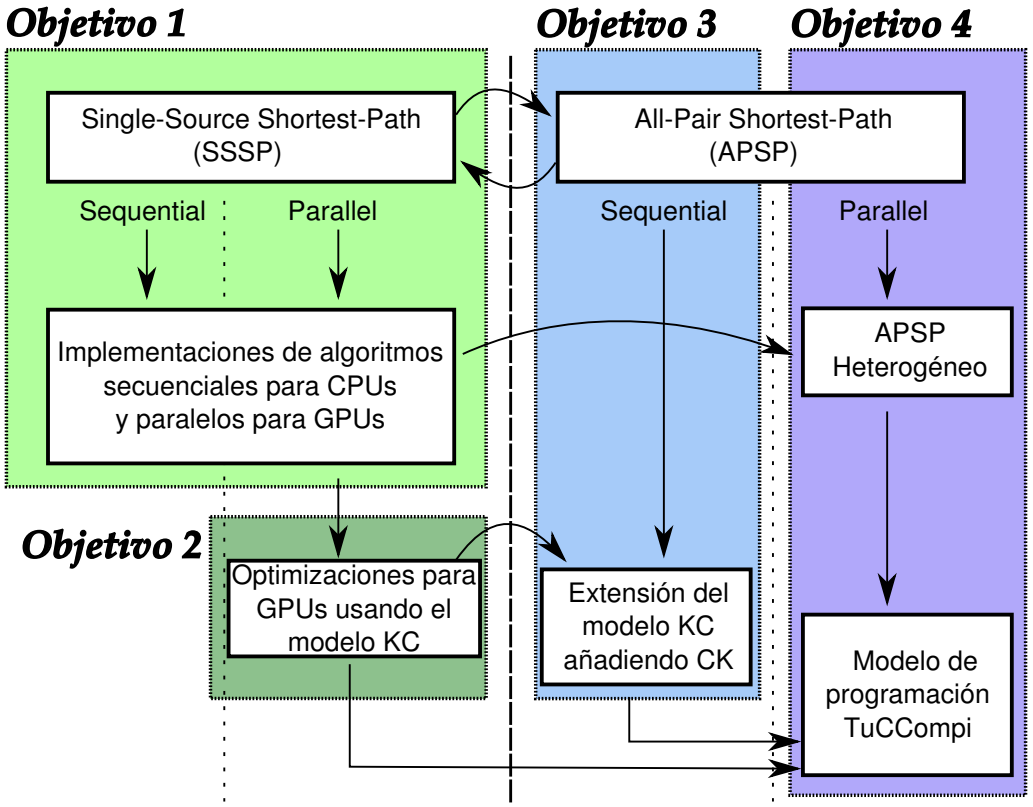


Figure R.1: Diferentes objetivos propuestos para el desarrollo de esta tesis doctoral.

caché de primer nivel, entre otros. Su adecuado ajuste, o tuneado, puede conllevar a importantes mejoras de rendimiento. Se debe revisar el estado-del-arte en busca de algún tipo de estudios o guías de tuneado que ayuden al programador con estas importantes decisiones que sirven para mejorar la solución propuesta.

$\binom{Propuesta\ y}{desarrollo}$ Debemos encontrar un modelo que provea de guías o pasos a seguir para poder ajustar de manera adecuada nuestra solución. Idealmente, ese modelo debería basarse en las características de los códigos que se ejecutan en la GPU, lo que se conoce como caracterización de los kernels (KC). Después de proveer al modelo con esta caracterización, éste debería recomendarnos cuáles son los valores que nos permitirán ejecutar nuestros kernels con un rendimiento óptimo o cerca del óptimo.

$(Medidas)$ Llevaremos a cabo experimentos que usen ambas versiones, tuneado y sin tunear, con el fin de comprobar si este proceso de predicción de valores es útil.

Objetivo 3: Resolver el problema del APSP usando funcionalidades nuevas de las GPUs modernas, y extender el modelo de caracterización de kernels con ellas.

$(Observación)$ Se ha realizado un completo estudio del trabajo previo, visitando tanto las soluciones secuenciales como sus variantes paralelas para el problema del APSP. Después de este análisis, el foco de la investigación se centrará sobre planteamientos basados en paralelización por productividad. Una de las nuevas características

que poseen las GPUs modernas es la capacidad de ejecutar varios kernels independientes a la vez, siempre y cuando haya recursos que no estén siendo utilizados en ese momento. Realizar un estudio de esta funcionalidad y su comportamiento nos permitirá entender mejor cómo funciona esta característica, aprovecharla para mejorar nuestros planteamientos, y ver si afecta al resto de parámetros de ejecución de las GPUs.

$\binom{Propuesta\ y}{desarrollo}$ Modificaremos nuestra solución del SSSP para que incluya la ejecución concurrente de kernels. Esta técnica nos permitirá resolver el problema del APSP a través del método basado en productividad conocido como $n \times SSSP$, donde se ejecuta cada SSSP con un nodo origen diferente de manera independientemente. Refinaremos un modelo de caracterización de kernels ya existente [16], considerando no sólo la nueva funcionalidad de la ejecución concurrente de kernels, sino también para que tenga en cuenta algunas de las características de los grafos de entrada.

$(Medidas)$ Llevaremos a cabo un completo conjunto de estudios sobre diferentes arquitecturas de las GPUs para verificar, no sólo la correctitud de la nueva solución que resuelve el APSP, sino también la validez de los valores del nuevo modelo.

Objetivo 4: Explorar la explotación de entornos heterogéneos para resolver el problema del APSP.

$(Observación)$ Durante el anterior exploración de la literatura del problema del APSP no se encontraron no se encontraron estudios que combinaran algoritmos paralelos con métodos basados en productividad paralela.

$\binom{Propuesta\ y}{desarrollo}$ Proponemos combinar ambas versiones de un algoritmo SSSP, secuencial y paralela para GPUs, en la misma implementación para resolver el APSP. De esta manera podemos aprovechar todo tipo de unidades computacionales de diferente naturaleza presentes en un mismo sistema de computación, asignando la versión secuencial a cada CPU core, y la versión paralela a cada GPU disponible. Además, implementaremos diferentes políticas de distribución de carga.

Por último queremos crear un nuevo modelo de programación que incorpore este novedoso planteamiento, y que además combine la obtención de valores adecuados para los parámetros de ejecución a través de la caracterización de los kernels. Por lo tanto, desarrollaremos un prototipo para este framework de programación que integre todas estas contribuciones descritas anteriormente.

$(Medidas)$ Llevaremos a cabo diferentes experimentos que usen grafos con diferentes características, y diferentes técnicas de distribución de carga, con el objetivo de evaluar la eficiencia del nuevo planteamiento en contraposición al tradicional uso de un único dispositivo de cómputo. Por último, probaremos el prototipo resolviendo el problema del APSP en diferentes clústers heterogéneos.

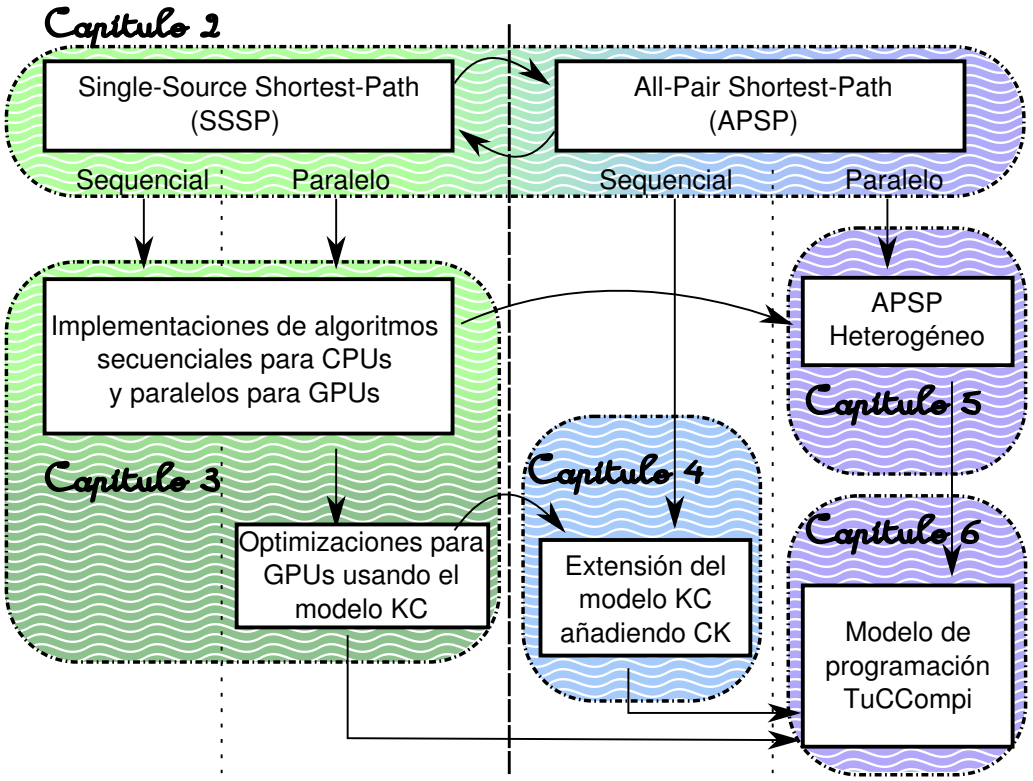


Figure R.2: Estructura del documento.

R.3 Estructura de la Tesis

Este documento está organizado de la siguiente manera (ver Fig. R.2). El capítulo 2 presenta un estudio del estado del arte de los algoritmos secuenciales y paralelos que resuelven los problemas del SSSP y del APSP, y un ejemplo de aplicación al mundo real de estos métodos. El capítulo 3 describe en profundidad el último algoritmo del estado del arte para GPUs que resuelve el problema del SSSP, y nuestra nueva implementación junto con una versión optimizada. El capítulo 4 introduce una extensión de un modelo que utiliza la caracterización de los kernels para predecir los valores óptimos de los parámetros de ejecución de las GPUs. El capítulo 5 presenta una solución para el problema del APSP que combina el uso de algoritmos paralelos diseñados para GPUs junto con algoritmos secuenciales, a través de métodos de productividad paralelos diseñados para un sistema heterogéneo. Además, presenta el uso de diferentes técnicas de balanceo de carga para este problema. El capítulo 6 describe un modelo de programación que simplifica la programación de sistemas heterogéneos que incluyen aceleradores hardware porque el modelo se encarga automáticamente de esconder los detalles de sincronización, despliegue y ajuste de estos dispositivos. Por último, el capítulo 7 contiene las conclusiones de esta tesis de doctorado, enumera las contribuciones que aporta y las publicaciones surgidas de este trabajo.

R.3.1 Síntesis de Capítulos y Contribuciones

En esta sección se describe una síntesis de los contenidos que se muestran a lo largo de esta tesis doctoral junto con los resultados y las contribuciones obtenidas. Además se enumeran los artículos publicados que recogen el trabajo de cada uno de ellos.

Capítulo 2: Estado del arte de los problemas de caminos más cortos

El objetivo de este tipo de problemas está relacionado con el cálculo de los caminos más cortos, y sus distancias, entre los diferentes nodos de un grafo. Dependiendo de que solución particular sea necesaria calcular, podemos identificar diferentes variantes de estos problemas. El problema del SSSP (Single-Source Shortest-Path) tiene como objetivo calcular todos los caminos más cortos entre un origen específico y el resto de nodos del grafo. Una extensión de este problema, la variante APSP (All-Pair Shortest-Path), tiene como objetivo calcular todos los caminos más cortos de todas las posibles parejas de nodos del grafo. Además, existen más derivaciones de este tipo de problemas, como por ejemplo el cálculo de un caminos entre una única pareja de nodos, entre un subconjunto de nodos a otro subconjunto, o incluso, de un subconjunto de nodos al resto del grafo. Las mejoras hechas sobre las variantes más generales (SSSP y APSP) normalmente pueden ser aplicadas sobre estos planteamientos derivados. Este capítulo de esta tesis se centra en el estudio de los algoritmos, secuenciales y paralelos, de estas variantes más generales.

El trabajo presentado incluye nuevas clasificaciones para los planteamientos paralelos en función de sus características, y hemos localizado algunas soluciones que no han sido exploradas que han sido el objetivo de estudio de esta tesis. Estos estudios han sido publicados en los siguientes artículos:

1. “Parallel Approaches to the Shortest Path Problem - A Survey,” H. Ortega-Arranz, Y. Torres, D. R. Llanos, and A. Gonzalez-Escribano, *To be submitted to ACM Computing Surveys*

2. “The Shortest Path Problem: Analysis and Comparison of Methods,” H. Ortega-Arranz, D. R. Llanos, A. Gonzalez-Escribano, *Book*, 1st edition, ser.(Synthesis Lectures on Theoretical Computer Science series), Morgan & Claypool.
   Online, DOI: 0.2200/S00618ED1V01Y201412TCS001 [20]

Capítulo 3: Un nuevo planteamiento para resolver el problema del SSSP usando GPUs

La alta capacidad de cómputo de los aceleradores hardware ha hecho que se dispare su explotación para obtener programas más rápidos. La programación de estos dispositivos se ha simplificado gracias a la aparición de lenguajes paralelos de alto nivel, como es el ejemplo de CUDA [21]. La aplicación de estos dispositivos para obtener mejoras dentro del contexto de los problemas de cálculo de caminos más cortos ha incrementado considerablemente durante los últimos años. Algunos ejemplos de estas implementaciones para las GPUs podemos encontrarlos en los trabajos de [22, 23], donde se utilizan modificaciones del algoritmo de Dijkstra.

Este capítulo presenta una nueva solución, basada en el algoritmo de Crauser [24], para resolver el problema del SSSP utilizando las GPUs. Esta solución supera al anterior planteamiento del estado del arte, propuesto por Martín *et al.* [23], en todas las familias de grafos evaluadas. Estas familias incluyen no sólo los grafos utilizados en sus propias experimentaciones, sino también casos de redes existentes en el mundo real. Utilizando los mismos valores para los parámetros de ejecución de las GPUs para ambas propuestas, nuestra solución funciona hasta 45 veces más rápido.

También hemos conseguido mejorar el tiempo de ejecución a través de un adecuado ajuste/tuning de los parámetros de ejecución particulares de los dispositivos GPU de NVIDIA, obteniendo hasta un 22.43% de mejora cuando se compara con la versión no ajustada/tuneada. Esta versión no tuneada utiliza los valores originales seleccionados por Martín *et al.* en su estudio. Además, hemos comparado nuestra versión mejorada con la versión secuencial optimizada procedente de la librería especializada en grafos Boost [25]. Nuestra versión funciona hasta 19 veces más rápido para algunas de las familias de grafos utilizadas consumiendo hasta 11.25 veces menos memoria.

Finalmente, también hemos realizado un estudio de las diferentes arquitecturas de los dispositivos GPUs de NVIDIA en función de las características del grafo a computar, dentro del contexto del problema del SSSP. Los resultados muestran como unas arquitecturas funcionan mejor en unos tipos de grafos, existiendo diferencias de hasta un 40.5% en los tiempos de ejecución.

El trabajo descrito se ha publicado en los siguientes artículos:

3. "Comprehensive Evaluation of a New GPU-based Approach to the Shortest Path Problem," H. Ortega-Arranz, Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos, *International Journal of Parallel Programming,* Springer, p. 1–21, 2015.
   Online, DOI: 10.1007/s10766-015-0351-z [26]

4. "A New GPU-based Approach to the Shortest Path Problem," H. Ortega-Arranz, Y. Torres, D. R. Llanos, and A. Gonzalez-Escribano, in *Proceedings of the 11th International Conference on High Performance Computing and Simulation*, ser.(HPCS'13), Helsinky, Finland: IEEE, 2013, pp. 505–511.
   Online, DOI: 10.1109/HPCSim.2013.6641461 [27].

Capítulo 4: Uso y extensión de un modelo para predecir configuraciones óptimas para la GPU aplicado al problema del APSP

Conseguir una implementación altamente paralela para GPUs es un trabajo asequible. Sin embargo, la optimización de los códigos ejecutados en estos aceleradores hardware representa todo un desafío. La razón principal de esta dificultad es el gran número de combinaciones con las que se puede ejecutar el mismo programa: numerosos valores para los parámetros de ejecución, decisiones de programación, técnicas de ajuste disponibles, etc. Existe una estrategia para poder atacar estos problemas de optimización de un modo sistemático: la caracterización de los kernels que se ejecutan en la GPU. Con esta caracterización es posible obtener una apropiada configuración para obtener mejores tiempos de ejecución sistemáticamente. Este capítulo utiliza estos criterios de caracterización para ajustar la ejecución de nuestra implementación propuesta para resolver el problema del

APSP. Esta implementación es una adaptación de nuestra solución del SSSP, propuesta en capítulo anterior, que resuelve el APSP a través del planteamiento por productividad $n \times SSSP$.

Hemos comprobado la validez del modelo de caracterización de kernels mediante una exhaustiva búsqueda en el espacio de soluciones, evaluando los valores más relevantes para los parámetros de ejecución de las GPUs de NVIDIA, y de las características de los grafos de entrada. También hemos evaluado su utilidad comparando los tiempos resultantes de utilizar la configuración predicha el modelo, con una de las configuraciones que sugiere la guía de programación de CUDA [21], obteniendo la primera una mejora de hasta un 62% con respecto a la segunda.

El trabajo descrito ha sido publicado en los siguientes artículos:

5. "Optimizing an APSP Implementation for NVIDIA GPUs Using Kernel Characterization Criteria", H. Ortega-Arranz, Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos, *The Journal of Supercomputing,* Springer, vol. 70, no. 2, pp. 786-798, 2014.
   Online, DOI: 10.1007/s11227-014-1212-z [28]

6. "A Tuned, Concurrent-Kernel Approach to Speed Up the APSP Problem," H. Ortega-Arranz, Y. Torres, D. R. Llanos, and A. Gonzalez-Escribano, in *Proceedings of the 13th International Conference on Computational and Mathematical Methods in Science and Engineering*, ser.(CMMSE'13), Almería, Spain: eds. I.P. Hamilton and J. Vigo-Aguilar, 2013, vol. 4, pp. 1114-1125.
   Online: CMMSE Proceedings [29]

Capítulo 5: Uso de sistemas heterogéneos y diferentes políticas de balanceo de carga para resolver el problema del APSP

Este capítulo estudia soluciones para el problema del APSP en grafos dispersos que combina el uso de algoritmos paralelos diseñados para GPUs junto con algoritmos secuenciales, a través de métodos de productividad paralelos diseñados para un sistema heterogéneo. Además, se han aplicado dos diferentes políticas de balanceo de carga para distribuir las tareas entre las diferentes unidades computacionales (CPUs y GPUs). La primera política divide equitativamente el espacio de trabajo entre todas las unidades computacionales independientemente de su naturaleza o capacidad de procesamiento. La segunda política mantiene una cola de trabajo de donde cada unidad computacional adquiere una nueva tarea cada vez que termina la anterior.

Hemos estudiado la importancia de este tipo de soluciones heterogéneas para grafos irregulares que dan lugar a dos tipos de tareas, pesadas y ligeras. Las tareas pesadas son procesadas más rápidamente por las GPUs que por las CPUs, mientras que con las tareas ligeras ocurre al revés. El primer estudio compara la resolución del APSP conociendo la distribución de la naturaleza de las tareas de los grafos para diferentes instancias de los dos tipos de políticas de distribución implementadas. El segundo estudio realiza la misma comparativa pero escogiendo tareas aleatorias para que la distribución sea desconocida. Se obtienen mejoras de algunas de las instancias heterogéneas de hasta un 65% y un 47%, para el primer y segundo estudio, comparado contra el uso de una única GPU

Los resultados también nos permiten concluir que el conocimiento de la naturaleza de los datos de entrada es muy importante porque permite al programador a asignar las tareas más costosas a los dispositivos con una mayor capacidad de cómputo. Esta información puede ser utilizada por las políticas equitativas para obtener los mejores tiempos de ejecución aunque su rendimiento se vería muy afectado por los cambios de la entrada. Por otro lado, las implementaciones que utilizan la cola de recuperación de trabajo tienen un rendimiento más estable porque son independientes de la naturaleza de los grafos de entrada.

El trabajo descrito ha sido publicado en los siguientes artículos:

7. "The All-Pair Shortest-Path Problem in Shared-Memory Heterogeneous Systems," H. Ortega-Arranz, Y. Torres, D. R. Llanos and A. Gonzalez-Escribano, in book *High-Performance Computing on Complex Environments,* ser. Series on Parallel and Distributed Computing. John Wiley & Sons, Inc., 2014, ch. 15, pp. 283-299. Online, DOI: 10.1002/9781118711897.ch15 [30]

Capítulo 6: Desarrollo de un modelo de programación multi-capa

Durante la última década las arquitecturas de procesamiento paralelo se han convertido en una poderosa herramienta para enfrentarse a problemas masivamente paralelos, como el APSP. Como muestra el capítulo anterior, la combinación de unidades computacionales de diferente naturaleza es la solución más prometedora para este tipo de problemas que requieren soluciones de cómputo de altas prestaciones (HPC). Pero también se ha visto que maximizar el rendimiento de los dispositivos como las GPUs requiere un extenso conocimiento en detalle de la arquitectura subyacente de estos aceleradores, convirtiéndose en una tediosa tarea manual asequible sólo para programadores experimentados.

Este capítulo presenta a TuCCompi, un modelo abstracto multi-capa que simplifica la programación de entornos heterogéneos que incluyen GPUs. Este modelo detecta automáticamente los diferentes recursos presentes en el clúster híbrido y trata de maximizar su explotación seleccionado la configuración más adecuada para el caso de los dispositivos GPUs gracias a la caracterización de los kernels que provee el programador. También se presenta la descripción de un prototipo de este modelo, junto con su evaluación en diferentes entornos heterogéneos, usando el problema del APSP como caso de estudio.

El trabajo descrito ha sido publicado en los siguientes artículos:

8. "TuCCompi: A Multi-Layer Model for Distributed Heterogeneous Computing with Tuning Capabilities," H. Ortega-Arranz, Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos, *International Journal of Parallel Programming,* Springer, p. 1–22, 2015. Online, DOI: 10.1007/s10766-015-0349-6 [31]

9. "TuCCompi: A Multi-Layer Programing Model for Heterogeneous Systems with Auto-Tuning Capabilities," H. Ortega-Arranz, Y. Torres, D. R. Llanos, and A. Gonzalez-Escribano, in *Proceedings of Workshop on High-level Programming for Heterogeneous and Hierarchical Parallel Systems*, ser.(HLPGPU'14), Vienna, Austria: HiPEAC 2014, pp. 18-25.
Online: HLPGPU Proceedings [32]

R.4 Conclusiones

El trabajo presentado en esta tesis doctoral nos permite responder a las preguntas de investigación propuestas con una respuesta afirmativa

(1) Gracias al estudio de la literatura sobre problemas de cálculo de caminos más cortos, hemos sido capaces de desarrollar una nueva solución paralela para el problema del SSSP, siguiendo las ideas propuestas por Crauser *et al.* [24], que es capaz de aprovechar la potente capacidad de cómputo de los dispositivos GPU. Esta implementación mejora el rendimiento de la anterior solución del estado-del-arte propuesta por Martín *et al.* [23]. Siguiendo las pautas propuestas en [28], hemos refinado el método de caracterización de kernels para obtener valores más adecuados para los parámetros de ejecución de las GPUs que conlleven a ejecuciones óptimas o cercanas al óptimo. La aplicación de esta metodología a nuestra implementación ha hecho que pudiéramos obtener valores más apropiados, que implicaron mejoras muy significativas en comparación con los valores recomendados por las guías de programación de CUDA [21].

(2) La combinación de diferentes modelos y lenguajes de programación nos ha permitido realizar implementaciones y novedosos estudios de soluciones, que estaban sin explorar, dentro del contexto de planteamientos basados en productividad paralela para el problema del APSP. Estas implementaciones abordan el uso de nuevas características de las GPUs para ejecutar de manera concurrente flujos de kernels diferentes, resolviendo cada uno de estos flujos distintos subproblemas SSSP en paralelo. Por otro lado, el uso combinado de CUDA y OpenMP permitió a otras implementaciones explotar al mismo tiempo ambas unidades computacionales presentes en el mismo sistema de memoria compartida, las GPUs y los núcleos de las CPUs. Por último, hemos propuesto un modelo de programación multicapa, y hemos desarrollado su prototipo con el que es posible coordinar de manera transparente todo un clúster heterogéneo de unidades computacionales de diferente naturaleza, utilizando internamente diferentes tecnologías como CUDA, OpenMP y MPI. Todos los experimentos llevados a cabo para esta novedosa formulación heterogénea demostraron que su uso conlleva mejoras en los rendimientos muy significativas comparadas contra las versiones homogéneas.

Chapter 1

Introduction

1.1 Motivation

Many problems that arise in real-world networks imply the computation of the shortest paths, and their distances, from a source to any destination point. Some examples include car navigation systems [1], traffic simulations [2], scheduled means of transport [3], logistic control [4], spatial databases [5, 6], Internet route planners [7], or web searching [8, 9]. Despite the importance of the shortest-path problem, algorithms to solve it are computationally costly, and in many cases commercial products implement heuristic approaches to generate approximate solutions instead. Although heuristics are usually faster and do not need much amount of data storage or precomputation, they do not guarantee the optimal route.

The aim of parallelizing these algorithms is, not only the immediate reduction of their execution time, but also their application in complex approaches, that use them as a step in their algorithms. In this way, this reduction makes their computation feasible where they were previously impracticable due to prohibitive temporal costs. An example related with the shortest-path context is the costly preprocessing phase of the modern methods and techniques used for routing in roadmaps. The preprocessing phase obtains values that will be stored and constantly used in later queries, where it is requested to calculate shortest paths, and their distances, between two vertices of the graphs. The preprocessing phase usually has costly execution times, but as long as it is going to be performed only once for long periods of time, it is an affordable cost. Thanks to these previous precomputed values, that in some cases, represent the calculation of shortest path distances between all vertices of the network, the query phase can be computed in the order of nanoseconds [10, 11].

The configuration of the roads of a map do not frequently change, so it could be plausible to pay the high temporal and spatial costs of computing these values once and store them. However, in other contexts, where this kind of static nature is not present in the network, or it is unknown, the reduction of these costs is highly significant. Additionally, in the previously mentioned context of route planning, the application of efficient parallel methods is compulsory if it is desired to take into account the current state of the roads in real-time, such as the information of the traffic, the road cuts due to maintenance or natural

events, or even the avoidance of low-speed/bad-quality roads for a particular driver.

The advent of mobile devices present a third challenge because of their small memory size, that limits the amount of precomputed data that can be stored. Most of the current approaches present a trade-off between the amount of memory used and the query time needed: The more memory used, the better the query time. However, the new devices nowadays have bigger computational capabilities, incorporating several processors that could alleviate this mentioned lack of memory, and therefore, giving a chance to faster parallel approaches instead of storing the data obtained through costly preprocessing.

1.1.1 Parallel Computing

Parallel computing, or also parallel processing, consists in using two or more devices for carrying out the computations at the same time, usually with the aim to reduce the high temporal costs needed in a single device. Although the application of parallel computing to complex problems does not lead to asymptotic-order reductions of the temporal costs of a problem, it frequently represents the only way to achieve solutions in competitive or reasonable execution times, and its use has proven to be critical when researching high performance solutions. Due to this fact, the evolution of computing not only has focused on developing faster processors, where the upper threshold of clock speeds is stalled for the moment, but also it has focused on the creation of new computer architectures involving more processing units, also improving the efficiency of the parallel computing. This architectural evolution has lead to two non-exclusive different trends, the multi-core and the many-core systems. Both have in common the huge increment of computational capabilities by the integration of several processing units.

The multi-core architecture refers to CPUs that contain two or more processing cores. These cores operate as separate all-purpose processors within a single chip. The use of multiple cores gets an increment of the total CPU performance without the need of raising the processor clock speed. Note that the concept of multiple cores is different than multiple CPUs. A multi-core computer can hold, for example, four cores on a single chip, whereas a multi-processor system may have four CPUs, each with only one core. Since the last trends favor energy savings, the high efficiency at low energy costs of these multi-core systems is imposing this architecture over the multi-processor architecture for domestic purposes. On the other hand, in academic or high performance contexts, the current machines combine both technologies, including multiple CPUs with multiple cores.

A system is considered as a many-core architecture when it has a number of processing units over dozens. However, with the fast increase of processing units of the computing systems, this value is subject to raise. In this category are included not only these supercomputers with a high number of CPUs with multiple cores, but also hardware accelerators, such as Graphics Processing Units (GPUs), or co-processors, such as the Xeon-Phi devices. Hardware accelerators are designed to integrate many specialized processing units to fulfill the need of processing high parallel applications in a high performance computing context.

Figure 1.1 shows a timeline evolution of the parallel computing from 1960, when the first supercomputing supporting this kind of computing was introduced, until these days, according to four different categories: Supercomputers, accelerators, mainstream

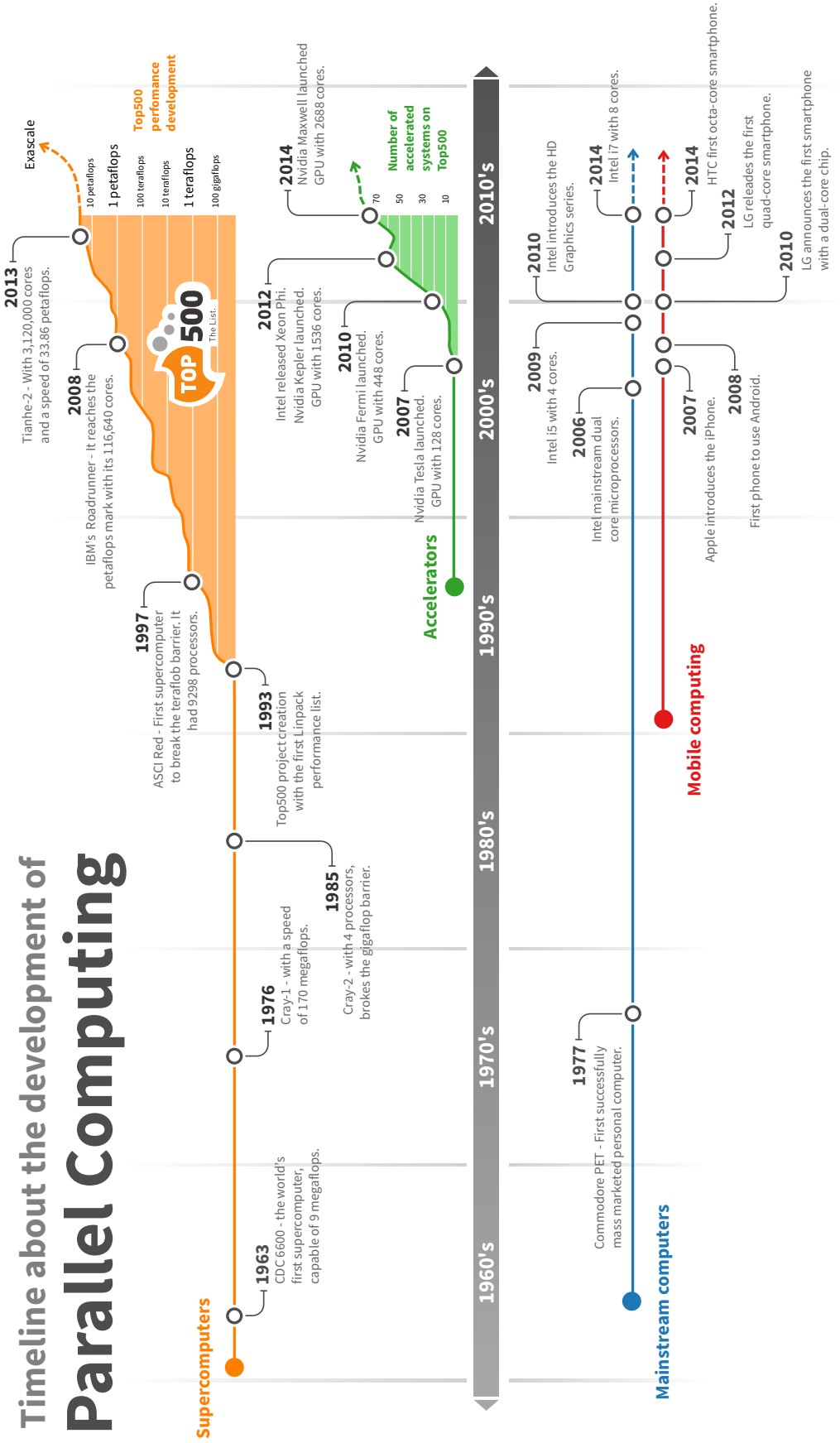


Figure 1.1: Evolution timeline of parallel computing.

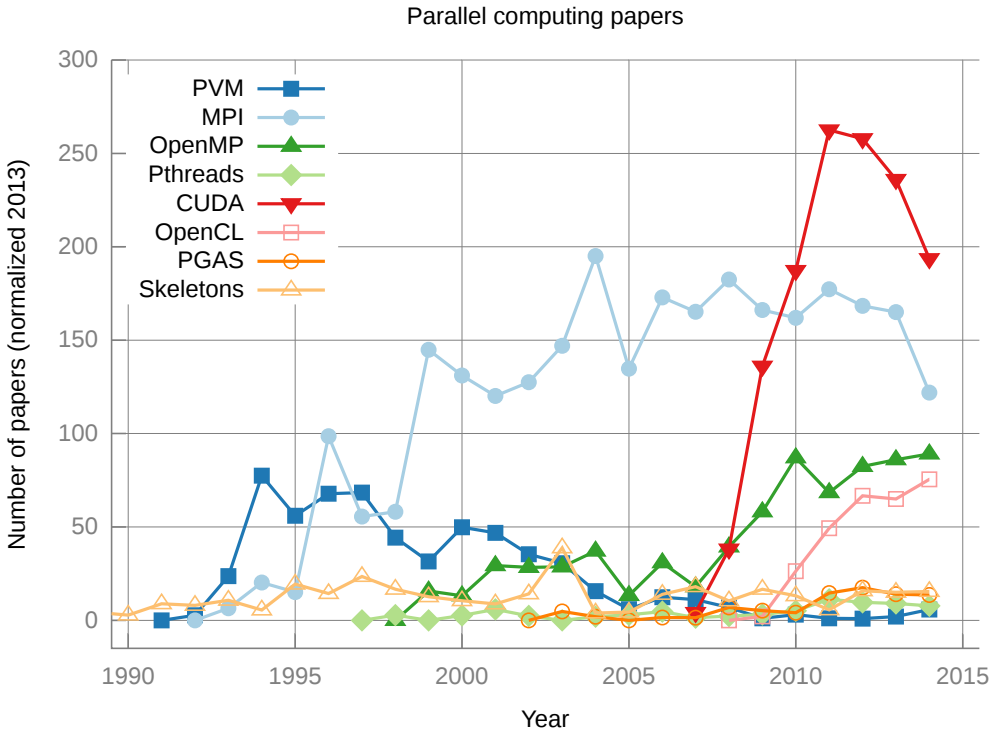


Figure 1.2: Number of papers related with different parallel computing technologies.

computers, and mobile computing, according to [33]. In spite of this impressive raise of computational capabilities, it is still a difficult task for the programmers to create optimal parallel solutions. These modern systems require to correctly manage some new practical issues, such as the data partitioning, data sharing, data transfers, the coordination and synchronization, or the migration to new computing paradigms as the SIMD model for some many-core systems. However, although programming in these devices it is not trivial, their use in the research activities is significantly growing, with the GPUs as the device for parallel computing that generates more scientific interest nowadays. Figure 1.2 depicts the trends of the scientific interest taken as the number of papers published in the literature inside the parallel computing category. These values have been obtained using the IEEE Xplore XML Gateway API [34].

1.1.2 GPUs for Parallel Computing

The most representative devices of the many-core architectures are the GPUs [12]. They were originally designed to help the CPU in processing the graphic data that will be display in the user's screen. However, due to their powerful computational capabilities, their use for different purposes became more popular, creating the trend of GPGPU (General-Purpose GPU) computing. In order to facilitate the programming on these devices, NVIDIA released in 2007 CUDA [13], a new programming model for GPGPU computing. Since then, the CUDA programming interface allowed to implement many sophisticated and efficient solutions for massively-parallel problems [14]. During these

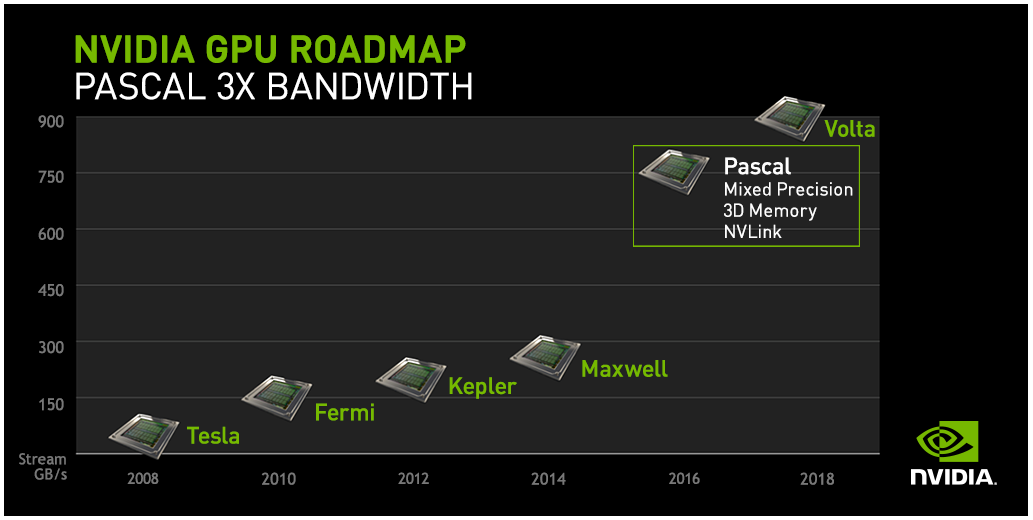


Figure 1.3: Bandwidth estimated for future architectures of NVIDIA GPU devices.

decades, the computational capabilities and resources available for GPU computing have been increased in an exponential fashion, and the plans of NVIDIA include to make these enhancements even more bigger for the future GPU architectures (see Fig. 1.3).

The implementations developed using the CUDA programming model are only deployable on NVIDIA GPUs devices. On the other hand, there are other programming languages that create an unified interface for all kinds of GPUs that support GPGPU computing. This is the case of BrookGPU [35], developed at Stanford's University, and OpenCL [36], that constitutes a standard API for programming, not only GPUs but also multi-core CPUs. Although the portability offered by these general approaches is key to exploit both GPUs and CPUs with less effort, NVIDIA continues releasing new and more competitive versions of its CUDA compiler and toolkits. This is due to the fact that there are many advanced features and configurations on the NVIDIA boards that can only be manipulated by using this programming model, and a correct handling of these parameters leads to optimized implementations that offer a significantly increment in terms of performance.

Depending on the particular application, it is relatively easy to develop a naïve GPU solution that returns good speedups, thanks to the high-level abstraction offered by the programming model described before. Nevertheless, to correctly tune the code in order to efficiently exploit all underlying GPU resources is a difficult task. It is necessary to have an in-depth knowledge of the device architecture and its resource management in order to predict its behavior, and tune up the configurations for an optimal performance. This responsibility lies directly on the programmers, that need to provide several values for the GPU running parameters, such as the threadblock size and shape for each different GPU function, or for the management of the L1 cache memory, among other examples.

The CUDA programming guidelines suggest the use of threadblock sizes that maximize the occupancy, for obtaining a good performance. However, some studies [15, 16] have shown that, in some cases, these values recommended by CUDA do not always lead to the optimum performance, leaving to the programmers the task of searching for the best values through time-consuming, trial-and-error tests.

1.1.3 Heterogeneous Computing

Heterogeneous computing [17, 18] refers to computing environments or systems that are built with computational devices of different nature. Some classic examples are conventional shared- or distributed-memory multi-processors containing multi-core CPUs, together with GPU devices. In contrast, in the traditional "homogeneous computing" all computational devices have the same nature and similar processing features.

One of the main advantages of using the heterogeneous computing paradigm, against the homogeneous model, is to allow the assignment of a particular kind of functions/tasks to the computational devices that better fits its processing requirements. Another important advantage is the possibility of increasing the processing capabilities of the system by taking advantage of every computational device present in the system, even if the features of the computation do not fulfill the optimal conditions for the best performance on some of these devices.

However, the programming of heterogeneous systems results in a considerable bigger effort compared with the programming of traditional homogeneous systems. The programmer should know the programming models and languages needed to take advantage of each different computational device, their limitations and particularities, in order to provide a different implementation of the part of the computation to be executed on each one. More difficulties appear when trying to achieve a proper management of the computational environment in order to obtain an optimum performance for a particular problem. This implies that the programmer knows which functions are better to be executed in which kind of computational device, and also, how much workload he should distribute between the different devices. All these problems make the programming for heterogeneous system a piece of craftsmanship, since there is a lack of general-purpose programming frameworks that handle this complexity in a transparent way.

1.2 Objectives of the dissertation

According to the identified problems described in the previous section, the research questions to be solved in this Ph.D. thesis is the following:

> *Is it possible to develop techniques and tools to derive more efficient parallel implementations to solve Shortest Path problems using: (1) The new modern Graphics Processor Units (GPUs) and their corresponding optimization tuning techniques, (2) heterogeneous environments composed by these hardware accelerators together with the use of traditional CPUs?*

1.2.1 Research Methodology

The research methodology carried out in this Ph.D. thesis is based on the software engineering method that has four different phases: Observe existing solutions; propose better solutions; build or develop these new solution; and measure and analyze its results [19]. This is an iterative methodology that is repeated to refine the solutions. It resembles the stages of the classical scientific method: Propose a question; formulate hypothesis; make predictions; and validate hypothesis.

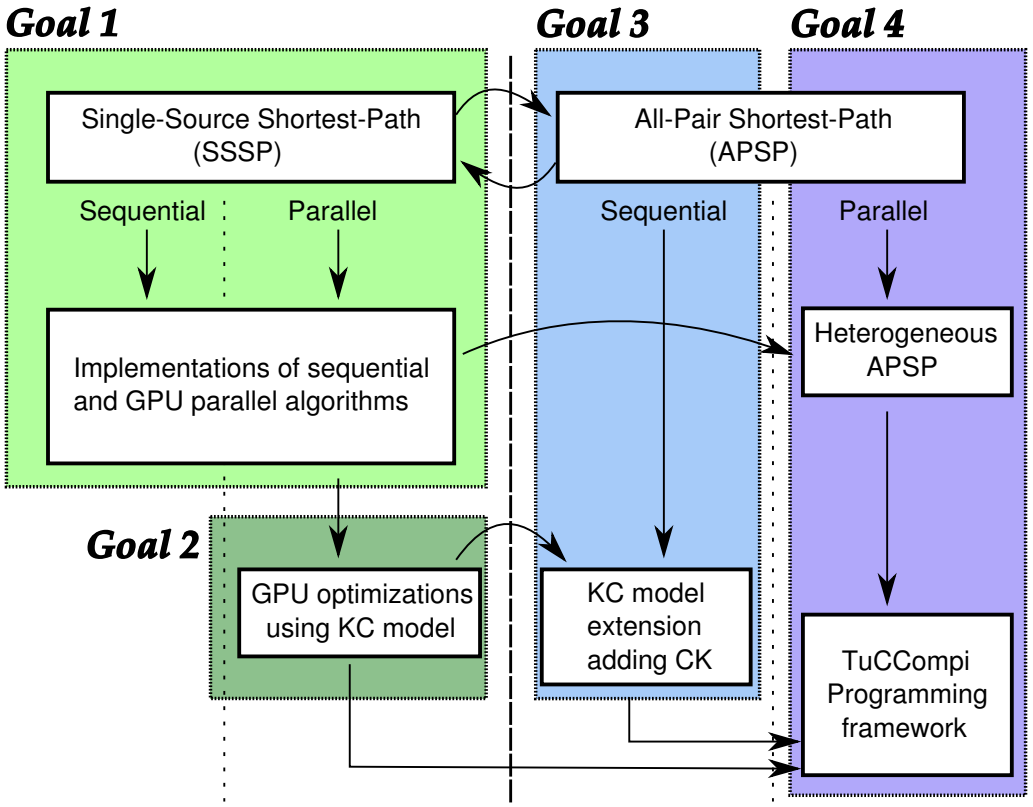


Figure 1.4: Goals and subgoals to be accomplished in this Ph.D. thesis.

1. *Observe existing solutions*. This is an exploratory phase where the related literature should be thoroughly analyzed in order to detect, not only the limitations that will be addressed during the research process, but also possible improvements and/or new solutions not contemplated yet.

2. *Propose better solutions*. This phase is dedicated to the analysis and design to find better solutions trying to overcome the limits or taking advantage of the improvement chances previously detected.

3. *Build or develop the solution*. The research methodology of this phase consists of building a prototype in order to demonstrate that the proposed solutions are feasible.

4. *Measure and analyze the new solution*. In the last phase of this research methodology, the implemented prototypes of the solutions are evaluated through experimental work, with the aim of corroborating whether they solve the problems discovered in the first phase.

1.2.2 Milestones

In order to be able to answer these research questions, we have accomplished the following subgoals (see Fig. 1.4). For each one of them, the phases of the research methodology previously described have been carried out.

Goal 1: Develop a new approach for the Single-Source Shortest-Path problem.

$(Observation)$ A complete study of the previous work is done, revisiting first the sequential solutions, and proceeding to their corresponding parallel variants, for the Single-Source Shortest-Path (SSSP) problem. After this analysis, the scope of the research is focused on algorithms for non-dense graphs.

$\binom{Proposal\ and}{developing}$ Combining different ideas from the literature, a new algorithm for GPUs will be proposed. We will implement the proposed algorithm using CUDA for the parallel version to be optimized and executed for the NVIDIA GPUs.

$(Measures)$ The feasibility of our proposal will be evaluated comparing it with the last state-of-the-art approach for GPUs, and also with a reference sequential version, using a suite of synthetic and real-world benchmarks. In both cases, significant speedups are expected to validate the proposal.

Goal 2: Optimize our implementation with a proper tuning of the GPU execution parameters.

$(Observation)$ The deployment of implementations in GPUs implies the choice of many parameters, such as the threadblock sizes, or the activation state and size of the L1 cache memory, whose proper tuning may lead to important performance improvements. A revision of the literature should be done in order to find some guidelines that can help the programmer with these important decisions, in order to enhance our solution.

$\binom{Proposal\ and}{developing}$ We should find a model that provides some guidelines for properly tuning these GPU parameters. Ideally, the model should be based on some features of the kernels implemented in our algorithm, known as kernel characterization (KC). After using these criteria to analyze our code, the model should return some values considered as optimal/near-optimal for the target GPU.

$(Measures)$ We will carry out experiments using both tuned and non-tuned versions, in order to check the usefulness of the values predicted by the model.

Goal 3: Solve the All-Pair Shortest-Path problem incorporating modern features of the new GPUs, and extend the kernel characterization model with them.

$(Observation)$ A complete study of the previous work is done, revisiting first the sequential solutions, and proceeding to their corresponding parallel variants, for the All-Pair Shortest-Path (APSP) problems. After this analysis, the scope of the research is focused on productivity-based parallelization approaches.

The modern GPUs has new features as the Concurrent-Kernel execution (CK) that allows to these devices to execute more than one kernel concurrently if they have enough free resources to compute them. A study of the functionality and behavior of this feature leads to a better understanding the computational capabilities of these devices, and thus, to a new approach that takes advantage of it.

$\binom{Proposal\ and}{developing}$ We will modify our SSSP solution in order to support the concurrent kernel execution. This technique will allow to use a different stream of kernels for a SSSP with a different source node, using the parallel APSP strategy known as the $n \times SSSP$ approach.

We will refine an existing GPU parameter tuning model [16], and extend it for the shortest path context, considering both the graph properties of the input sets, and the concurrent-kernel execution of the modern GPUs.

$(Measures)$ We will carry out a complete set of experiments with different GPU architectures in order to verify the correctness of both the shortest path distances returned by the new APSP approach, and the predicted values returned by the new extended GPU parameter tuning model.

Goal 4: Explore the exploitation of heterogeneous environments for the All-Pair Shortest Path problem.

$(Observation)$ During a previous exploration of the literature related with parallel approaches that solve the APSP problem, it was not found studies that combine parallel algorithms with parallel productivity-based methods.

$\binom{Proposal\ and}{developing}$ We propose the combination of both sequential and GPU parallel versions into the same implementation in order to solve the APSP problem using computational devices of different nature concurrently. This will lead to a new approach that, to the best of our knowledge, has not been explored in the literature. This implementation will use the computational devices of a shared-memory system, assigning to each one its corresponding version: The sequential one for the CPU cores, and the GPU parallel version for those CPU cores responsible of handling a GPU. Additionally, we will enrich the solution with different load-balancing techniques.

Finally, we want to create a new programming model incorporating this novel approach, combining also the knowledge obtained from empirical study regarding the predictions for tuning the GPU parameters and the concurrent kernel execution. Therefore, we will develop a prototype for this proposed programming framework that integrates all these contributions together.

$(Measures)$ We will carry out several experiments using graphs with different properties, and different load-balancing techniques, in order to evaluate the efficiency of the new approach against the traditional use of only one computational device at a time. Finally, we will test the prototype of the programming framework by solving the APSP problem in different heterogeneous clusters.

1.3 Document Structure

This document is organized as follows (see Fig. 1.5). Chapter 2 presents a survey of the state of the art of the sequential and parallel algorithms that solve the SSSP and the APSP

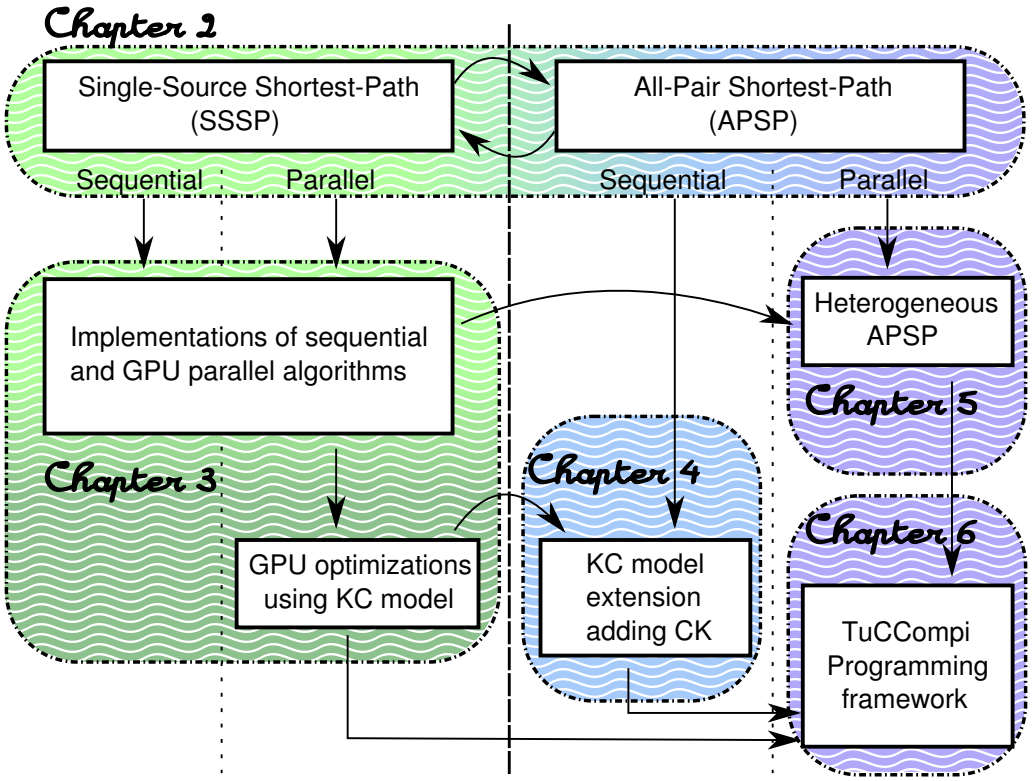


Figure 1.5: Document structure.

problems, and a real-world application example of these methods. Chapter 3 describes in depth a previous state-of-the-art algorithm for the SSSP problem in GPUs, and our new implementation, together with an optimized variant. Chapter 4 introduces a extended model that predicts optimized values for GPUs using the kernel characterization criteria. Chapter 5 presents a solution for the APSP problem, combining parallel algorithms and parallel-productivity methods in a heterogeneous system, together with different load-balancing techniques. Chapter 6 describes a programming framework that simplifies the programming on heterogeneous systems including hardware accelerators, by hiding the details of synchronization, deployment, and tuning. Chapter 7 contains the conclusions of this Ph.D. thesis enumerating its contributions and the corresponding publications.

Chapter 2

State of the Art of the Shortest-Path Problem

The objective of this kind of problems is to find the shortest possible path that connects one or more source nodes with one or more target vertices, and compute their distances. Depending on the required solution, there are different algorithms for the different variants of the Shortest-Path problem. The algorithms that solve the Single-Source Shortest-Path (SSSP) problem compute the shortest paths between a specific source node and the remaining nodes of the graph. The solutions for the All-Pair Shortest-Path (APSP) problem look for all shortest path between all pair of nodes of the graphs. Furthermore, more particular approach derivations of these two general variants can be found when computing shortest paths: just from a specific source node to a particular target node, One-Pair Shortest-Path (OPSP), from many source nodes to the remaining graph vertices, Multi-Source Shortest-Path (MSSP), or even, from many sources nodes to just a particular set of target nodes, Many-to-Many Shortest-Path (MMSP). The improvements made in the general solutions can be ported also to the derived approaches. Thus, the research of this Ph.D. thesis is focused on the first two variants, the SSSP and the APSP problem.

This chapter studies the different algorithms and solutions for both problems, and also their parallel approaches. Additionally, these algorithms are gathered following some proposed criteria, describing new taxonomies or classical classifications. The review shown in this chapter is made with best efforts; nevertheless, due to the huge workspace of approaches, few techniques and particular solutions could not be listed here.

2.1 Brief Introduction to Graph Theory

In this section we present a summary of graph theory concepts, focused on the ones related to the shortest-path problem. More general introductions to this topic can be found in classical books such as [37] or [38].

The easiest way to represent the information of a network is to represent it as a graph, where every link will be an edge, and every possible joint to change to another link will be a vertex. Some groups of vertices could represent cities, stations or only intersection points, depending on the scenario to be considered.

Definition 1. *Let V be a set of elements called **vertices**. An **edge** is a tuple (u, v) that represents a link between the vertices u, $v \in V$, The vertices that are connected through an edge are called **adjacent**.*

Every vertex v is usually depicted as a point in the graph, whereas every edge e is usually depicted as a line that connects two and only two vertices. A collection of nodes connected by edges, that represents some kind of entity or networks, such as roadmaps, Internet, or social networks, among others, is called a graph.

Definition 2. *A **graph** G is a pair $G = (V, E)$ composed by a set of vertices V, also called nodes, and a set of edges E, also called arcs. A graph $G' = (V', E')$ is a **subgraph** of graph $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$.*

Definition 3. *Let n be the **number of nodes** of the graph, $n = |V|$, and m the **number of edges** of the graph, $m = |E|$.*

There are some graphs whose edges (u, v) have a restricted direction or connection, allowing only the traversing from node u to node v and not vice versa. They are called *directed graphs*, or simply *digraphs*. On the other hand, if all edges of the graph can be traversed in both directions, they are called *undirected graphs*.

Definition 4. *A **directed graph**, or **digraph**, is a graph $G = \{(V, E)/(u, v) \neq (v, u) : (u, v), (v, u) \in E\}$ where the edge (u, v), that connects node u with node v, only can be traversed from u to v, and therefore is different from edge (v, u).*

Definition 5. *An **undirected graph** is a graph $G = \{(V, E)/(u, v) = (v, u) : (u, v), (v, u) \in E\}$ where all edges can be traversed in both directions.*

This last discrimination is important because in a directed graph the link between node u and node v can exists, but not its corresponding backward connection. Some current examples of these different behaviors can be found in social networks, such as Facebook and Twitter. In the former one, the users are represented as nodes, and the friendship between these users as links. Friendship in this context is a two-way relationship that involves a two-way connection, whereas in the latter one, the users, represented also as nodes, can follow other users without compulsory being followed by them, a one-way connection. Additionally, the cost of traversing an edge from one node to its corresponding adjacent node can be different from its backwards traversing. Some examples can be found in road networks, when a particular one-way road may cross a mountain thanks to a tunnel, while the opposite-way road goes up and down the mountain involving many curves, with a very different cost.

Definition 6. *An **edge-weighted graph** $G = (V, E, w)$, or simply a **weighted graph**, is a graph $G = (V, E)$ which has a weight function associated to each edge, $w : E \to \mathbb{R}$. This function represents the cost of traversing the edge. A graph that is not associated with a weight function is known as an **edge-unweighted graph**, or simply an **unweighted graph**.*

Each node of the graph can be connected to a different number of other nodes through different edges. Actually, a node can be isolated from the network, being not connected to any other node. The mean number of connections of a node represents a very important

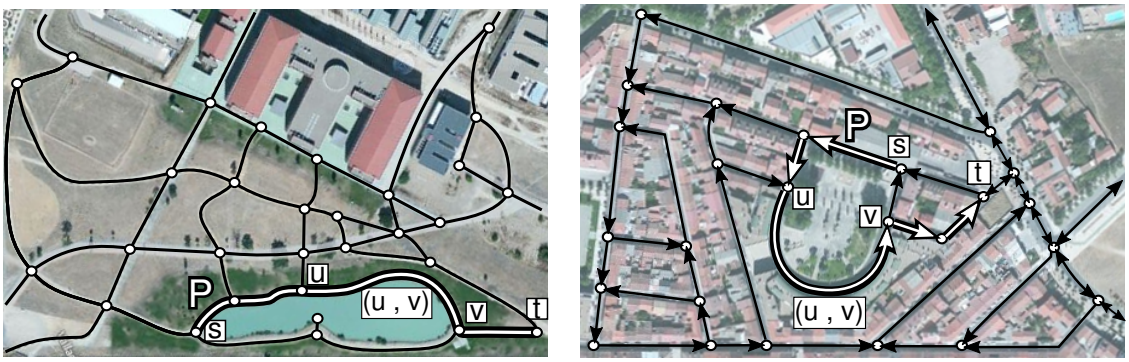


Figure 2.1: Examples of paths in (a) an undirected graph and (b) a directed graph.

feature of the graph. Graphs can be classified as *sparse* and *dense* graphs, depending on the proportion between the cardinalities of the two sets that compound the graph, the vertex set V and the edge set E. Moreover, if the graph is directed, more properties can be defined to distinguish between outgoing and incoming connections.

Definition 7. *A graph is considered as a **sparse graph** if the number of edges is in the order of the number of vertices, $m \in O(n)$. On the other hand, a graph is considered as a **dense graph** when $m \in \Theta(n^2)$.*

Definition 8. *The **degree** of a node v is the number of edges connected to it. For digraphs, the **fan-out degree** of a node u, $d^+(u)$, represents the number of edges leaving a vertex u, whereas the **fan-in degree** of a node u, $d^-(u)$, is the number of incoming edges towards a vertex u.*

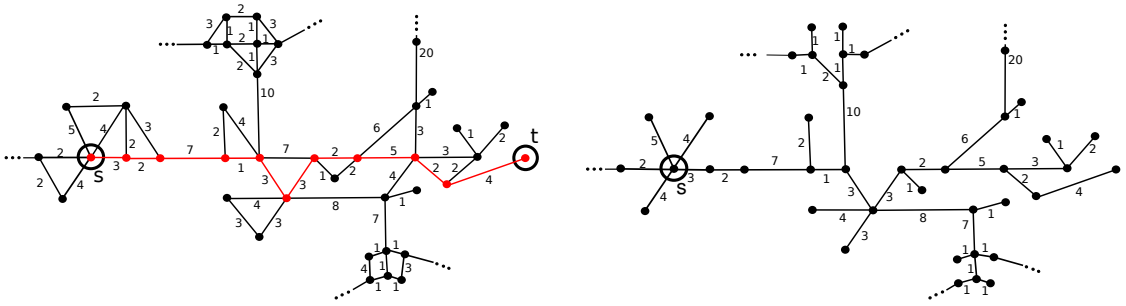
Definition 9. *The **successors** of a vertex u, $Succ(u)$, are the set of nodes that can be reached from node u using one of its outgoing edges. The **predecessors** of a vertex v, $Pred(v)$, are the set of nodes that can reach the node v using one of its incoming edges.*

Figure 2.1 shows two examples of graphs, one undirected and one directed. The node s in the undirected graph (a), has *degree* three because it is connected with other three vertices. On the other graph (b), the node s has a fan-out degree of one, and a fan-in degree of two, because it has only one outgoing edge and two incoming ones, respectively. One of its incoming edges is from node v. This node v is a predecessor of node s. Respectively, node v is a successor of node u.

Definition 10. *A **path** $P = \langle s, ..., u, ..., v, ..., t \rangle$ is a sequence of vertices connected by edges, from a source vertex s to a target one t. A path $P' = \langle u, ..., v \rangle$ is a **subpath** of P if its sequence of vertices and edges are contained in P with the same ordering.*

Definition 11. *The **weight** of a path, $w(P)$, is the sum of all the weights associated to the edges involved in the path.*

Definition 12. *An undirected graph is considered as a **connected graph** if there exists at least one path connecting for each pair of nodes (u, v) of the graph. A directed graph is*



(a) (b)

Figure 2.2: Examples of (a) the shortest path from node s to node t, highlighted in red, and (b) the shortest path tree from node s.

*considered as a **connected digraph** if there exists at least one path connecting each pair of nodes (u, v) of the graph in any direction, either backwards or forwards. A digraph is **strongly connected** if there exists paths connecting each pair of nodes (u, v) of the graph in both directions, from u to v and also from v to u.*

Any person, when leaves his/her home towards a destination, is describing a path in his/her way traversing different streets. In a similar way, the data packages in Internet depict paths by crossing different routers to reach their target computer. The carrier companies, with delivery functions, also perform routes from a specific source to different locations. However, in most cases, it is desired that this "package transportation" is carried out without wasting more resources than necessary. Thus, the path to be traversed should be the one with the lowest possible cost.

Definition 13. *The **shortest path** between two vertices s and t is the path with the minimum weight among all possible paths between s and t. The **minimum distance** between s and t, $d(s, t)$ or simply $d(t)$ when talking about a source node s, is the weight of the shortest path between them.*

An example of a shortest path between two vertices is depicted in Fig. 2.2 (a), where the shortest path from node s to node t is highlighted in gray. This path has a total weight of $w(P) = 32$, that is the shortest path distance. There are many algorithms whose objective is to compute the shortest path, and its cost. Most of them usually carry out some computations that calculate tentative costs, that can be reduced if possible, until they are "settled" as final shortest-path distances.

Definition 14. *We denote by $\boldsymbol{\delta(s, t)}$, or simply by $\boldsymbol{\delta(t)}$, the temporal tentative distance between s and t during the computation of $d(t)$.*

Nevertheless, there are many situations where it is desired to calculate, from one source, the shortest paths not only to a single target but also to the whole network locations.

Definition 15. *The **shortest path tree** of a graph from source node s is the composition of every shortest path from s to the remaining nodes.*

Weight / type	Algorithm	Time complexity
Unweighted	BFS [43]	$O(m+n)$
$\mathbb{R}_{\geq 0}$	Dijkstra [44]	$O(m+n\log n)$
$\mathbb{R}_{\{1..C\}}$	Goldberg [45]	$O(m+n\log C)$
$\mathbb{R}$ / Und	Pettie [46]	$O(m+n\log\log n)$
$\mathbb{R}$	Bellman-Ford [47, 48]	$O(mn)$
$\mathbb{Z}_{\geq 0}$ / Und	Thorup [40, 49]	$O(m+n)$
$\mathbb{Z}_{\{0..C\}}$ / Dir	Thorup [50]	$O(m+n\log\log min\{n, C\})$
$\mathbb{Z}_{\notin\{0,1\}}$	Goldberg [51]	$O(m\sqrt{n}\log min\{w(e) : e \in E\})$

Table 2.1: Best-bounds SSSP algorithms for different types of graphs and edge-weights restrictions. The abbreviations *Und* and *Dir* refer to undirected and directed graphs respectively.

Figure 2.2 (b) shows the shortest path tree computed from node s, involving all shortest paths to the remaining vertices of the graph.

The following section describes this classical graph problem of computing all shortest path and their distances from a source node, together with a taxonomy of solutions for the Single-Source Shortest-Path problem, and also including the explanation of one of the most relevant algorithms, the Dijkstra algorithm.

2.2 The Single-Source Shortest-Path (SSSP) Problem

The Single-Source Shortest-Path (SSSP) problem consists on computing the shortest paths, from a source vertex s, to every vertex $v \in V$ that is reachable from s. If v is unreachable from s, then $d(s, v) = \infty$. SSSP algorithms usually apply iterative methods that label the vertices with a tentative distance from s, $\delta(v)$, found so far. These tentative distances can be updated with a lower value if a new path is found through another vertex u that is shorter than the previous one. That is, relaxing the edge (u, v), $\delta(v)$ is updated if $\delta(v) > \delta(u) + w(u, v)$.

In this section, a brief review of the classical solutions to the SSSP problem is presented, including the description of Dijkstra's algorithm.

2.2.1 Taxonomy of SSSP Algorithms

The SSSP algorithms can be classified in two main categories: *Label-setting algorithms*, and *label-correcting algorithms*. Table 2.1 shows some of the best time-bound algorithms for different types of graphs and edge-weights restrictions. See [39, 40, 41, 42] for more details.

Label-setting algorithms

This kind of algorithms settles the tentative distance of a selected vertex v as the minimum distance from the source node s at each iteration. This distance is the optimal one and it will not be updated any more. After that, the algorithm traverses the outgoing edges of

Algorithm 1 The generic label-correcting algorithm

1: **for** $\forall v \in V$ **do**
2: $\quad \delta(v) = \infty$;
3: $\quad \delta(s) = 0$; $predecessor(s) = 0$;
4: **end for**
5: **while** $\exists$ edge $e = (u, v) : \delta(v) > \delta(u) + w(e)$ **do**
6: $\quad \delta(v) = \delta(u) + w(e)$;
7: $\quad predecessor(v) = u$;
8: **end while**

the selected vertex trying to update the $\delta(u_i)$ of its adjacent vertices u_i. Hence, at most n iterations are required to settle all vertices in the graph.

The basic label-setting approach is Dijkstra's algorithm [44]. In summary, it handles three subsets of V: *Settled*, *reached* and *unreached* vertices. The settled vertices have found the optimal distance value, $d(v) = \delta(v)$. The reached vertices have a finite $\delta(v)$, whereas the unreached vertices have a $\delta(v) = \infty$. Initially, the algorithm settles s with $d(s) = 0$, it traverses its outgoing edges reaching new vertices v_i, if necessary it updates their $\delta(v_i)$. The process is repeated, settling the vertex with the minimum distance label from the reached subset, until it is empty. Its time complexity is in $O(n^2)$ [43]. If the graph is sufficiently sparse, this bounds can be improved to $O(m + n \log n)$ using Fibonacci heaps [52].

Another kind of data-structure used is the so-called *buckets*. They used a parameter Δ, named *bucket width*. A bucket is a linear array B where each B_i stores the set of vertices whose $\delta(v) \in [i \cdot \Delta, (i + 1) \cdot \Delta)$. Having $\Delta \leq \Delta_0 = min\{w(e) : e \in E\}$, the vertices in the first non-empty bucket can be settled in any order. If $\Delta > \Delta_0$, we need to find the smallest $\delta(v)$ in the bucket to preserve the label-setting nature of the algorithm. Alternative bucket approaches include nested (multiple levels) buckets and/or buckets of different widths [53].

Label-correcting algorithms

In label-correcting algorithms, all distances are considered tentative until the final iteration, when all of them become settled. A generic SSSP label-correcting algorithm selects an arbitrary edge $e = (u, v)$ that does not fulfill the optimality condition $\delta(v) > \delta(u) + w(e)$, and updates $\delta(v)$ appropriately. It repeats the process until there are not more edges that violate the optimality condition.

The time cost depends on the order of edge relaxations. The worst case would be a relaxing order in where all edges always violate the optimality condition. For graphs with integer weights bounded by a constant C ($w(v) \in [1, C]$), it has a cost of $O(n^2 \cdot m \cdot C)$, and $O(m \cdot 2^n)$ otherwise [54].

The classic Bellman-Ford version [47, 48] is an efficient implementation of the generic label-correcting algorithm. Its algorithm establishes an order for the edges of a graph, and relaxes them one by one in the same order at each iteration, updating distance labels if necessary, until none of them changes during a complete pass. Each iteration requires $O(m)$ relaxations and it needs $n - 1$ passes. Therefore, it has a running time in $O(mn)$.

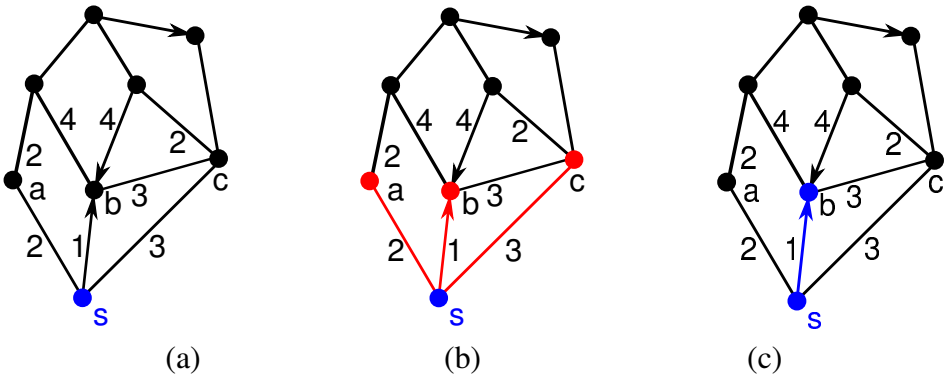


Figure 2.3: Dijkstra's algorithm steps: Initialization (a), edge relaxation (b), and settlement (c).

Improved label-correcting algorithms add to a set Q those vertices whose tentative distance label $\delta(v)$ has been decreased. After this, they select a node u from Q in each pass, relaxing only its outgoing edges and updating tentative distances if necessary, instead of computing all edges again.

There also exists a label-correcting version of the bucket approach. This version chooses $\Delta > \Delta_0$ and extracts all nodes from the first non-empty bucket. However, the vertices that were previously extracted, and whose tentative distance is now updated with a lower value, should be stored again in the corresponding bucket to be processed with its new value. These approaches are known as the *approximate bucket* implementation of Dijkstra's algorithm.

2.2.2 Dijkstra's Algorithm

The best well-known solution for the SSSP problem is Dijkstra's algorithm [44]. This section describes it in detail as well as several relevant implementation choices.

Dijkstra's algorithm constructs minimal paths from a source node s to the remaining nodes, exploring adjacent nodes following a proximity criterion. This exploring process is known as *edge relaxation*. When an edge (u, v) is relaxed from a node u, it is said that node v has been *reached*. Therefore, there is a path from s through u to reach v with a tentative shortest distance. Node v is considered *settled* when the algorithm finds the shortest path from source node s to v. The algorithm finishes when all nodes are settled.

The algorithm uses an array, D, that stores all tentative distances found from the source node, s, to the rest of the nodes. At the beginning of the algorithm, every node is unreached and any distances are known, so $D[i] = \infty$ for all nodes i, except the current source node $D[s] = 0$. Note that the reached nodes, that have not been settled yet, and the unreached nodes are considered unsettled nodes. The algorithm proceeds as follows (see Fig. 2.3):

1. (Initialization) It starts on the source node s, initializing the distance array $D[i] = \infty$ for all nodes i and $D[s] = 0$. Node s is settled, and is considered as the *frontier node* f ($f \leftarrow s$), the starting node for the edge relaxation.

2. (Edge relaxation) For every node v adjacent to f that has not been settled (nodes a, b, and c of Fig. 2.3), a new distance from source node s is found using the path through f, with value $D[f] + w(f, v)$. If this distance is smaller than the previous value $D[v]$, then $D[v] \leftarrow D[f] + w(f, v)$.

3. (Settlement) The non-settled node b with the minimal value in D is taken as the new frontier node ($f \leftarrow b$), and it is now considered as settled.

4. (Termination criterion) If all nodes have been settled, the algorithm finishes. Otherwise, the algorithm proceeds once more to step 2.

In order to recover the path, every reached node stores its predecessor, so at the end of the query phase the algorithm just runs back from the target node, through these stored predecessors, till the source node is reached.

Regarding the complexity of Dijkstra's approach, the initialization has a cost in $O(n)$. The updating operation, that has a constant cost, is performed m times in the worst case. The linear search in D for the node u with the lowest value (step 3) is performed in $O(n)$. Since it should be done n times in the worst case, this leads to a total upper bound of $O(n + m + n^2) \in O(n^2)$ [43].

Dijkstra with priority queues

The most efficient implementations of Dijkstra's algorithm for sparse graphs, that are graphs with $m << n^2$, use a priority queue to store the reached nodes [43]. The use of this data structure helps to reduce the asymptomatic behavior of Dijkstra's algorithm. To implement a priority queue, the following operations are needed:

- **Insert with priority:** The structure will be sorted with respect to a key, called the priority. The insertions of new elements are done according with their priority, to keep the structure sorted in non-decreasing order. In our case, the key associated to each node is the total distance found so far from the source node.

- **Update element's priority:** This operation changes the value of the key of an element and reorders the queue.

- **Find minimum element:** This operation finds the element of the structure with the lowest priority (key), in our case the lowest distance.

- **Delete minimum element:** This operation, also called as *pop* operation, deletes the element from the structure with the lowest priority.

If traditional binary heaps are used, such as Williams' heaps [55], each inserting, deleting or updating operation takes a time in $O(\log n)$. Selecting the next node to be processed (the current node) simply implies to find the minimum element, with a temporal cost in $O(1)$, and to delete it, with a cost in $O(\log n)$. Implementing the priority queue with binary heaps decreases the algorithm's asymptotic time complexity to $O((m + n) \log n) \in O(m \log n)$. If Fredman and Tarjan's Fibonacci heaps [52] are used instead, this time complexity is reduced to $O(n \log n + m)$. The Relaxed heaps [56] achieve the same amortized time bounds as the Fibonacci heaps with fewer restrictions. For a thoroughly review of the use of priority queues see [57, 58].

Algorithm	Year	Parallelization
Fine-Grain Parallel SSSP [59]	1997	Inner
Crauser [24]	1998	Outer
Δ-stepping [60, 42]	2003	Outer
GPU Label-Correcting [61, 22]	2007	Outer
GPU Label-Setting_F$_{\Delta=0}$ [23]	2009	Outer
GPU Label-Setting_U$_{\Delta=0}$ [23]	2009	Outer
GPU Parallel Bellman-Ford [62]	2011	Inner-Outter
Coarse-Grain Parallel SSSP [59]	1997	Disjoint
Tang [63]	2008	Disjoint

Table 2.2: Parallel SSSP algorithm classification.

2.3 Parallel Solutions for the SSSP (Π-SSSP)

The greek term Παράλληλος was used to refer to some actions that were performed at the same time, or concurrently. As this is the case for the following algorithms that are described in this section, we wanted to use the notation Π-SSSP, for those parallel algorithms that solve the SSSP problem. We have used the same notation rule for other contexts with the same meaning during the dissertation of this Ph.D. thesis, such as Π-APSP to refer to parallel APSP algorithms.

We can distinguish between two kinds of parallel strategies that can be applied around the SSSP algorithms. The first parallelizes the sequential SSSP algorithm in its internal operations, and the second performs several SSSP algorithms in parallel in disjoint subgraphs. The following paragraphs describe an overview of both strategies in more detail.

2.3.1 Parallelizing the Internal Operations of the SSSP Algorithm

The key of the parallelization of a single sequential Dijkstra algorithm is the inherent parallelism of its loops. For each iteration of Dijkstra's algorithm, the *outer loop* selects a node to compute new distance labels. Inside this loop, the algorithm traverses with the help of an *inner loop* its outgoing edges in order to relax the old distance labels. After these relaxing operations, the algorithm calculates the minimum tentative distance from the unsettled nodes set in order to extract the next frontier node. Table 2.2 summarizes a list of parallel solutions, classified according to the parallelization method implemented.

Inner Loop Parallelism: Single-vertex Parallel Edge-traversing

Parallelizing the *inner loop* implies to simultaneously traverse the outgoing edges of the frontier node f. Each of these outgoing edges (f, v) is assigned to a different processing unit that checks, first, whether v belongs to the unsettled set U, and, if so, evaluates the relaxing condition, $\delta(f) + w(f, v) < \delta(v)$, with the aim of keeping the lowest value of them.

One of the algorithms presented in [59], the *Fine-Grain Parallel SSSP*, is an example of this kind of parallelization. Their authors have implemented a parallel Bellman-Ford

algorithm mixed with Dijkstra's ideas. The set of edges is divided into different disjoint subsets in such a way that the number of outgoing edges of the same vertex is balanced through these subsets. That is, if node u is connected to nodes x, y, and z, and node v to a, b, and c, three edge subsets will be created: The first one will contain the edges (u, x) and (v, a); the second one will contain (u, y) and (v, b); and, finally, the third one will contain (u, z) and (v, c). Each processing unit handles one of these subsets, together with a local heap that stores the tentative distances of the reached nodes. In every iteration, each processing unit communicates to the others its local minimum δ_{PUi}, and, through a reduce operation, all of them keep the global minimum. This minimum value corresponds to the node that will be the following *frontier node* f, and it is broadcast together with its $\delta(f)$. Then, each processing unit traverses its corresponding subset of outgoing edges associated to this current frontier node, relaxing whenever possible the distances of its successor vertices, and storing them in its local heap. A processing unit sends to the rest a fake node with infinite value when its local heap is empty. The algorithm repeats this process until there are not more nodes to compute in any of the local heaps.

Outer Loop Parallelism: Multiple-vertex Sequential Edge-traversing

Parallelizing the *outer loop* from a label-setting approach implies to compute, at each iteration i, a frontier set F_i of nodes that can be settled in parallel without affecting the algorithm's correctness. The main problem here is to identify the set of nodes v whose tentative distances from the source vertex s, $\delta(v)$, are actually the minimum shortest distance, $d(v)$. As the algorithm advances in the search, the number of reached nodes that can be computed in parallel increases considerably.

Some label-setting algorithms that are based on this idea are the ones proposed in [24, 23]. Both algorithms define a limit value, that we name as the Δ threshold, that determines at each iteration which unsettled nodes have at that moment the minimum possible shortest path distance, and thus, that can be safely settled. Note that the unsettled nodes with a tentative distance lower or equal than this threshold can be settled and selected as a frontier node without the need of following a particular order between them, and therefore, it is possible to process all of them at the same time. The work described in [24] tries to compute a bigger threshold, with the help of precomputed values, compared with the work described in [23]. However, this latter work is designed to be executed in GPUs instead of using classic CPU cores. As we will see, one of the main contributions of this Ph.D. Thesis is based in the use of both algorithms, taking advantage of the aggressive enhancement that represents to compute a bigger threshold in the development of a GPU implementation of the SSSP algorithm. A full detailed description can be found in the following chapter.

The parallelization of the *outer loop* for label-correcting approaches follows a similar idea of augmenting, this threshold beyond the safety limit. By taking this risk, these algorithms favor the early, tentative settlement of some unsettled nodes whose current tentative distances are beyond this safe limit, making available the parallel computation of more nodes, but at the cost of recomputing later some results if their speculation proves to be wrong. Note that no settlement is real till the end of the algorithm, because the last computation could invalidate all the previous results. Analogously to the *frontier set,*

F_i, used in the label-setting algorithms, we denominate as the *gambling set,* G_i, the set of nodes that the label-correcting algorithms consider "good enough" to be tentatively settled.

Δ-Stepping is an example of a label-correcting algorithm [60, 42] that also parallelizes the *outer loop* of the original Dijkstra's algorithm, by exploring in parallel several nodes grouped together in a bucket. Several buckets are used to group nodes with different tentative distance ranges, $(0, \Delta]$, $(\Delta, c_1\Delta]$, $(c_1\Delta, c_2\Delta]$, and so on. At each iteration, the algorithm relaxes in parallel all the outgoing edges from all the nodes in the lowest range bucket. It first selects the edges that end on nodes inside the same bucket (light edges), and later the rest (heavy edges). Note that this algorithm can find that a node that is currently being processed has to be recomputed if another node reduces the tentative distance of the first one at the same time, implying a possible bucket change of its reached nodes. The implementations for GPUs presented in [61, 22] go even further, by considering part of the *gambling set* G_{i+1}, all the unsettled nodes whose tentative distances have been updated in the immediately previous iteration i.

Combining Inner and Outer Parallelism: Multiple-vertex Parallel Edge-traversing

The combination of both parallelization strategies would result in an algorithm where each processing unit traverses a single outgoing edge, or a subset of outgoing edges, from a node: (a) That belongs to the frontier set, for a label-setting approach, or (b) that belongs to the gambling set, for a label-correcting philosophy. To the best of our knowledge, it does not exists any implementation or study considering the former approach, that is, a combination of both parallelizations following the restrictions of the label-setting criteria. On the other hand, the work described in [62] presents a modified label-correcting implementation of the Bellman-Ford algorithm ported to GPUs, that suits well with this double parallelization behavior. This implementation associates to each GPU thread one edge (u, v) of the graph, checking in each iteration, if it is possible, the relaxation of the tentative distance of v by $\delta(u) + w(u, v)$, similar to Bellman-Ford's algorithm.

2.3.2 Deploying Sequential SSSPs in Disjoint Subgraphs Concurrently

There are approaches that exploit parallelism by executing several sequential SSSP algorithms in parallel in different parts of the graph. Implementations of these parallel algorithms usually need some simple kind of graph partitioning, from which they obtain a set of disjoint subgraphs of the input graph. Each processing unit is responsible of handling a disjoint subgraph, executing a sequential SSSP algorithm in there. Every k iterations, the computation is stopped to perform a communication phase. In this phase, each process exchanges the distance information regarding the boundary vertices of its subgraph with the process that handle the corresponding adjacent subgraphs, in order to update, if necessary, their boundary tentative distances. If the tentative distance of a boundary vertex is relaxed during this information exchange, the algorithm has to recompute the boundary vertex in the corresponding subgraph, repeating the search with the new value in order to ensure correctness. Although the algorithms that lie under this category follow the label-correcting idea of recomputing the results due to this communication phase, the

SSSP algorithms deployed inside each subgraph can use any of the previously described approaches, label-setting or label-correcting algorithms.

The work presented in [63] is an example of using a label-setting algorithm, based on Dijkstra's algorithms with priority queues, with the described computation and communication phases using disjoint subgraphs. In the exchanging information step, if the tentative distance of a boundary node is relaxed, the algorithm simply reinserts this node with the new value into the priority queue of the corresponding subgraph. Once this value represents the minimum distance of the priority queue, the boundary node will become the corresponding frontier node. After that, the algorithm will relax, if needed, its corresponding successor distances, reinserting the nodes also in the queue, and repeating the spreading of the exchanged value. The algorithm continues by iterating and updating the boundary distances, until there are not any remaining nodes in the priority queues, and there is not more boundary updates.

Finally, another algorithm presented in [59] follows the same ideas, but applying an optimized Bellman-Ford algorithm on each subgraph. This modified version stores whether the tentative distance of a vertex has been relaxed. In each iteration, the algorithm first checks these conditions, and if the distance was updated, it tries to relax the corresponding successors. Every time a distance is relaxed, a flag is set indicating that there are more possible relaxations. In the communication step, the algorithm proceeds with the updates if necessary, it checks the flag, and it resets it again for the following iteration. These steps are repeated until, at the end of the communication step of an iteration, the algorithm finds that there are no flags set, meaning that it is not possible to relax distances anymore.

2.3.3 Deploying Parallel SSSPs in Disjoint Subgraphs Concurrently

To best of our knowledge, there are not works that try to combine the use of a parallel SSSP algorithm combined with the approach, described above, of deploying several of them in disjoint subgrahps. It remains as an open question whether this kind of combination is really efficient against the other known approaches described above.

2.4 The All-Pair Shortest-Path (APSP) Problem

The All-Pair Shortest-Path (APSP) problem consists on computing the shortest paths between all pair of vertices v_i, $v_j \in V$ such that $i \neq j$. For a graph with $n = |V|$, the output of the algorithm is a $n \times n$ matrix $D_{i,j} = (d_{i,j})$ such that $d_{i,j}$ is the distance of the shortest path from vertex v_i to vertex v_j. In this section, a review of the sequential solutions to the APSP problem is presented,

2.4.1 Taxonomy of APSP Algorithms

Depending on the behavior of each APSP algorithm, they can be classified as dynamic programming or productivity-based approaches.

Dynamic-programming These approaches give solutions to different subproblems and combine them to create the solution of the main problem. A dynamic-programming algorithm saves subproblem answers, avoiding the work of recomputing them when they are required to solve bigger subproblems.

Productivity-based These approaches computes independent subproblems using algorithm particularly designed for them instead for the complete problem. Usually, these approaches do not share the information results obtained from a subproblem in order to fast the solution of the following one, or if they do, this sharing is reduced for a particular subset of solutions.

According to the size of their spatial complexity, the APSP algorithms can also be divided into linear spatial complexity and quadratic spatial complexity algorithms.

Linear spatial-complexity algorithms The first group stores the edges and its weights in adjacency lists, having a spatial complexity in $\Theta(n+m)$. Usually, these algorithms solve the APSP problem by executing a SSSP algorithm n times, once for each vertex as the source.

Quadratic spatial-complexity algorithms The second group uses a matrix W, called adjacency matrix, to represent the connections between vertices, that are, the edges and their weights. For an edge $e = (u, v) \in E$, the matrix W has the corresponding entry $W_{u,v} = w(e)$. If $e = (u, v) \notin E$ then $W_{u,v} = \infty$. Therefore, the spatial complexity of these algorithms is in $\Theta(n^2)$. Usually, this kind of algorithms takes advantage of the partial results of other subproblems, that represent tentative shortest path distances, stored as elements of the matrix W. The main drawback of all these algorithms is the big space consumption, that makes them impractical for large graphs.

Combining both criteria proposed we obtain a four-class classification. The following paragraphs describe the most representative state-of-the-art algorithms that match in each category of our classification, describing their properties. Table 2.3 shows the best-time bounds algorithms for each type of graphs or edge-weights restrictions. See [41] for an overview.

Productivity-based algorithms with Linear spatial complexity

Due to its small space consumption (in $O(m + n)$), these algorithms are a better choice for large sparse graphs than the quadratic spatial-complexity ones.

The first naïve approach to solve the APSP problem is to execute a SSSP algorithm for each vertex $v \in V$. This strategy is known as the n-SSSP algorithm, or also the $n \times SSSP$ approach. If the SSSP algorithm used is Dijkstra's (n-Dijkstra) combined with Fibonacci heaps, it will have a time-cost algorithm in $O(n^2 \log n + nm)$.

For graphs with negative-weight edges but no negative-weight cycles, Johnson's algorithm [74] uses the Bellman-Ford algorithm for reweighting the edge weights to positive ones. Then, it executes n times the Dijkstra's algorithm, leading to the same time-cost

Weight / type	Algorithm	Time complexity
Unw / Dir	Zwick [64, 65]	$O(n^{2.575})$
Unw / Und	Seidel [66], Galil *et al.* [67, 68]	$\tilde{O}(n^{\omega})$
Unw / Und $^{m \ll n^{\omega-1}}$	Chan [69]	$O(mn/\log n)$
$\mathbb{R}_{\geq 0}$	n-Dijkstra [44]	$O(mn + n^2 \log n)$
	Karger *et al.* [70], McGeoch [71]	$O(m'n + n^2 \log n)$
$\mathbb{R}$ / Dir	Pettie [72, 73]	$O(mn + n^2 \log\log n)$
$\mathbb{R}$	Johnson [74]	$O(mn + n^2 \log n)$
$\mathbb{R}$ / Dense	Floyd-Warshall [75, 76]	$O(n^3)$
	Han [77]	$O(n^3 \log\log n/(\log n)^2)$
$\mathbb{Z}$	Hagerup [78]	$O(mn + n^2 \log\log n)$
$\mathbb{Z}_{\{-M..M\}}$ / Dir	Zwick [64, 65]	$\tilde{O}(M^{0.68} n^{2.575})$
$\mathbb{Z}_{\{1..M\}}$ / Und	Shoshan and Zwick [79]	$\tilde{O}(Mn^{\omega})$

Table 2.3: Best time-complexity bounds of APSP algorithms for different graphs. The abbreviations *Unw*, *Und* and *Dir* refer to unweighted, undirected and directed respectively. The ω exponent is the smallest constant for which matrix multiplication can be performed using algebraic operations in $O(n^{\omega+O(1)})$. The $\tilde{O}(f)$ notation is a shorthand for $O(f \cdot (\log n)^{O(1)})$ to hide not-so-interesting, polylogarithmic factors.

than Dijkstra's algorithm with Fibonacci Heaps. Later, Pettie [72, 73] reduced this bound to $O(n^2 \log\log n + nm)$ for directed graphs.

Yanagisawa have proposed in [80] the use of a Multi-Source Shortest-Path (MSSP) algorithm, specifically the *centralized shortest-path search* [81], instead a of SSSP algorithm to solve the APSP problem faster. To summarize, this algorithm partitions the graph into regions composed by B vertices and later executes the MSSP search n/B times, once for each region, leading to a productivity-based algorithm with increased linear spatial complexity in $O(m + Bn)$.

Productivity-based algorithms with Quadratic spatial complexity

The quadratic spatial complexity algorithms take advantage of the computations previously done for other vertices to speedup the following ones. The Dijkstra-based algorithms with quadratic spatial complexity are faster in sparse graphs than other quadratic spatial-complexity algorithms that are designed to be more efficient for dense graphs.

Karger [70] and independently McGeoch [71] introduced the term *essential edges*, referring to those edges of the graph that actually participate in shortest paths. These algorithms save unnecessary operations leading to a running time complexity in $O(nm' + n^2 \log n)$, where m' is the number of essential edges. Demetrescu and Italiano [82] used this idea to propose a similar algorithm, using a Dijkstra-algorithm variant with $O(|LSP| + n^2 \log n)$ running time complexity, where LSP is the set of local shortest paths in the graph.

Peng [83] optimized the Dijkstra algorithm for two well-known complex networks: Scale-free networks and Small-world networks.

Algorithm 2 The dynamic programming Floyd-Warshall algorithm.

```
1: D ← W
2: for k ← 1...n do
3:    for i ← 1...n do
4:       for j ← 1...n do
5:          d_{i,j} ← min(d_{i,j}, d_{i,k} + d_{k,j})
6:       end for
7:    end for
8: end for
9: return D
```

Author	Year	Time cost bound
Floyd-Warshall [75, 76]	1962	$\Theta(n^3)$
Fredman [84]	1976	$O(n^3(\log\log n/\log n)^{1/3})$,
Takaoka [85]	1992	$O(n^3(\log\log n/\log n)^{1/2})$
Dobosiewicz [86]	1990	$O(n^3/(\log n)^{1/2})$
Han [87]	2004	$O(n^3(\log\log n/\log n)^{5/7})$
Takaoka [88]	2004	$O(n^3(\log\log n)^2/\log n)$
Takaoka [89]	2004	$O(n^3\log\log n/\log n)$
Zwick [90, 91]	2004	$O(n^3(\log\log n)^{1/2}/\log n)$
Chan [92, 93]	2005	$O(n^3/\log n)$
Han [94, 95]	2006	$O(n^3(\log\log n/\log n)^{5/4})$
Chan [96]	2007	$O(n^3(\log\log n)^3/(\log n)^2)$
Han [77]	2012	$O(n^3\log\log n/(\log n)^2)$

Table 2.4: Time-bound evolution for FW-based algorithms in directed graphs with real edge weights.

Dynamic-programming algorithms with Quadratic spatial complexity

The classic solution of the APSP problem is a dynamic programming approach created by Floyd [75], inspired in the work of Warshall [76]. The Floyd-Warshall (FW) algorithm (see Algorithm 2) iterates n times through all the elements of the matrix $W_{n\times n}$. It leads to a $\Theta(n^3)$ time cost algorithm. Having $D^0 = W$, it computes new matrices D^k in each iteration k, trying to relax the old values of $D^{(k-1)}$ by $d^k \leftarrow min(d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)})$. Therefore, the spatial complexity of the algorithm is in $\Theta(n^2)$. Note that only one matrix $D_{n\times n}$ is needed if the relaxations are always made on it.

The complexity of the FW algorithm has been reduced since then. Starting with Fredman's algorithm [84] in 1976 that was the first in breaking the cubic complexity, until now with the last approach of Han [77] (see Table 2.4). In fact, computing the APSP problem is computationally equivalent to computing the product of two matrices, where the multiplication and addition operations are replaced by addition and minimum operations respectively. However, the order of the loops cannot be changed arbitrarily as in the case of matrix multiplication.

There are some implementations of the FW algorithm using different ways to improve

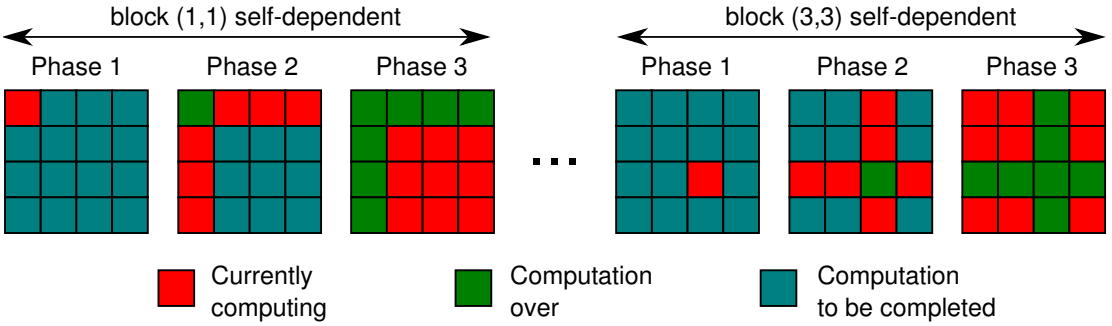


Figure 2.4: Phases of the FW algorithm proposed by Venkataraman [97], that later was used by Katz and Kider [98] for GPUs.

its performance. Venkataraman [97] optimizes the cache performance introducing a tiled version (see Fig. 2.4), where the values of the adjacency matrix are computed in a particular order through three phases. The first phase computes the distances of the current diagonal block. These diagonal bocks are known as self-dependent blocks because the values needed to compute the new distances are all contained into it. Afterwards, these new calculated values are used in the second phase, to compute the blocks of the same row and column, the half-dependent blocks. Finally, in the third phase, all remaining tiles are computing using the values obtained in the previous phase. The algorithm finishes by repeating this step (with its three phases) n times, updating in each iteration the adjacency matrix W, that becomes the final result at the end. More improvements were made in [99, 100] by applying standard cache-friendly optimizations.

Duin [101] avoids redundant updates by computing the shortest paths in order of length. It takes advantage of the dynamic programming principle of optimality of subpaths [43]. To obtain this, it builds two shortest path trees per node, giving a computational overhead effort that is not compensated for sparse graphs. Han [102] proposed an automatic tuning framework that produces very fast implementations of the FW algorithm.

Better performance algorithms have been designed for particular input graphs, such as graphs without edge weights, *unweighted graphs*, or only integer weights, in their directed or undirected graph variants (see Table 2.3). Some of them apply fast matrix multiplication algorithms[1] to the all-pairs shortest paths problem.

2.5 Parallel Solutions for the APSP (Π-APSP)

There are two different trends to apply parallelization to the APSP problem. The first one is to modify the purely sequential algorithmic solutions to support parallel computation, whereas the second is related with the productivity of executing independent SSSP in different computational resources. In this section we present an overview of the state of the art for both approaches (see Table 2.5).

[1] See [103] for an overview, and for details on the last achievement, $O(n^{\omega})$ with $\omega < 2.3727$.

Parallel Dynamic-programming	Parallel Productivity-based
1999 Diament *et al.* [104]	1999 Diament *et al.* [104]
2004 Micikevicius [105]	2006 Sen [109]
2007 Harish *et al.* [61]	2007 Harish *et al.* [61]
2008 Katz and Kider [98]	2008 Okuyama *et al.* [110, 111]
2009 Buluç *et al.* [106]	2009 Harish *et al.* [22]
2009 Harish *et al.* [22]	2010 Yanagisawa [80]
2013 Wu [107]	2014 Hajela and Pandey [112]
2014 Djidjev *et al.* [108]	2014 Hajela and Pandey [113]

Table 2.5: Different existent implementations for both parallel APSP strategies, parallel dynamic-programming and parallel productivity-based approaches.

Algorithm 3 Pseudo-code for the naïve parallel Floyd-Warshall algorithm

1: $D \leftarrow W$
2: **for** $k \leftarrow 1 \ldots n$ **do**
3: **for all** elements of the Adjacency Matrix D where $1 \leq i, j \leq n$ in parallel **do**
4: $d_{i,j} \leftarrow min(d_{i,j}, d_{i,k} + d_{k,j})$
5: **end for**
6: **end for**
7: **return** D

2.5.1 Strategy A: Parallel Dynamic-programming Solutions

There are different methods to parallelize a sequential APSP algorithm. This section gathers and describes these techniques briefly, and their corresponding results.

Naïve Parallelization of FW

The naïve parallel version assigns the computation of the element of the adjacency matrix to different computational units (see Alg. 3). Diament *et al.* made in [104] an analysis of costs of this approach, and developed an implementation for a CPU cluster.

Micikevicius presented in [105] a GPU implementation for this approach, but the experimentation was only applied to small graphs due to memory restrictions. Lately, Harish *et al.* in [61] presented more results with bigger graphs. In both GPU implementations, each element of the adjacency matrix is computed by a different GPU thread.

Blocking Approach for FW

The divide-and-conquer version of the Floyd-Warshall algorithm is known as the the FW blocking approach. This approach divides in blocks the adjacency matrix and computes them in a particular order with the aim to take advantage of the cache memories while maintaining the correctness of the result. Diament *et al.* also presented in [104] a cost analysis together with a FW blocking implementation, and a comparison to their naïve approach. The use of the caches significantly outperformed the non-blocking solution.

Katz and Kider [98] presents a CUDA implementation for APSP of this approach,

efficiently using the GPU cache/shared memory for this blocking technique (based on the work proposed by Venkataraman [97]). The algorithm first partitions the adjacency matrix W into tiles, or blocks, of equal size. In the implementation of Katz and Kider, this partitioning is strongly related to the threadblock size used to compute the algorithm in the GPU. The next step of the algorithm is composed by three sequential phases (see Fig. 2.4), that are carried out using a different GPU kernel for each phase. The first phase computes the distances of the self-dependent block. Note that this self-dependent block is processed using only a GPU threadblock in one multiprocessor of the hardware accelerator. On the other hand, on phases two and three, where the distances of the half-dependent blocks and the remaining ones are respectively computed, one GPU threadblock is responsible of computing its assigned tile. This implementation achieved a speedup up to 5× over the parallel FW implementation presented in 2007 by Harish *et al.* [61].

Wu [107] extended the work of Katz and Kider by creating a version that creates subblocks of the tiles to efficiently compute bigger graphs. This author also presented an implementation that supports the execution of the algorithm across systems with multiple GPUs, or in a GPU cluster.

Based on Matrix Multiplication

Harish *et al.* [22] have implemented a solution for the APSP using a parallel matrix multiplication with blocking approach. These authors modify the parallel version of matrix multiplication proposed by Volkov and Demmel [114], replacing the multiplication and addition operations used in their kernel with addition and min operations, respectively. Additionally, they skip the addition and minimum computation of all entries that involve an infinite value, a decision that do not affect the correctness of the algorithm. For fully connected small graphs, the times of their approach are slightly larger compared to the Katz and Kider variant, whereas its implementation showed to be up to 4× faster for general large graphs.

Based on using Gaussian Elimination

The work of Buluç *et al.* [106] presents a recursive APSP algorithm based on Gaussian elimination (see Alg. 4). This algorithm divides the computation of each APSP step recursively into two sub-APSPs computations, six matrix multiplications, and two matrix additions. These sub-APSPs work with graph subsets involving graphs of half the size. The recursion ends when the base case is reached, that is, when the graph subset has 16 or fewer vertices. Then, the FW algorithm is applied using the matrix multiplication GPU kernel of Volkov and Demmel [114]. Their implementation achieved a speedup up to 10× over the parallel FW implementation presented by Harish *et al.* [61].

Afterwards, Harish *et al.* [22] improved this last version by incorporating their proposed modifications, described in the previous paragraphs, to the matrix multiplication kernel of Volkov and Demmel. These modifications led to a speedup of more than 2× over the original code.

The experimental results of both papers showed that this parallel approach has the best results compared with the previously presented parallel FW and the parallel $n \times SSSP$ algorithms, that will be described in the following section, in dense and sparse graphs,

Algorithm 4 Pseudo-code for recursive parallel APSP algorithm of Buluç *et al.* [106]

func_APSP(A)

1: **if** $N < \beta$ **then**
2: $A \leftarrow FW(A)$;
3: **else**
4: $A = \begin{bmatrix} A_{11}A_{12} \\ A_{21}A_{22} \end{bmatrix}$;
5: $A_{11} \leftarrow func\_APSP(A_{11})$;
6: $A_{12} \leftarrow A_{11}A_{12}$;
7: $A_{21} \leftarrow A_{21}A_{11}$;
8: $A_{22} \leftarrow A_{22} \oplus A_{21}A_{12}$;
9: $A_{22} \leftarrow func\_APSP(A_{22})$;
10: $A_{21} \leftarrow A_{22}A_{21}$;
11: $A_{12} \leftarrow A_{12}A_{22}$;
12: $A_{11} \leftarrow A_{11} \oplus A_{12}A_{21}$;
13: **end if**

respectively. However, their authors have only tested this method with very small graphs ($n < 10.000$). Additionally, if the running times for larger sparse graphs of this approach are estimated following its tendency, it seems to have worse scalability against the parallel $n \times SSSP$ strategy (see the experimental results of [22, 106] for more details).

2.5.2 Strategy B: Parallel Productivity-based Solutions

There is a second kind of parallel approaches that computes the APSP problem using the sequential solution of performing a SSSP algorithm for each vertex of the graph. They usually have better performance than the parallelization of dynamic-programming APSP algorithms for non-dense graphs [43, 22, 104, 106]. These approaches can be classified regarding the aspect they try to parallelize according to [115]:

- **Source-partitioned Solutions,** that parallelize the serial n executions of a sequential SSSP algorithm, (Π - ($n\times$SSSP)).
- **Source-parallel Solutions,** that involve the execution of a parallel SSSP algorithm instead of a sequential one. We want to distinguish two particular configurations inside this approach:
  - **Sequential Source-parallel Solutions,** the n executions of the used parallel SSSP algorithm are executed sequentially, ($n\times$(Π - SSSP)).
  - **Partitioned Source-parallel Solutions,** the n executions of the used parallel SSSP algorithm are executed in parallel, (Π - ($n\times$(Π - SSSP))).

Figure 2.5 shows the taxonomy of parallel solutions for the APSP problem focused on the productivity-based approaches, where each green bar represents the execution of a SSSP problem. Close to each productivity-based approach it is depicted the distribution of the SSSP problems in the different computational units (CU). We have introduced the

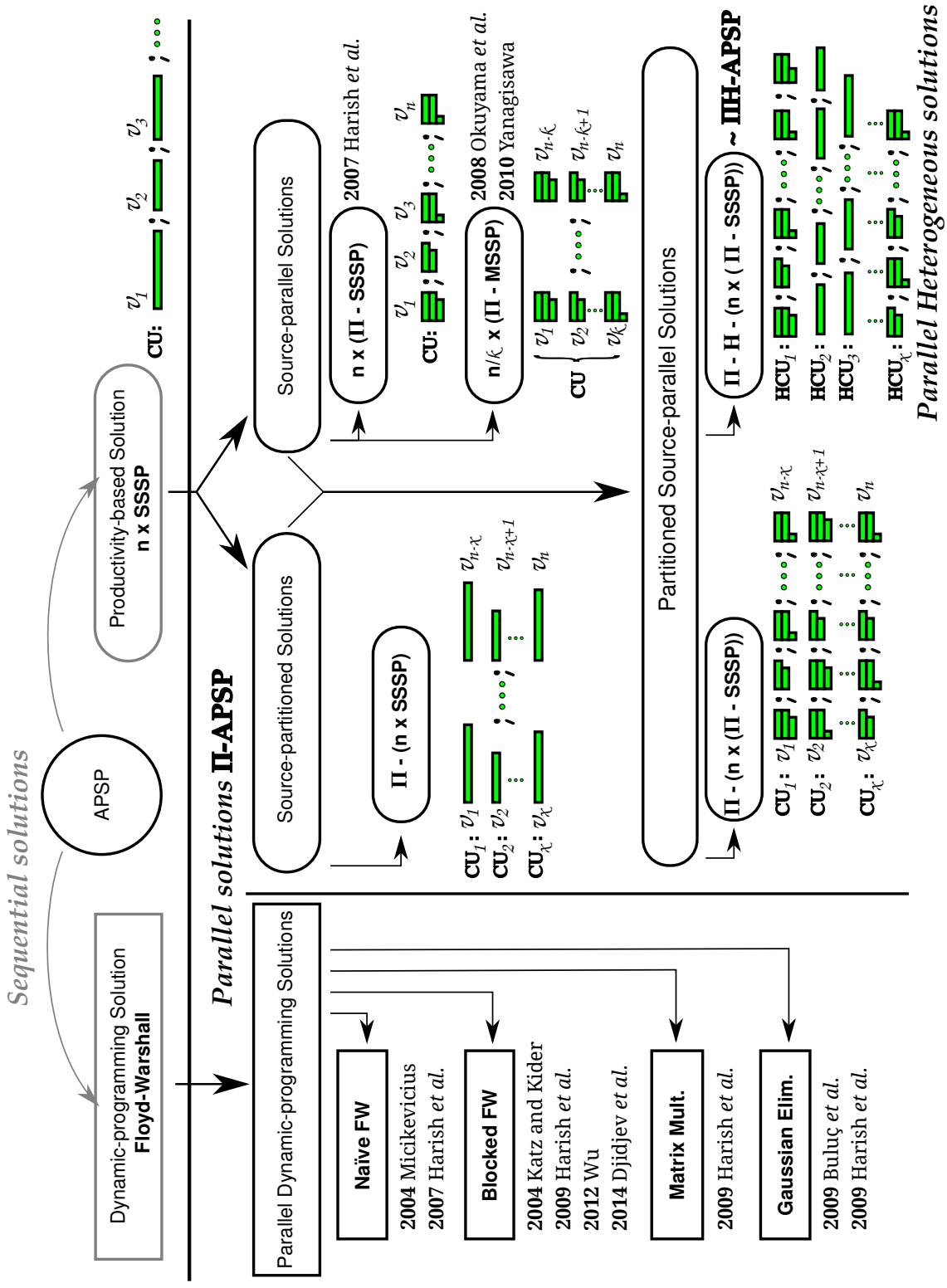


Figure 2.5: Taxonomy of parallel solutions for the APSP.

notation ΠH-APSP, for the Partitioned Source-parallel Solutions deployed in systems that contains heterogeneous computational units (HCU). These ΠH-APSP approaches are one of the main points of study of this Ph.D. thesis.

The following paragraphs describe the implementations found in the literature according to the classification previously described.

Source-partitioned Solutions

This approach parallelizes the execution of sequential SSSP algorithms. Although the implementation of this approach is very simple, it does not attract the interest of the current scientific community since more efficiency can be obtained parallelizing also the SSSP algorithm involved in the execution, and even less since the arrival of parallel algorithms for hardware accelerators. The last studies regarding this approach are [104, 109].

Sequential Source-parallel Solutions

These algorithms use parallel SSSP implementations that are executed in sequential order. We can find two slightly different implementations from the literature that differ in the algorithm used. The first uses the mentioned parallel SSSP solution, whereas the second involves a more complex MSSP algorithm.

$n\times$(Π - SSSP) Harish *et al.* in their works [61, 22] have presented implementations following this strategy. They have used as the SSSP algorithm their own version implemented for GPUs, that is executed n times, taking a different node as source each time. Briefly resumed, their SSSP algorithm iteratively executes a sequence of kernels with as many GPU threads as vertices of the graph, that is n, in order to parallelize the internal operation of the outer loop, solving one SSSP problem in parallel at once. When this algorithm has finished, another one is launched but changing the source node, and so on until n executions.

$n/k\times$(Π - MSSP) Okuyama *et al.* in [110, 111] have extended this previous work by using more complex GPU kernels. They first group the vertices of the graphs into subsets of k nodes. The kernels they launch have $k \times n$ GPU threads, solving in one kernel execution k SSSPs instead only one. Then, their approach needs just n/k executions instead n. However, the key of this modification is not only the reduction of the number of executions, but also the sharing of information obtained during the computation between the nodes of the same subset through the GPU shared memory. They have obtained speedups up to 1.9× compared with the original approach, but only for small graphs ($n < 1000$). As the number of vertices of the graph increases, the running times of both approaches tend to be the same [111]. Yanagisawa *et al.* in [80] have implemented a similar solution for CPU cores using SIMD instructions.

Partitioned Source-parallel Solutions

This strategy parallelizes both dimensions at the same time, using a parallel SSSP algorithm that are executed concurrently. To best of our knowledge there are no studies re-

garding this kind of approach. This is due the joint use of different parallel programming models is becoming stable nowadays.

The last trends of High Performance Computing (HPC) are focused on using all computational resources available on a processing platform. One of the efforts of this Ph.D. thesis is to study and develop both homogeneous and heterogeneous implementations following this yet unexplored Partitioned Source-parallel approach.

After the publication of our proposals, described in the following chapters, another works have been developed in the scientific community. This fact proves the importance of these approaches in the new HPC era. These works are due to Hajela and Pandey [112, 113]. The first one implements a solution involving several instances of the GPU parallel implementation of the Bellman-Ford algorithm, in a multi-GPU environment (homogeneous approach, Π - ($n\times$(Π - SSSP))). The last work adapts the previous solution in order to take advantages of the CPU cores present in the computational environment (heterogeneous approach, ΠH - ($n\times$(Π - SSSP)), or simply ΠH - APSP).

2.6 Application Example: Shortest-Path Algorithms applied to roadmaps

The aim of parallelizing these algorithms is, not only the immediate reduction of the execution time of themselves, but also their application in complex approaches, that use them as a step in their algorithms. Usually, the parallelization of this step is the only way to make feasible the complete approach due to the high temporal costs. One straight example related with the shortest-path context are the costly preprocessing phases of the modern methods and techniques used for routing in transportation networks. These approaches compute shortest paths, and their distances, between two vertices of the graphs in the order of nanoseconds [10, 11], thanks to complex previously precomputed values, that in some cases, represent the solution of the whole APSP problem.

The topology of a road network hardly changes, so it is feasible to pay the high temporal and spatial costs of computing these values once and store them. In other contexts, where the static nature of the network is not present, such as routing in the Internet [7], web searching [8, 9], real-time logistic control [4], scheduled means of transport [3, 116], or traffic simulations [2], among others, the reduction of these costs is highly significant. Additionally, in the previously mentioned context of routing, if it is desired to take into account the current state of the roads in real-time, such as the information of the traffic, the road cuts due to maintenance or natural events, or even the avoidance of low-speed/bad-quality roads for a particular driver, the application of efficient parallel methods is compulsory. The advent of mobile devices present a third challenge because of their small memory size, that limits the amount of precomputed data that can be stored. Most of the current approaches present a trade-off between the amount of memory used and the query time needed: The more memory used, the better the query time. However, the new devices nowadays have bigger computational capabilities, incorporating several processors that could be exploited to alleviate this mentioned lack of memory, by recomputing values instead of storing them.

Many algorithms have been developed so far to solve the shortest path problem from

a source vertex to a target vertex. A thoroughly revision of the state of the art, and a comparison of their relative strengths/weakness through an case-study application is shown in [20]. The following sections describe the problem of routing between two points, the classic algorithms that solve it, and also the modern techniques in roadmaps together with their corresponding preprocessing phases.

2.6.1 One-Pair Shortest-Path Problem

The One-Pair Shortest-Path problem (OPSP), also known as the Point-to-Point Shortest-Path problem, is a variation of the classical SSSP problem (see Sect. 2.2), where it is only computed the shortest path from a source node to a specific target node, and its corresponding shortest path distance. The classic algorithm that solve this problem is a slight variation of the naïve Dijkstra algorithm. It differs from the original solution in the termination criteria. As the only path needed to compute is the one from source node s to target node t, the Dijkstra algorithm works as usual until it settles the destination vertex.

As from the beginning of the computation it is known which is the destination node, it is possible to create new improved algorithms that use this information in order to reduce the running times. Deploying two Dijkstra searches, one from the source node, and another from the target node, until they meet in the middle is one example of the usage of this information. Applying heuristics, or pruning criteria, also results in new more efficient algorithms.

Bidirectional Search

The Bidirectional Search algorithm [117, 118, 119] alternates between two Dijkstra's searches, one from s to t, and a second one, called backward search, from t to s. In the backward search, the key of a node v is the distance from v to the target node t. Note that, in a directed graph, the reversed edges should be considered for the backward search.

Having two searches to perform, it is possible to maintain two separate queues and use a turn policy to take the next node, or to put all the reached nodes in the same queue, taking always the one with the lowest key, independently of which search was being considered. Both methods work correctly [120].

The termination condition for bidirectional search is slightly different than Dijkstra's. Let μ be the minimum distance between s and t found so far. At the beginning, $\mu = \infty$. When an edge (u, v) is relaxed in the forward search, with node v previously settled in the backward search, then a path from s to t has been found, with a known distance. If this distance is less than μ, then μ is updated and v belongs to the shortest path found so far. The same occurs when a node u already settled by the forward search is reached by the backward search. Algorithm finishes when both searches settle the same node. At that point, the shortest path can be retrieved by traversing the predecessors (in both directions) of the node that triggered the last update of μ.

Comparing with Dijkstra's algorithm, this strategy, in terms of efficiency, is almost twice as fast as the original algorithm (see [120] for details).

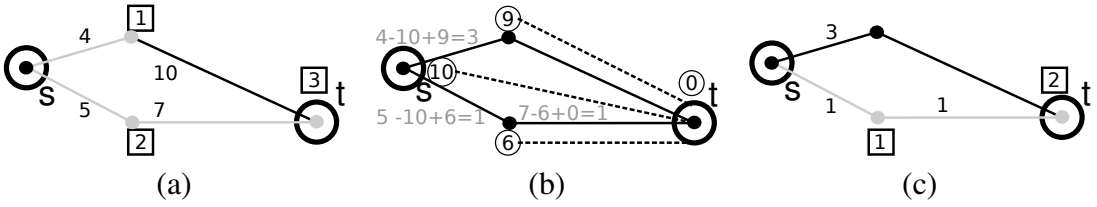


Figure 2.6: Settled nodes reordering from Dijkstra's to A*.

Goal-Directed A* Search

This approach [121] uses heuristics that try to avoid nodes not involved in the shortest path. Heuristics use domain information in order to look-ahead in the decision process, trying to improve query times. However, if they are applied alone, their use does not always lead to shortest paths. As we will see, an appropriate heuristic combined with Dijkstra's algorithm ensures correct results.

In a goal-directed A* search, a heuristic (or potential) function $h'(v)$ returns a value that represents a lower-bound estimation of the remaining distance from that node v to the target one t. The query phase applies as a normal Dijkstra's algorithm, where edge weights $w(u, v)$ are substituted with $w'(u, v) \leftarrow w(u, v) - h'(u) + h'(v)$. This new value represents the additional cost with respect to the heuristic estimation from the source, $h'(s)$, in case edge (u, v) is taken. The closer the lower bound value to the real one, the better the obtained results.

Dijkstra's algorithm uses distances from the source to reached nodes as the only criteria to decide which node will be selected next. Starting with the situation depicted in Fig. 2.6 (a), Dijkstra's algorithm will choose the upper branch, while the lower branch belongs to the shortest path. On the other hand, A* estimates a lower bound distance from all reached nodes to the target, for example, the Euclidean distance from the node to the target for spatial graphs. Figure 2.6 (b) shows these the Euclidean distances in circle-bounded numbers. With such an estimation, the transformation of edge weights, $w'(u, v) \leftarrow w(u, v) - h'(u) + h'(v)$, leads Dijkstra's algorithm to take a correct decision faster (see Fig. 2.6 (c)). For the route planning case in road maps, the Euclidean distance is a lower bound and therefore a valid estimator of $h(x)$. However, the search space is not always reduced enough to palliate the heuristic cost insertion (see experimental results in [120, 122]).

There are also approaches that combines these heuristics with the bidirectional strategy, having two potential (heuristic) functions, one for the forward search h'_s, and other for backward search h'_t. To use the termination condition that stops when both searches meet at the same node, proper potential functions should be selected. This issue leads to two different set of approaches: Those that impose new termination conditions, called Symmetric approaches [123], and those that use consistent potential functions, called Consistent approaches (see [124] for details).

2.6.2 The Importance of Preprocessing: Routing Algorithms as Example

During the last decade, preprocessing strategies have led to huge query speedups [125, 126] compared to the classic algorithms. Their speedups rely on the costly preprocessing phase, where they compute and store some strategic values that later will be used in the query phase. Depending on the nature of the resulting information from the preprocessing phase, and thus, its use in the query phase, the algorithms can be grouped in two big families: hierarchical and non-hierarchical.

Hierarchical Algorithms

The hierarchical methods aim to discover a hierarchy in the graph that, for road networks, usually corresponds with their hierarchical nature. A graph hierarchy is a division of the graph vertices into levels. To define it, some precomputation is needed. There are different precomputations that can be done in the graph depending on the hierarchical algorithm chosen. The precomputation generates additional information that is attached to the graph, and it will be used in order to speed up the query time.

The most representative approaches that are in this category are *Highway Hierarchies*, *Highway Hierarchies with Distance Table*, *Highway-Node Routing*, *Contraction Hierarchies*, *Transit-Node Routing* and *Hub-based Labeling*. We will briefly describe the preprocessing and query phases of these algorithms.

The first hierarchical approach that led to a considerable speedup over Dijkstra's algorithm is **Highway Hierarchies (HH)** [127, 128]. This algorithm uses a preprocessing phase to identify edges that are involved in shortest paths connecting neighborhoods (groups of vertices). These edges will be considered *highways*. Together with their associated nodes, these highways will constitute the following level graph G_{i+1} in the hierarchy. Before creating the next level, the preprocessing phase contracts the current graph G_{i+1}, removing particular vertices and shortcutting them. The level information of all non-contracted graphs is then used to augment G, creating a graph G^+ that will be used as input in the query phase. During the query phase, two Dijkstra searches are performed, a forward one from the source vertex, and a backward one from the target vertex. Both searches traverse the input graph, climbing to upper levels until they meet.

The **Distance Table (DT)** [129] is an optimization of the HH algorithm. It allows the queries to avoid the searches in the highest level of a hierarchy, by precomputing the all-to-all distances between vertices of this level. The HH approach together with this optimization were the inspiration of several hierarchical approaches.

Highway-Node Routing (HNR) [130] needs a subset of vertices V' that have some kind of importance in the graph before building the graph levels. Usually, this subset is computed through a HH preprocessing phase. Once the multilevel graph is created, the query works similarly to HH.

Transit-Node Routing (TNR) [131] also needs a subset of important vertices V', called transit nodes. Depending on the proposals, this subset can be calculated through a graph partition or using the resulting set of applying a preprocessing HH phase. The preprocessing phase of TNR calculates the distance from every vertex v to its access points. These access points are the transit nodes that cover the shortest-path tree whose

root is v. The preprocessing stores, for each vertex in the graph, the shortest path distances to its access nodes, and the shortest path distances between all pair of transit nodes. Then, the query phase is performed just looking up into this distance table.

Contraction Hierarchies (CH) [132, 133] simplified the precomputation of HH using only a contraction step. The preprocessing phase just contracts the vertices following a priority total order, $rank(v)$, and shortcuts them. Its query is similar to HH, considering the vertex total order as levels in the graph. That is, the forward search only considers those edges $e = (u, v) : rank(u) < rank(v)$ whereas the backward one considers those edges $e = (u, v) : rank(u) > rank(v)$.

The **Hub-based Labeling (HL)** approach [134, 10] consists on defining two labels, forward and backward, for each vertex in the graph. Each label stores all pairs of vertices and their corresponding distances, found during a previous CH search, from the vertex considered. The query just intersects the nodes stored in the forward label of the source vertex with the nodes of the backward label of the target vertex.

Non-hierarchical Algorithms

The non-hierarchical preprocessing methods aim to avoid settling unnecessary vertices using information obtained in a preprocessing phase that does not follow a hierarchical structure. The nature of these approaches is diverse, extracting different sets of data during the preprocessing phase. The following subsections will describe the approaches that are in this category.

The most representative approaches that are in this category are *Landmark-based routing*, *Geometric Containers*, *Edge Flags*, *Reach-based routing*, *Precomputed Cluster Distances*, and combinations of some of them, such as *REAL*, or *ReachFlags*. The preprocessing phase of these approaches differs between them because each one computes specific values that needs in its query phase.

Landmark-based routing (ALT) [124, 135] calculates the distances from a subset of vertices, called landmarks, to the remaining vertices and vice-versa. Applying the triangle inequality with this distances in the query phase, it computes better lower bounds for the distance from a vertex to the target vertex compared with a A* algorithm [121].

The **Geometric Containers (GC)** [136] and **Edge Flags (EF)** [137, 81] approaches consist on computing a region that a search can reach through shortest paths if it traverses an edge e. GC preprocessing phase calculates for each edge $e = (u, v)$ the subset of vertices y that are in a shortest path starting with e, $P = \langle u, v, \ldots, y \rangle$. In order to reduce the space consumption, the algorithm just stores a geometric shape that contains them. In a GC query phase, the search can prune those edges whose containers (geometric shapes) do not contain the target vertex. EF needs a previous region definition of the graph. Its preprocessing phase computes all the shortest paths that end in every region. The edges $e = (u, v)$ involved in shortest path towards a region r are tagged with a flag. In an EF query phase, the search can prune those edges that do not have the flag to the region where the target vertex belongs.

The **Reach**-based [122] routing consists on using an upper bound, called *reach value*, for each vertex to prune the search. This upper bound is like the maximum cost that a vertex can bear to belong to a shortest path when a search reaches it. The reach value

of a vertex v is computed as follows. A vertex v in a shortest path $Q = \langle s, \ldots, t \rangle$ has a reach value $r(v, Q)$, that is the minimum value between distance from source, $d(s, v)$, and distance to goal, $d(v, t)$. If vertex v belongs to more than one shortest path between any pair of vertices, its reach value $r(v, G)$, $r(v)$, in a graph G will be the maximum reach value from all the shortest paths that it belongs to. Performing a Dijkstra algorithm in the query phase, the insertion of a vertex v into the priority queue is avoided if is reached with $\delta(v) > r(v)$ and with estimated distance to target $d_e(v, t) > r(v)$.

The **Precomputed Cluster Distances (PCD)** [138, 139] approach is based on upper and lower bounds to prune the search in the query phase. It needs a previous clustering process of the graph vertices. The preprocessing phase computes these bounds calculating the shortest-path distances between clusters. In the query phase, the search handles a general upper bound and computes a lower one for every reached vertex. If this lower bound is higher than the current upper one, then the vertex is pruned.

REAL [135] and **ReachFlags** [140] are improved versions of the Reach algorithm. The former is an improved bidirectional variant of Reach combined with ALT, and the latter is a combination of an improved variant of Reach and EF.

Hierarchical and Non-hierarchical Combinations

Other proposals combine hierarchical with non-hierarchical techniques. The use of separator nodes with multilevel techniques allow the creation of a series of interconnected overlay graphs in a hierarchical fashion. HPMLG [141] and multilevel CRP [142] are some examples. Highway Hierarchies Star (HH*) [143] combines the goal-directed technique of ALT with Highway Hierarchies. SHARC [144, 145] combines a hierarchical multilevel approach with edge flags (SHortcut + ARC flags). Finally, CALT, CHASE and TNREF [146, 140] are combinations of Core-Based Routing + ALT, Contraction Hierarchies + Edge Flags and Transit-Node Routing + Edge Flags respectively.

Parallel Opportunities for Routing Algorithms

Most of the algorithms deployed during the preprocessing phase of the routing approaches are shortest-paths solutions, or slightly modified variations. The following list shows a brief description of the algorithms that may be present in a preprocessing phase together with the notation we have used to address them:

- SSSP_f, any SSSP algorithm, label-setting or label-correcting, which search traverses the edges using the natural direction (forward), computing shortest paths from a node to its successors.
- SSSP_b, any SSSP algorithm, label-setting or label-correcting, which search traverses the edges using the opposite direction (backward), computing shortest paths from the predecessors of a node v to v.
- LS-SSSP, a label-setting SSSP algorithm.
- FW, any variant of the Floyd-Warshall algorithm.
- GP, any graph partitioning algorithm.

Alg.	Preprocessing step	Π-Approach
HH	Candidate search	*Π-Prod.* of LS-SSSP$_f$
	Candidate eval.	*Π-Eval.*
	Network contraction	*Π-Spec.*
DT	Distance comp.	*Π-Alg.* of FW or *Π-Prod.* of SSSP$_f$
HNR	Subset V' selection	*Π-Prod.* of LS-SSSP$_f$ and *Π-Eval.*
	Covering set comp.	*Π-Prod.* of SSSP$_f$
	Network creation	*Π-Eval.*
CH	Ordering	*Π-Eval.*
	Contraction	*Π-Spec.*
TNR	Subset T selection	*Π-Alg.* of FW or *Π-Prod.* of SSSP$_f$
	Access dist. comp.	*Π-Prod.* of LS-SSSP$_f$
	Transit dist. comp.	*Π-Alg.* of FW or *Π-Prod.* of SSSP$_f$
HL	Label creation	*Π-Eval.* and *Π-Spec.*
ALT	Landmark selection	*Π-Prod.* of SSSP$_f$
	Distance comp.	*Π-Prod.* of SSSP$_f$ and SSSP$_b$
GC	Container comp.	*Π-Prod.* of SSSP$_f$
EF	Region definition	*Π-Alg.* for GP
	Edge-flags comp.	*Π-Prod.* of SSSP$_b$
Reach	Distance comp.	*Π-Alg.* of FW or *Π-Prod.* of SSSP$_f$
PCD	Cluster definition	*Π-Alg.* for GP
	Distance comp.	*Π-Prod.* of SSSP$_f$

Table 2.6: Different parallel strategies that can be applied to the preprocessing phase of the most relevant routing algorithms for roadmaps.

The first intuition for parallelizing the preprocessing phase of a routing approach is to parallelize the sequential algorithm they used before (*Π-Alg.*).

Additionally, parallel productivity-based approaches (*Π-Prod.*) can also be deployed if an algorithm has to be applied for each element h of a subset H, using all possible productivity formulations. Remember that some of these formulations include the combination of concurrently executing different instances of parallel algorithm, as we have seen for the APSP problem in Sect. 2.5.2.

There are preprocessing computations that are simple checks or element evaluations where no parallel algorithm is needed. However, we can apply the parallel productivity-based concept in these cases where every evaluating operation can be computed independently from the rest (*Π-Eval.*).

Finally, some preprocessing procedures have an inherent sequential nature in where each iteration needs to know the results of the previous one. For that special cases it could be designed a particular algorithm following the ideas of both parallel productivity-based approaches and speculative techniques. This algorithm would consider each iteration as independent of the rest, and it would compute the following iteration by distributes speculative possibilities of it to the available computational devices of the preprocessing platform. When the results needed to proceed with the following iteration are available, the computational devices will store or discard the computations they made according to these

values. We refer to this possible parallel solution as *Π-Spec.*.

Table 2.6 shows a summary of the parallel approaches we can apply in order to parallelize the preprocessing phases, of routing algorithms, described in the previous section.

2.7 Summary

The study of parallel approaches that solve shortest-paths problems has shown that the last trends use the Graphics Processing Units (GPUs) due to their powerful capabilities. The exploitation of these devices in graphs with non-negative edge weights is one of the main objectives of this Ph.D. thesis. Our work is focused on this kind of graphs due to the wide variety of real-world graphs that fulfills this property.

In the context of the Single-Source Shortest-Path (SSSP) problem, the last state-of-the-art implementation is presented in the work of Martín *et al.* [23]. This work is a port of Dijkstra's algorithm for the GPU devices, but without full taking advantage of their powerful capabilities due to the sequential nature of the original algorithm. However, we have seen in the literature, how other authors found some mathematical formulations that allow to increase the number of situations where it is possible to apply parallel computations [24]. Implementing these formulations for a GPU approach, similarly to Martín *et al.* solution, can lead us to a new GPU parallel solution with better performance.

Additionally, we have not found any study, from those that present a GPU solution, that takes care about the important runtime configuration parameters that are needed to configure a GPU execution of the algorithm. The correct tuning of this approaches can lead us to solutions with lower execution times due to the proper use of the computational capabilities of this kind of hardware accelerators. The following chapter, Chapter 3, presents the description and development of the new GPU parallel solution, while Chapter 4 shows an exhaustive study of the runtime configuration parameters that are involved around the shortest-path context in general, and for our solution in particular.

In the context of the All-Pair Shortest-Path (APSP) problem, we have found that there are two different strategies to compute the paths and distances, dynamic-programming and productivity-based approaches, together with their corresponding parallelization versions. For the particular case of non-dense graphs, that is the focus of this Ph.D. thesis, the parallel approach of the productivity-based strategy usually delivers better performance than the parallelization of dynamic-programming APSP algorithms [43, 22, 104, 106]. We have found different parallel approaches inside this productivity-based strategy that are classified according to the dimension they try to parallelize [115], but not all of them have been studied and/or implemented in the solutions of the literature. Since the last trends of High Performance Computing (HPC) are focused on using all heterogeneous computational resources available on a processing environment, one of the efforts of this Ph.D. thesis is to study and develop both homogeneous and heterogeneous implementations, following this yet unexplored Partitioned Source-parallel approach to solve the APSP.

These contributions have been published in the following papers:

1. "Parallel Approaches to the Shortest Path Problem - A Survey," H. Ortega-Arranz, Y. Torres, D. R. Llanos, and A. Gonzalez-Escribano, *To be submitted to ACM Computing Surveys*

2. “The Shortest Path Problem: Analysis and Comparison of Methods,” H. Ortega-Arranz, D. R. Llanos, A. Gonzalez-Escribano, *Book*, 1st edition, ser.(Synthesis Lectures on Theoretical Computer Science series), Morgan & Claypool.
Online, DOI: 0.2200/S00618ED1V01Y201412TCS001

Chapter 3

Using GPUs to solve the Single-Source Shortest-Path Problem

An important current trend in parallel computation is to exploit the use of hardware accelerators, such as Graphics Processing Units (GPUs). Their powerful capabilities have triggered their massive use to speed up highly parallel computations. Programing for these devices has been simplified by the introduction of high-level data parallel languages, such as CUDA [21]. A CUDA executable program has some configuration and execution parameters, such as the threadblock size, and the L1 cache state, whose wise combined use can lead to significant performance gains. The application of GPGPU (General Purpose computing on GPUs) to accelerate problems related with shortest-path problems has increased during the last few years. Some GPU solutions to the SSSP problem have been previously developed using different algorithms, such as Dijkstra's algorithm in [22, 23].

This chapter presents a new parallel approach to solve the SSSP problem in GPUs, for non-negative edge weights, based on Crauser's algorithm [24]. We present an experimental comparison of our algorithm with both its sequential version on CPU, and the fastest parallel GPU implementation due to Martín *et al.* [23]. Additionally, our GPU version has been enhanced by applying two CUDA optimization techniques: A proper selection of the threadblock size, and the configuration of the L1 cache memory. The results of this GPU-optimized version are compared with an optimized implementation of Dijkstra's algorithm taken from the Boost Graph Library [25]. We have used three different CUDA architectures (Fermi GF100, Kepler GK104, and Kepler GK110B) for our experimentation, also making an architectural comparison of them in order to distinguish which one is better for each kind of graph.

3.1 Defining the Frontier Set and the $\triangle$ Threshold

As we stated in the previous chapter, Dijkstra's algorithm, in each iteration i, calculates the minimum tentative distance between all the nodes that belong to the unsettled set, U_i. The node with the minimum tentative distance of U_i will be the next frontier node (see Dijkstra's algorithm in Sect. 2.2.2). If there are several unsettled nodes with the same minimum tentative distance, the sequential algorithm will settle them one by one. As

Algorithm 5 Pseudo-code of Martín's GPU implementation for Dijktra's algorithm.

```
1: while (Δ ≠ ∞) do
2:    gpu_kernel_relax(U, F, δ);                    //Edge relaxation
3:    vec_minimals = gpu_kernel_minimum(U, δ);      //Settlement step_1
4:    Δ = min( vec_minimals )
5:    gpu_kernel_update(U, F, δ, Δ);                //Settlement step_2
6: end while
```

the graphs we are interested on do not have negative weights associated to the edges, the choice/order of these settlements does not affect the correctness of the algorithm results because it is impossible to reduce even more their tentative distance.

The label-setting parallelization of Dijkstra's algorithm is based on *safely settling* at the same time all possible unsettled nodes. The term *safely settling* means that after traversing the outgoing edges of this frontier set F_i, it is not needed anymore to recompute these results, because they are not wrong. We named Δ_i to a limit value, computed in each iteration i, that ensures that any unsettled node u with $\delta(u) \leq \Delta_i$ can be safely settled. Note that the bigger the value of Δ_i, the more parallelism could be exploited. These safely settled nodes will become the frontier nodes of the following iteration $i + 1$, creating a new *frontier set*, $F_i + 1$. The outgoing edges of all these current frontier nodes will be traversed in parallel to reduce the tentative distances of their adjacent nodes.

Description of the structures

Besides the basic structures needed to hold nodes, edges, and their weights, we define three vectors that are used to store node properties:

(a) $U[v]$, which stores whether a node v is an unsettled node;

(b) $F[v]$, which stores whether a node v is a frontier node; and

(c) $\delta[v]$, which stores the tentative distance from source to node v.

3.1.1 Martín's GPU Algorithm

In this subsection, we describe the GPU approach developed by Martín *et al.* [23]. In that work, the four Dijkstra's algorithm steps described in Sect. 2.2.2 are ported to a GPU programming model (see Alg. 5). It is composed of three kernels that execute the internal operations of the Dijkstra vertex *outer loop*.

The Frontier Set

The Martín *et al.* algorithm uses a conservative enhancement to increase the frontier set, inserting only those nodes with the same minimum tentative distance. According to the notation presented above, their frontier set on any iteration i, F_{i+1}, is composed of every node $x \in U_i$ with a tentative distance $\delta(x)$ equal to Δ_i, being $\Delta_i = \min\{\delta(u) : u \in U_i\}$:

$$F_{i+1} = \{x \in U_i / \delta(x) == \min\{\delta(u) : u \in U_i\}\}$$

Algorithm 6 Pseudo-code of the *relax kernel*.

gpu_kernel_relax(U, F, δ)

```
1: tid = thread.Id;
2: if ( F[tid] == TRUE ) then
3:    for all j successor of tid do
4:       if ( U[j] == TRUE ) then
5:           δ[j] = Atomic_min( δ[j], δ[tid] + w(tid,j) );
6:       end if
7:    end for
8: end if
```

Algorithm 7 Pseudo-code of the *update kernel*.

gpu_kernel_update(U, F, δ, Δ)

```
1: tid = thread.Id;
2: F[tid] = FALSE;
3: if ( U[tid] == TRUE ) and ( δ[tid] = Δ ) then
4:     U[tid] = FALSE;
5:     F[tid] = TRUE;
6: end if
```

After defining the frontier set, the authors have implemented two variants for the traversing process of the outgoing edges: (1) The Successors variant, that follows the traditional methods of the Dijkstra algorithm, traversing the outgoing edges of the frontier nodes; and (2) the Predecessors variant, that traverses the incoming edges of the unsettled nodes in a backward way, looking for frontier nodes.

Successors Variant

The successors variant iteratively deploys the three GPU kernels that execute the steps of Dijktra's algorithm, until the termination condition is fulfilled. At that point, all nodes have been either settled and their shortest path distances are known, or they are unconnected to the source (infinite shortest path distance).

- The *relax kernel* (Alg. 6) calculates new tentative shortest path distances for the adjacent unsettled successors of the current frontier nodes, $f \in F_i$. If these new computed distances are lower than the current one, the minimum is kept. A GPU thread is associated for each node in the graph. Each thread first checks if its assigned node u belongs to the current frontier set by checking the boolean state stored in $F[u]$. Then, those threads responsible of a frontier node f traverse the outgoing edges (f, v), reducing/relaxing the distances of the unsettled adjacent nodes, by checking if the new value, $\delta[f] + w(f, v)$, is lower than the previous one, $\delta[v]$. Note that, having several concurrent GPU threads relaxing distances from different frontier nodes, a race condition may happen, if two or more threads read the same previous tentative distance, $\delta[v]$, compute a lower one, $\delta[f_t]+w(f_t, v)$, but the higher value is

Algorithm 8 Pseudo-code of the *relax kernel* in Predecessors variant. Changes respect to the successors variant are colored in red.

gpu_kernel_relax_predecessors(U, F, δ)

```
1: tid = thread.Id;
2: if ( U[tid] == TRUE ) then
3:    for all j predecessor of tid do
4:       if ( F[j] == TRUE ) then
5:           δ[tid] = min( δ[tid], δ[j] + w(j,tid) );
6:       end if
7:    end for
8: end if
```

the last stored. For this reason it is necessary to ensure an "atomic behavior" when computing this relaxing process (see Line 5 of Alg. 6).

- The *minimum kernel* computes the minimum tentative distance of the nodes that belongs to the U_i set. To do so, they use the *reduce4* method included in the CUDA SDK [147], but applying a minimum comparison instead of a sum operation. In a previous step of the reduction process, each thread resets its corresponding value of the shared memory to infinite, and it will not be updated/reduced for the settled nodes when reading from the δ vector for performing the reduction. In this way, the settled nodes are left out from the minimum computation process. This kernel returns an array of a few minimum values, that are lately reduced in the CPU. The resulting value of this final reduction is the Δ_i limit, that in Martín's algorithm corresponds with the minimum tentative distance available in the iteration i. If all nodes have been settled, the *minimum kernel* returns an array of infinite values, that means that either all nodes have been already settled, or the remaining ones are not connected to the source node. Remember that, at the beginning of the algorithm, all tentative distances were initialized to infinite.

- The *update kernel* (Alg. 7) settles the nodes that belong to the unsettled set, $v \in U_i$, whose tentative distance, $\delta(v)$, is equal to Δ_i. This task extracts the settled nodes from U_i. The resulting set, U_{i+1}, is the following-iteration unsettled set. The extracted nodes are added to F_{i+1}, the following-iteration frontier set. Each GPU thread checks, for its corresponding node v, whether $U(v) \wedge \delta(v) = \Delta_i$. If so, it assigns v to F_{i+1} and deletes v from U_{i+1}, changing the corresponding elements of the flag arrays.

Predecessors Variant

This variant differs from the traditional Successors variant in the way it reduces the tentative distances of the unsettled nodes. That is, for every unsettled node, the Predecessors algorithm checks whether any of its predecessor nodes belong to the current frontier set (see Alg. 8). In that case, if the new distance through this frontier node is lower than the previous one, the tentative distance is relaxed. The GPU predecessors implementation assigns a single thread for each node in the graph. The *relax kernel* only computes those

threads assigned to unsettled nodes $u \in U_i$. Every thread traverses back the incoming edges of its associated node looking for frontier nodes.

3.2 Applying Crauser's Ideas to Increase the $\triangle$ Threshold

This section describes a proposal to parallelize the *outer loop* of Dijkstra's algorithm following the ideas of Crauser *et al.* [24] of incrementing Δ_i. Our proposed GPU approach will be named GPU Crauser in this dissertation. As explained above, the main problem of this kind of parallelization is how to identify as many nodes as possible that can be safely inserted in the following *frontier set*.

3.2.1 Crauser's Algorithm

Parallelizing the *outer loop* of Dijkstra's algorithm requires the identification of which nodes can be settled at once, and thus, can be used as frontier nodes concurrently. Martín's algorithm inserts into the following frontier set, F_{i+1}, all nodes with the minimum tentative distance in order to process them simultaneously. However, Crauser's algorithm introduces a more aggressive enhancement, allowing the insertion of nodes that have bigger tentative distances than the minimum δ_i into this frontier set. This leads to a higher number of frontier nodes that can be processed at the same time in the following iteration.

The algorithm needs to compute in each iteration i, for each node of the unsettled set, $u \in U_i$, the sum of: (1) its tentative distance, $\delta(u)$, and (2) its *Out-Crauser Value*, representing the minimum weight of its outgoing edges, $\omega(u) = \min\{w(u,z) : (u,z) \in E\}$. Then, from these computed values, it calculates the total minimum value, also called the Δ_i threshold: $\Delta_i = \min\{(\delta(u) + \omega(u)) : u \in U_i\}$. Finally, an unsettled node, u, can be safely settled, becoming part of the next frontier set F_{i+1}, only if its $\delta(u)$ is lower than or equal to the calculated threshold, $\delta(u) \leq \Delta_i$.

$$F_{i+1} = \{x \in U_i / \delta(x) \leq \min\{(\delta(v) + \omega(v)) : v \in U_i\}\}$$

Figure 3.1 shows a graph example where the *Out-Crauser Values* of the nodes s, a, b, and c are highlighted, as well as their associated edges. Figure 3.2 shows the first steps of the Crauser algorithm solving the SSSP for the source node s. When relaxing the frontier node s, we reach the nodes a, b, and c, so we relax their tentative distance, that has a previous value of infinite (see Fig. 3.2(b)). Dijkstra's algorithm would settle the node b, with $\delta(b) = 1$ as the minimum of all tentative distances. Martín's algorithm performs the same settlement, only inserting node b into the following frontier node, because there are no more unsettled nodes with the same minimum value ($\Delta_0 = \delta_0(b) = 1$). On the other hand, the Crauser algorithm, due to the larger Δ threshold ($\Delta_0 = \delta_0(b) + \omega(b) = 1 + 3 = 4$), can settle all reached nodes composing a bigger frontier set F_1 (see Fig. 3.2(c)).

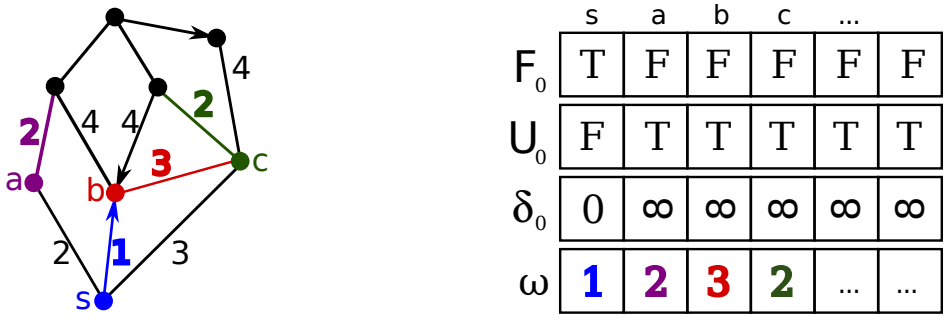


Figure 3.1: Examples of a graph with Crauser's out values, ω. Involved edges and weights are highlighted in the figure. A snapshot of the structures used by the algorithm in the first iteration ($i = 0$) is shown at the right of the figure.

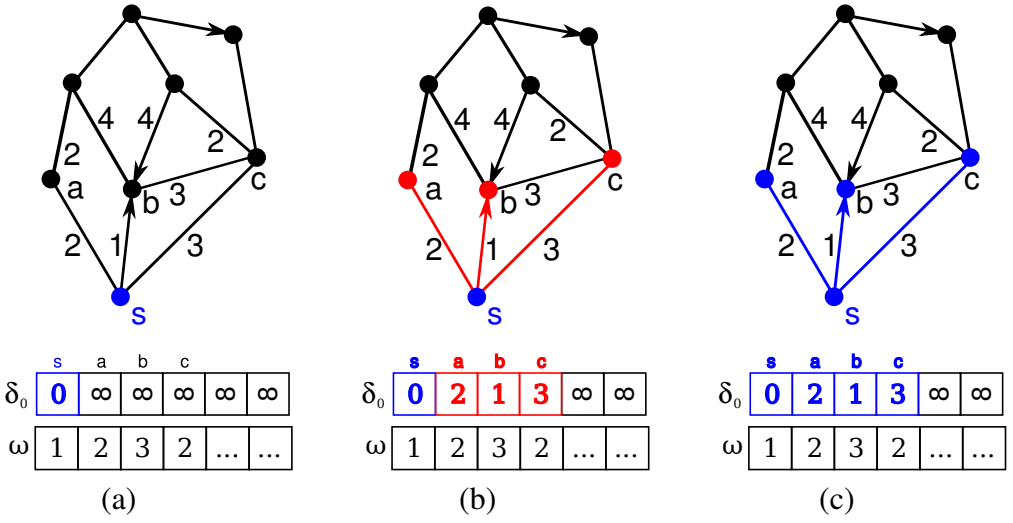


Figure 3.2: Crauser's algorithm steps: Starting point (a), edge relaxation (b), and settlement (c).

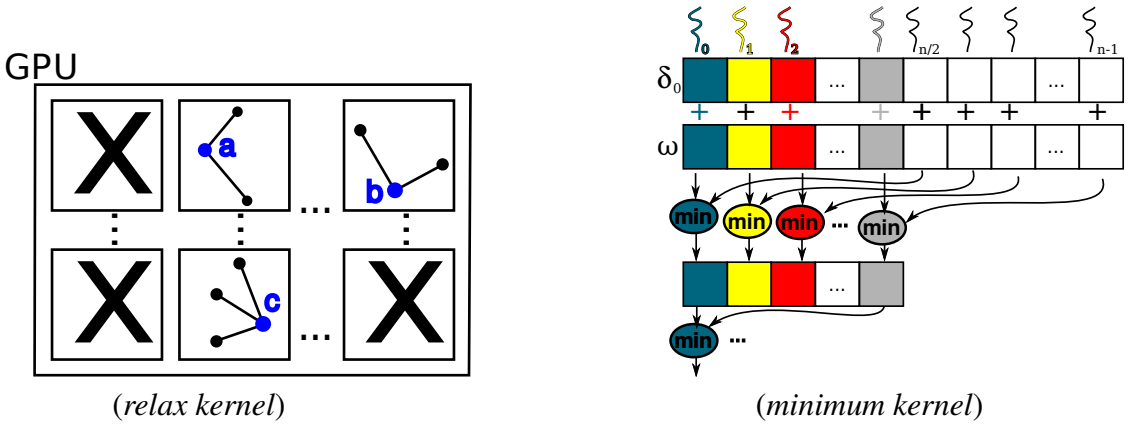


Figure 3.3: Graphical descriptions of the behavior of the GPU threads for: the *minimum kernel* (right), and *relax kernel* (left) in the second iteration ($i = 1$) for the graph used as example.

Algorithm 9 Pseudo-code of our Crauser *minimum kernel*, based on the *reduce4* method of the CUDA Software Development Kit (SDK). Additional code with respect to Martín's *minimum kernel* is colored in blue.

gpu_kernel_min(U, δ, ω, numVertices, dv_aux)

```
1: tid  = thread.Id;
2: tid2 = thread.Id + blockDim.x
3: lid  = local.thread.Id;

4: shared[lid] = inf;
5: __syncthreads();

6: int data1,data2 = inf;

7: if ( U[tid] == TRUE ) then
8:    data1 = δ[tid] + ω[tid];
9: end if

10: if ( tid2 < numVertices ) then
11:    if ( U[tid2] == TRUE ) then
12:       data2 = δ[tid2] + ω[tid2];
13:    end if
14: end if

15: shared[lid] = min( data1, data2 )

16: for ( int stride = blockDim.x/2; stride > 0; stride >>= 1 ) do
17:    if ( lid < stride ) then
18:       shared[lid] = min( shared[lid], shared[lid + stride] );
19:    end if
20:    __syncthreads();
21: end for

22: if ( lid == 0 ) then
23:    shared[lid] = min( shared[lid], shared[lid + 1] );
24:    shared[lid] = min( shared[lid], shared[lid + 2] );
25:    dv_aux[block.Id] = shared[lid];
26: end if
```

Algorithm 10 Pseudo-code of our Crauser *update kernel*. Modifications with respect to Martín's *update kernel* are colored in blue.

gpu_kernel_update(U, F, δ, Δ)

```
1: tid = thread.Id;
2: F[tid] = FALSE;
3: if ( U[tid] == TRUE ) and ( δ[tid] ≤ Δ ) then
4:     U[tid] = FALSE;
5:     F[tid] = TRUE;
6: end if
```

3.2.2 Porting Crauser's Ideas to a GPU Implementation

The implementations of both Martín's and Crauser's algorithms look very similar because they introduce a slight modification over the same base, the Dijkstra algorithm. One of the main differences is the computation of the *Out-Crauser Values*. These values are like a fixed "feature" of the nodes, since they do not change during the execution of the algorithm. Therefore, their calculation is carried out in a precomputation phase, before the algorithm starts the iterative process. The implementation of the first kernel of the loop, the *relax kernel*, is identical, since the edge traversing for relaxing distances is the foundation of Dijkstra's algorithm. Figure 3.3 (left) shows a graphical illustration of the behavior of the GPU threads when computing the *relax kernel* functions for the second iteration of the example graph proposed in Fig. 3.2, described in the previous subsection. The differences are in the computation of the Δ thresholds, located in the *minimum kernel*, and in the settling process based on these limit values, located in the *update kernel*:

- The *minimum kernel* computes the minimum tentative distance of the nodes that belongs to the U_i set, plus the corresponding Out-Crauser Values. Algorithm 9 shows the code based on the *reduce4* method of the CUDA SDK, and the modifications needed to apply the Crauser ideas. We insert an additional sum operation per thread before the reduction loop. Figure 3.3 (right) shows a graphical illustration of the behavior of the GPU threads when computing our modified *minimum kernel*. The resulting value of this reduction is the new increased Δ_i. Some NVIDIA CUDA devices can maximize the occupancy with threadblock sizes that are multiples of three. The modifications of lines 23-24 aim to make possible the compatibility of this reduction algorithm with more threadblock sizes. Not only sizes that are multiples of two, but also multiples of three.

- Algorithm 10 shows how each single GPU thread of the *update kernel* now checks, for its corresponding node v, its belonging to the current unsettled set, and if its tentative distance is lower to or equal than the Δ_i threshold ($U(v) \wedge \delta(v) \leq \Delta_i$). If that is the case, it assigns the node to the next frontier set and removes it from the unsettled set. This usually results in a bigger frontier set for the following iteration, F_{i+1}, or, in the worst case, equal to Martín's version.

3.3 Experimental Evaluation of the GPU-SSSP Algorithm

In this section, we describe: The methodology used to design and carry out experiments to validate our approach; the different scenarios we considered; the platforms and input sets used; and the experimental results obtained.

3.3.1 Methodology

We have evaluated three different sets of experiments with different objectives. The first set studies the relevance of our new GPU approach against the previous GPU algorithm of Martín *et al.* described in the state of the art. The second set of experiments compare our GPU approach against an optimized sequential version for CPUs. Finally, the third

set analyzes how the GPU architectural differences of the NVIDIA boards used affect the execution times for input sets with different characteristics.

For all the studied experiments, we have randomly selected 100 source nodes from the graph, using an uniform distribution, to solve 100 SSSP problems, obtaining an average of the execution times. The details of the input sets selected will be described in the following section.

A description of the platforms and devices used, the tuning values used for the optimized GPU version, and the experimental scenarios considered are presented below.

Target Architectures

We have selected NVIDIA GPU devices with two different CUDA architecture families (Fermi and Kepler), with very different features. The NVIDIA GPU devices used in our experimentation are placed in different host machines:

- The first host machine is an Intel(R) Core i7 CPU 960 3.20GHz, with a global memory of 6 GB DDR3, that runs an Ubuntu Desktop 10.04 (64 bits) OS. It contains two boards that we named:

  Fermi: a GeForce GTX 480 which CUDA architecture is the Fermi GF100, and

  Kepler: a GeForce GTX 680 which CUDA architecture is the Kepler GK104.

- The second host machine for the third board is an Intel(R) Xeon E5 2620 2.1GHz, with a global memory of 32GB DDR3 running an Ubuntu Server 14.04 (64 bits) OS. We named the board it contains:

  Titan: a GeForce GTX Titan Black which CUDA architecture is the Kepler GK110B.

The experiments have been launched using the 295.41 64-bit NVIDIA driver. The latter machine described is where the sequential CPU implementations have been executed, due to the big amounts of global memory available to allocate the relaxed heap structure. All programs have been compiled with gcc using the -O3 flag.

Proper Tuning Values for GPU Optimized Executions

The programmers of GPU devices have to decide by themselves the values for some running configuration parameters. CUDA programming guidelines [13, 21] give some recommendations in order to avoid values that lead to terribly non-efficient execution times. These recommendations are based on the correct use of the underlying hardware resources.

As it will be described in detail in the following chapter, there are still significant differences between different values in the ranges suggested by CUDA. Even there are cases where none of these recommended values are the optimal ones for a kernel execution. Best values can be detected by exhaustive trial-and-error test. However, we are going to use the values presented in Table 3.1 for our GPU optimized version. These values can be predicted using kernel characterization techniques, which are presented in Sect. 4.6.

size/kernel	deg2		deg20		deg200		≥ deg1000	
	F	K	F	K	F	K	F	K
24k relax	192-A	128-A	192-A	128-A	192-A	192-A	192-A	192-A
24k min	192-A	96-A	128/192-A	96-A	192-A	128-A	192-A	128-A
24k update	192-A	128-A	192-A	128-A	192-A	128-A	192-A	128-A
49k relax	192-A	256-A	256-A	256-A	256-N	256-A	256-N	256-A
49k min	192-A	96-A	128-A	96/128-A	192-A	256-A	192-N	256-A
49k update	192-A	128-A	192-A	128-A	192-A	256-A	256-N	256-A
98k relax	192-A	128-A	256-N	256-A	384-A	256-A	384-A	256-A
98k min	128-A	128-A	192-N	96-A	192-A	128-N	192-A	128-N
98k update	192-A	128-N	192-N	256-N	192-N	128-N	192-N	128-N

Table 3.1: Values selected for threadblock-size and L1 cache state through kernel characterization process for Fermi (F) and Kepler (K). L1 states are: (A) Augmented or (N) Normal.

Experiment I: Comparison Against a State-of-the-art Algorithm

From the suite of different implementations described in [23], we used as reference the fastest versions fully implemented for GPUs, that we name GPU Martín in this study. We have left out the slower ones; the hybrid approaches that execute some phases on the CPU and others on the GPU. The following paragraphs describe the experimentation scenarios we have considered:

Scenario A: GPU Martín vs GPU Crauser

The first evaluated scenario is designed to compare both GPU implementations, GPU Martín and GPU Crauser. Additionally, we have adapted our algorithm to the predecessors variant. This allows to make experimental comparisons with both, successor and predecessor, GPU Martín's variants.

To fairly compare the performance gain of our approach to Martín *et al.* results, we used in our approach the same CUDA configuration values used by them in their study: (1) 256 threads per block for all kernels; and (2) L1 cache normal state (16KB). We named this configuration the *default configuration*.

Scenario B: GPU Crauser vs Optimized GPU Crauser

The second scenario is designed to compare the best approach from Scenario A, our GPU Crauser successors variant using the *default configuration*, to its optimized version using the best values for the GPU parameters (see Table 3.1).

Additionally, we compare this last GPU version against the sequential execution of Crauser's algorithm ported for CPUs (CPU Crauser).

Experiment II: Boost Graph Library Comparison

We compare the results of our tuned approach with an optimized implementation of Dijkstra's algorithm in a state-of-the-art library: The Boost Graph Library [25]. This sequential implementation uses Relaxed Heaps as the structure to implement the priority queues where the reached nodes are stored and sorted. This comparison between both optimized approaches will show what are the conditions for which each different approach is better.

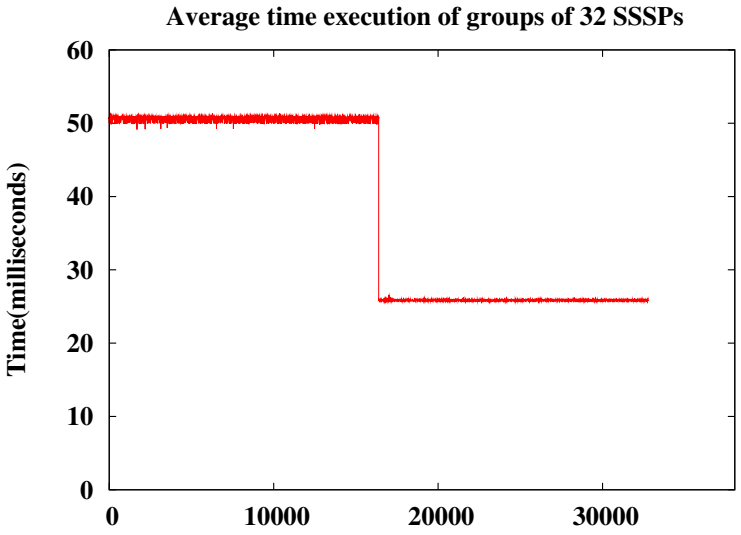


Figure 3.4: Temporal cost of the different source nodes in the graph for the Kepler GPU.

Experiment III: NVIDIA Architectural Comparison

Since the characteristics and features of the upcoming GPUs are significantly different, such as the number of cores (from 480 to 2880), the clock rate frequency of these cores (from 1.4 Ghz to 0.98 Ghz), or the bandwidth of memory transfers, among others, it is relevant to study how these variations affect to the final execution times. We compare the results obtained launching the optimized GPU Crauser version on the different experimental boards (Fermi, Kepler, and Titan), using the synthetic-random graph collection and benchmarking/real-world networks. This comparison will show which are the architectural features that better fit with the graph properties.

3.3.2 Input set characteristics

In this section we describe the three different input sets used for our experiments. The first one is due to Martín *et al.* [23]. We used for the first experiment (Experiment I) the input set of synthetic graphs of Martín *et al.*'s study [23] to fairly compare the performance differences between both GPU algorithms, GPU Martín vs GPU Crauser.

- **Martín's graphs:** These graphs have sizes that range from 2^{20} to $11 \cdot 2^{20}$ vertices. This kind of graphs are very sparse, as we have kept the fan-out degree they chose, $d^{+}(v) = 7 : v \in V$. The generator tool creates seven adjacent predecessors for each vertex. They inverted the generated graphs in order to study approaches based on the successors version. The edge weights are integers that randomly range from 1 to 10. As these graphs present a biased distribution of the nodes, where the first half of vertices need more time to compute the solution than the second half (see Fig. 3.4), we will use the following graph input sets for the remaining experiments.

The two remaining graph collections have been used in all designed experiments. The second input set is composed by a collection of random graphs generated with a tech-

nique designed to produce random structures with specific properties. The third input set contains real-world and benchmarking graphs provided by research institutions.

- **Synthetic random graphs:** We used the random-graph generation technique presented in [148] to create a second input set. This decision was taken in order to: (1) avoid dependences between a particular graph structure of the input sets and performance effects related to the exploitation of the GPU hardware resources; and (2) avoid focusing on specific domains, such as road maps, physical networks, or sensor networks among others, that would lead to a loss of generality.

  Briefly described, this technique generates random graphs for a given cardinality of the vertex set, and a given edge-density of the graph. However, the process of adding edges between any pair of nodes is completely random, so it is possible that the generated graph is not fully connected. To solve this issue, after the creation process, we modify the first edge of each node of the graph to force its connection with the node with the next number on an arbitrary-chosen ordering. This produces a *line* of all nodes of the graph. Note that this modification ensures the complete connectivity of the graph.

  In order to evaluate the algorithmic behavior for some graph features, we have generated a collection of graphs using three sizes (24 576, 49 152, and 98 304) and five fan-out degrees (2, 20, 200, 1 000, and 2 000). We named random24k, random49k, and random98k, to the collection of graphs with 24 576, 49 152, and 98 304 nodes, respectively. These sizes, smaller than Martín's graphs, were chosen with the aim of discovering the threshold where the sequential CPU version executes faster than the GPU. Weights are integers randomly chosen, and uniformly distributed in the range $[1 \ldots 100]$.

- **Dimacs and social-network graphs:** We used this set of graphs in order to observe if the evaluated approaches not only performs well in synthetic laboratory graphs but also in real contexts. We used some of the real-world and benchmarking graphs facilitated by DIMACS [149, 150], such as the walshaw graphs (low degree), clustering graphs (low degree), rmat-kronecker graphs (medium-high degree), and the social-networks graphs (medium degree), including also a graph based on the flickr structure provided by [151].

All the graphs of every input set used are stored using the Compressed Sparse Row (CSR), depicted in Fig. 3.5, that involves the following elements:

1. Vector `nodes[ ]` with length `n+1`, which stores in the position `x` the index of the vector `edges[ ]` where the list of adjacent nodes of vertex x starts. The list of adjacent nodes for node `x` ends in the index value stored in `nodes[x+1]`.

2. Vector `edges[ ]` with length `m`, which stores the adjacency nodes, the tail nodes of the edges.

3. Vector `weights[ ]` with length `m`, which stores the weights associated to the edges of `edges[ ]`.

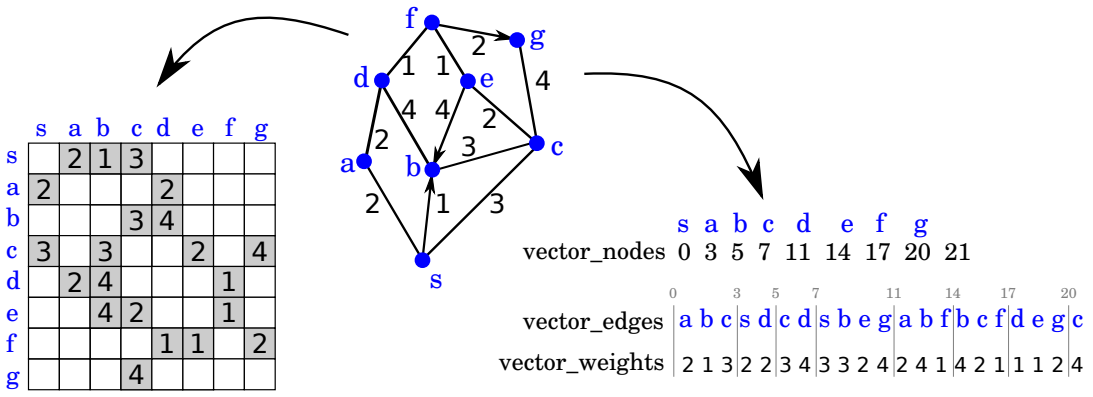


Figure 3.5: Example of a graph representation using an adjacency matrix (left) and the Compressed Sparse Row (CSR) storage format (right).

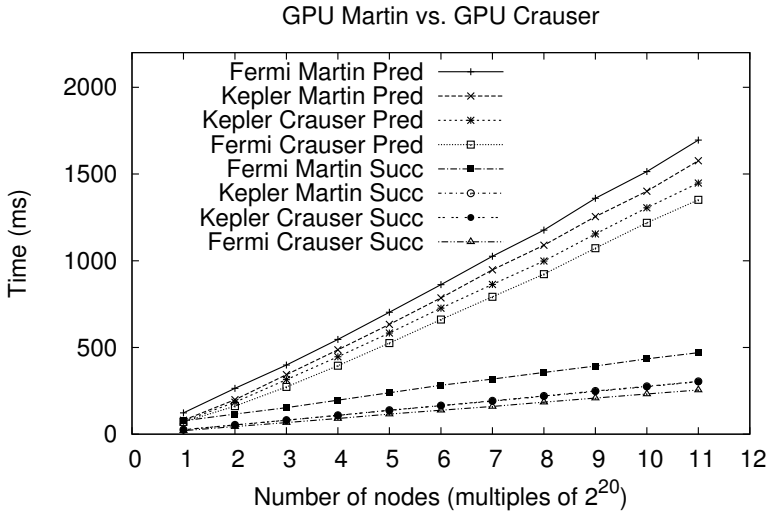


Figure 3.6: Scenario A.1: Execution times of GPU Martín vs GPU Crauser using Martín graphs.

3.3.3 Experimental Results I - State of the Art Comparison

Scenario A: GPU Martín vs GPU Crauser

A.1 - Martín Graphs: The execution times for both parallel algorithms, GPU Martín and GPU Crauser, with their respective variants, Successors and Predecessors, carried out in the different CUDA architectures, Fermi and Kepler, are shown in Fig. 3.6. Due to the extreme sparse nature of the graphs, there are not so many possibilities to take advantage of the parallelism present in Crauser's algorithm. Therefore, Fermi board, with a higher clock rate, delivered better results than Kepler. Performance improvement in Fermi board, over Martín's algorithm, goes from 20.29%, for the Predecessors variant, up to 45.80%, for the Successors variant. On the other hand, these improvements are not so significant in the Kepler GK104 architecture, involving just 1.07% and 8.14% for Successors and Predecessors variants respectively.

A.2 - Random Graphs: Figure 3.7 shows the execution times for the synthetic random graphs. The x-axis represents the fan-out degree of the graphs in logscale. Note that there are additional results for the 98k case (degree 5, 10, 50, 100, and 500) shown in this figure, included in order to obtain smoother plots and clarify the trends and thresholds. In order to clarify the figure, we have only shown the results obtained with the CPU Crauser version, and the results of our GPU Crauser version executed in Titan, because the execution times for the remaining GPU boards present the same trends related to size and degree. Regarding the size, as expected, the execution times increase as the graph size gets bigger. Having more nodes in the graph implies that there are more distance combinations to be computed.

However, the algorithms have a complex behavior depending on the degree. The graphs with a lower degree (2 to 20) give fewer possibilities to take advantage of the algorithm parallelism because, in each iteration, there are fewer nodes that can be inserted in the next frontier set. For the GPU Martín algorithm, this fact leads to worse execution times than the tested sequential version (CPU Crauser).

As the degree increases in the graphs, all methods reduce their execution times, more drastically for the parallel ones. However, when higher degrees are reached (1 000 and 2 000) the execution times rise again. Although it could seem that, with a higher degree, it may be possible to take better advantage of the parallel algorithms, the computation performed in the relax kernel increases because there are more distances to be checked. Additionally, this checking operation must be done with an atomic instruction serializing the execution of two or more threads that try to access the same memory position simultaneously.

The speedups obtained with our GPU Crauser solution versus (a) the sequential implementation, and (b) the GPU Martín version, are shown in Table 3.2. The most representative results for synthetic random graphs appear for the ones with a lower degree, up to 18.42× against the CPU, and up to 32.20× compared to the GPU Martín.

A.3 - Real-world Grahps: Figure 3.8 shows, in logarithmic scale, the execution times for real-world and benchmarking graphs. All approaches have similar trends and behaviors to those in synthetic random graphs, except for the GPU Martín algorithm, that cannot beat the CPU Crauser version for some graph families. Our GPU approach still returns the fastest execution times.

The speedups obtained with our GPU Crauser solution versus (a) the sequential implementation, and (b) the GPU Martín version, are shown in Table 3.3. The highest speedups obtained, for the benchmarking/real-world set, are up to 45.38× and up to 130.25× against the CPU Crauser and the GPU Martín approaches, respectively.

Scenario B: GPU Crauser vs Optimized GPU Crauser

B.1 - Martín Graphs: Figure 3.9 shows the execution times for the fastest variant, the GPU Crauser Successors variant, using the default and the optimized configurations. Experiments are carried out in the Fermi and Kepler boards. The use of parameter tuning techniques, which take better advantage of the GPU hardware resources, leads to faster execution times of up to 9.75% for Fermi, and up to 5.11% for Kepler architectures.

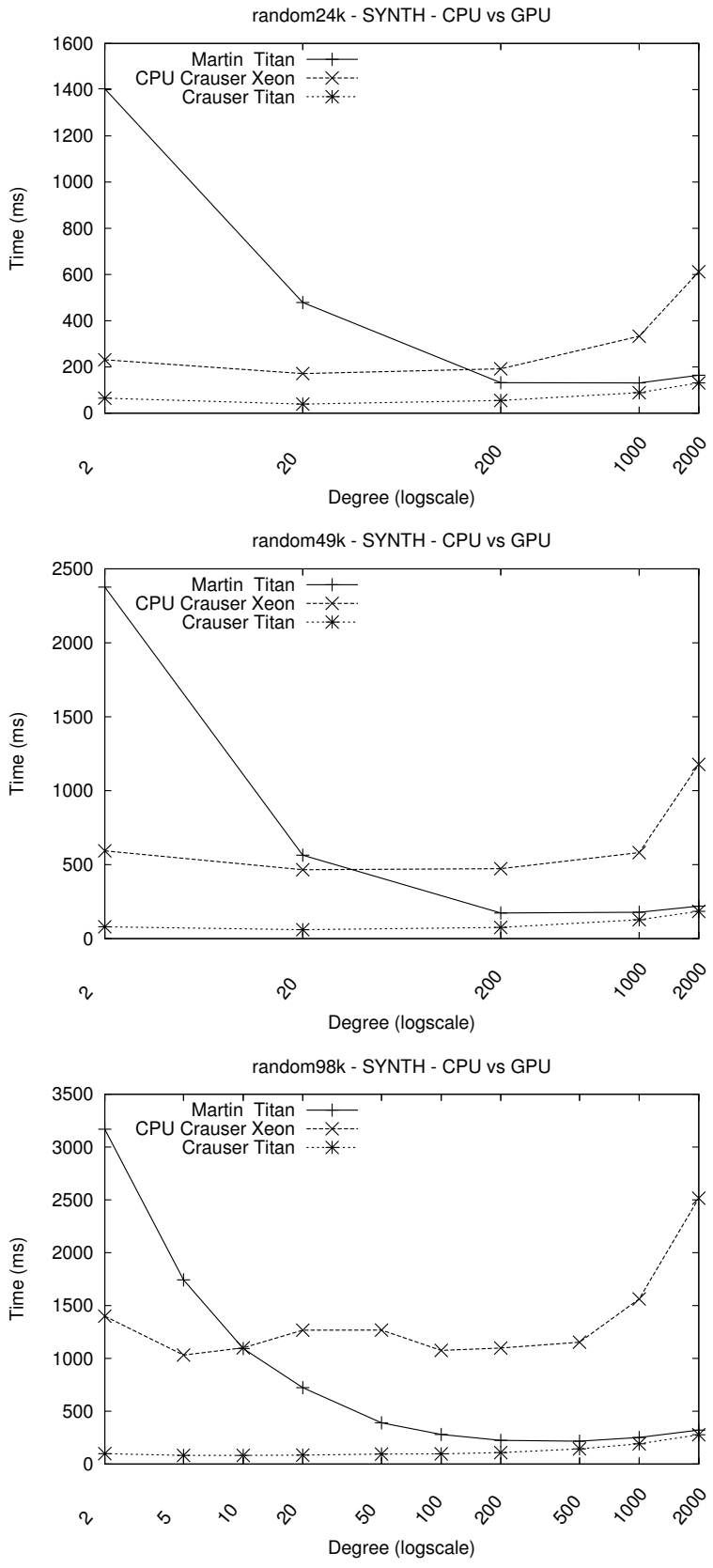


Figure 3.7: Scenario A.2: Execution times of GPU Martín vs GPU Crauser using synthetic random graphs.

	Fermi GPU C vs.		Kepler GPU C vs.		Titan GPU C vs.	
	CPU C	**GPU M**	**CPU C**	**GPU M**	**CPU C**	**GPU M**
24k-d2	5.90×	24.61×	5.54×	24.66×	3.56×	21.62×
24k-d20	6.07×	11.04×	5.98×	11.23×	4.33×	12.11×
24k-d200	3.84×	2.50×	3.94×	2.39×	3.49×	2.40×
24k-d1000	3.00×	1.43×	3.08×	1.40×	3.76×	1.48×
24k-d2000	3.30×	1.22×	3.51×	1.21×	4.66×	1.25×
49k-d2	10.78×	29.97×	10.47×	29.20×	7.39×	29.62×
49k-d20	10.55×	9.66×	10.63×	9.62×	7.76×	9.40×
49k-d200	5.93×	2.18×	6.32×	2.10×	6.26×	2.30×
49k-d1000	3.62×	1.35×	3.80×	1.32×	4.55×	1.40×
49k-d2000	4.85×	1.20×	4.98×	1.18×	6.39×	1.19×
98k-d2	18.42×	31.85×	18.01×	31,70×	14.22×	32.20×
98k-d20	17.33×	7.75×	17.63×	7.73×	14.97×	8.55×
98k-d200	8.23×	1.87×	9.07×	1.86×	10.18×	2.07×
98k-d1000	5.91×	1.27×	6.13×	1.24×	8.16×	1.31×
98k-d2000	6.23×	1.13×	6.32×	1.13×	9.09×	1.16×

Table 3.2: Scenario A.2: Speedups of GPU Crauser (GPU C) vs. the sequential implementation (CPU C), and vs. GPU Martin (GPU M) for the used GPU boards. Speedups above 10× are highlighted in grey.

	Fermi GPU C vs.		Kepler GPU C vs.		Titan GPU C vs.	
	CPU C	**GPU M**	**CPU C**	**GPU M**	**CPU C**	**GPU M**
walshaw	19.23×	27.99×	19.22×	28.00×	16.57×	29.20×
clustering	25.43×	23.97×	26.68×	26.63×	25.41×	30.61×
kronecker	27.09×	3.78×	30.76×	4.05×	43.05×	5.21×
social net.	33.30×	129.71×	37.09×	130.25×	45.38×	125.99×

Table 3.3: Scenario A.3: Best speedups obtained for each real-world/benchmarking family graph of GPU Crauser (GPU C) vs. the sequential implementation (CPU C), and vs. GPU Martín (GPU M).

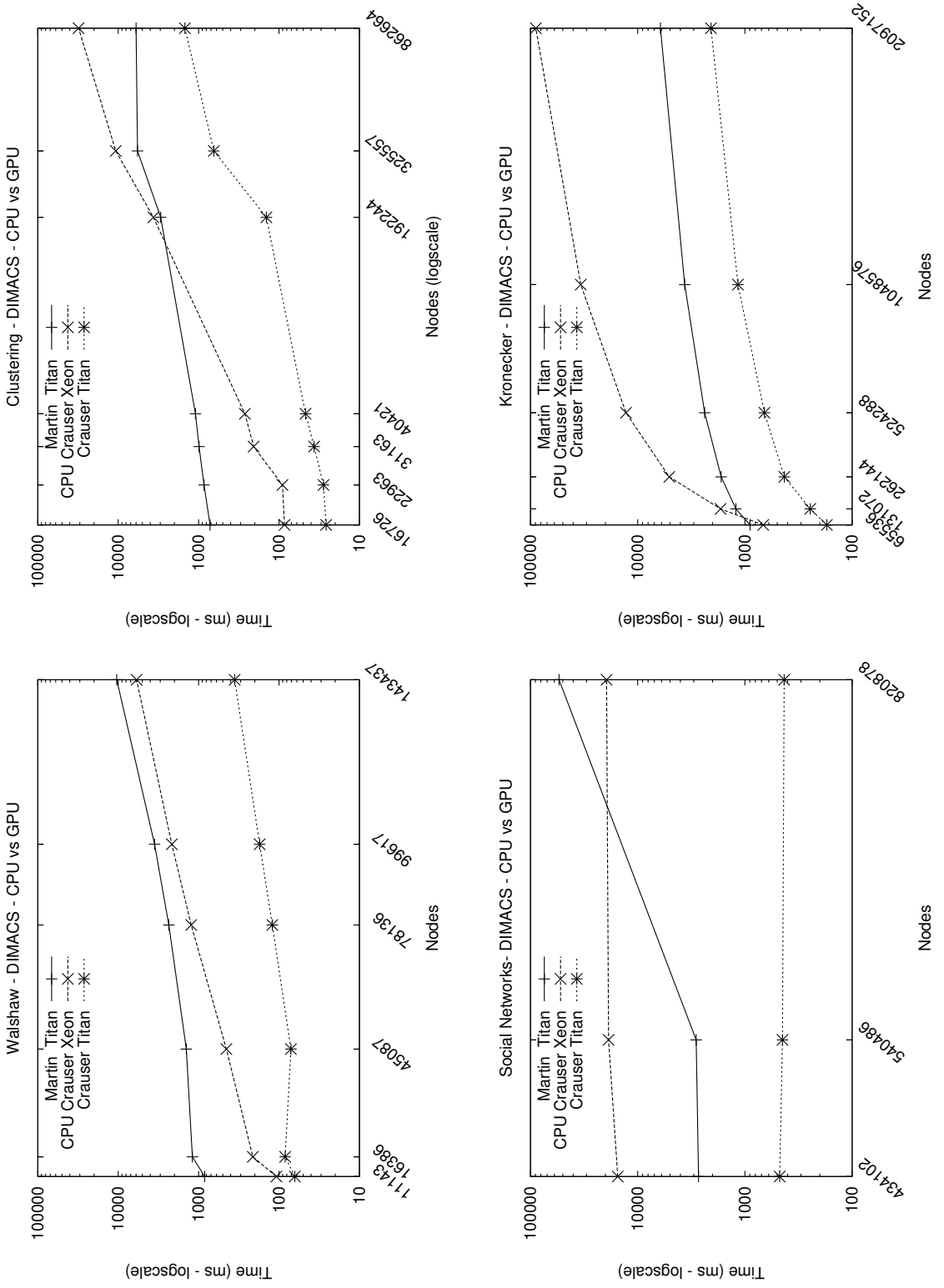


Figure 3.8: Scenario A.3: Execution times of GPU Martín vs GPU Crauser using real-world graphs.

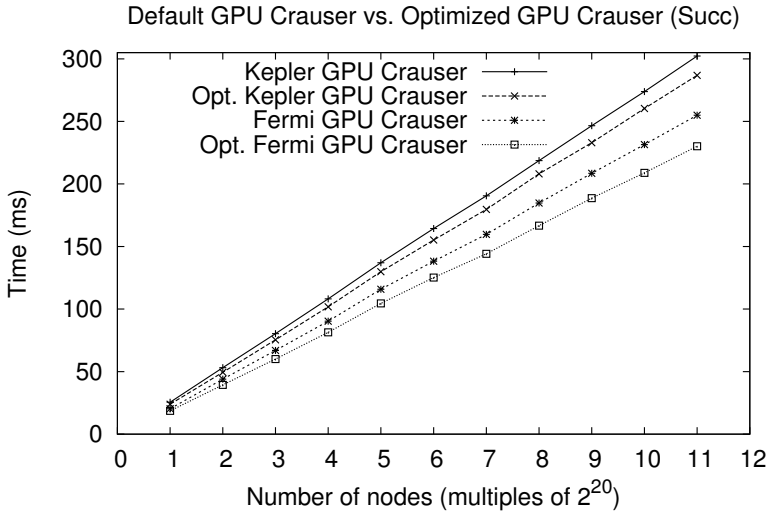


Figure 3.9: Scenario B.1: Execution times of GPU Crauser vs its optimized version using Martín graphs.

Comparing the execution times of the optimized GPU Crauser Successors variant with its analogous sequential version, the CPU Crauser Successors, we observe that the use of the GPU offers a speedup of up to 26.68×, for Fermi's architecture, and up to 21.4×, for Kepler's architecture.

B.2 and B.3 - Random and Real-world Grahps: Figure 3.10 shows that the use of the parameter tuning technique to choose optimized configuration parameters (thread-block size and L1 cache configuration) leads to significant percentages of performance improvements for this set of graphs. The most significant are 14.39% for Fermi, 22.28% for Kepler, and 22.93% for Titan (see Table 3.4). Note that, for the particular scenario of graph 49k with degree greater than 200, there is hardly any improvement because the proper values selected coincide with those used in the *default configuration*.

The performance gains obtained for real-world graphs are up to 9% for walshaw graphs and almost 6% for social networks and kronecker graphs (see Fig. 3.11).

Applicability of configuration parameter values to other architectures

In all the previous experimental results where the Kepler board (GK104 Kepler architecture) and the Titan board (GK110B Kepler architecture) appear, we can see that the configuration parameter values classified as appropriate values for the Kepler board are also applicable to Titan with the same trends. This happens because they belong to the same CUDA architecture family.

3.3.4 Experimental Results II - Boost Graph Library Comparison

Figure 3.12 shows, for the synthetic random graphs, the execution times of the sequential reference Dijkstra Boost library implementation [25] that uses relaxed heaps, and our op-

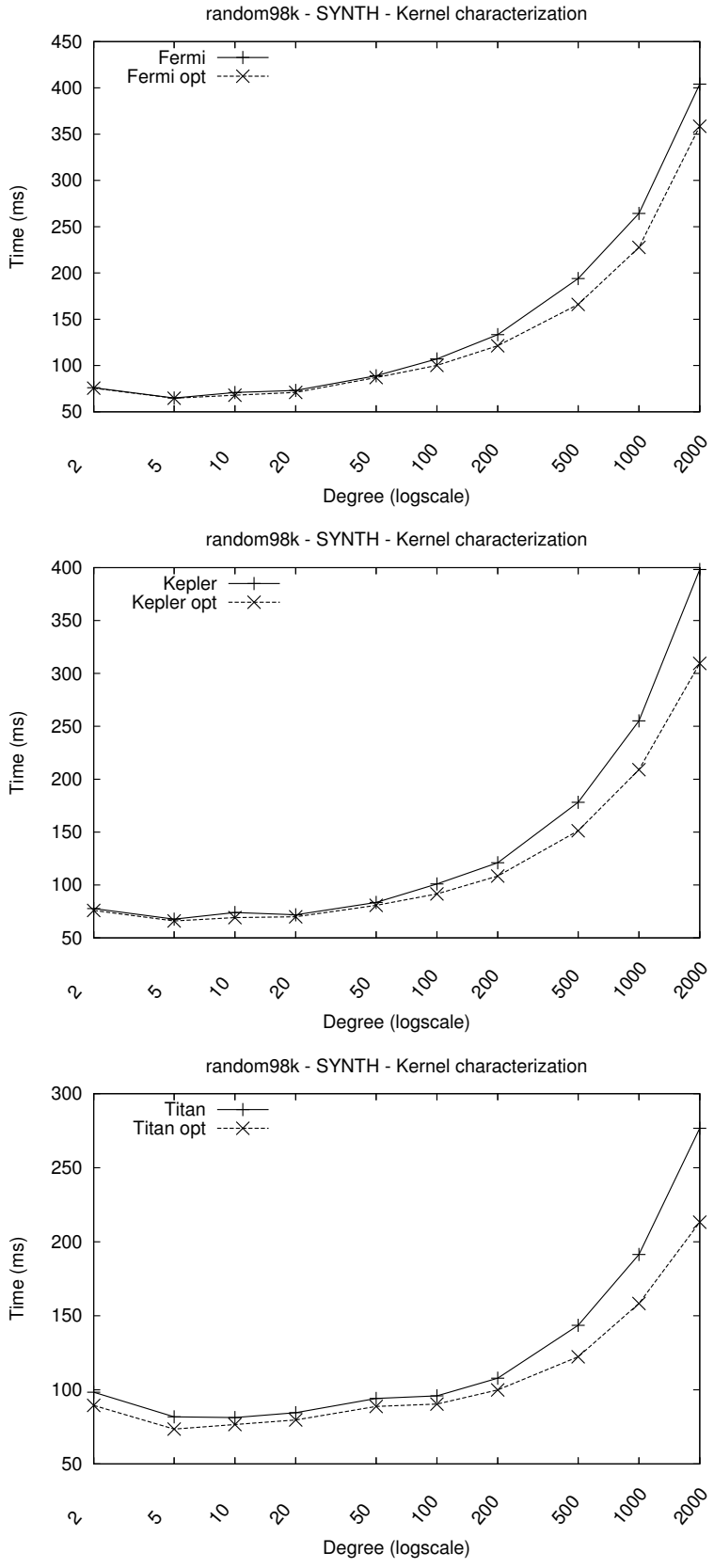


Figure 3.10: Scenario B.2: Execution times of GPU Crauser vs its optimized version using random graphs in the Fermi, Kepler, and Titan boards.

GPU	24k			49k			98k		
boards	≤20	200	>200	≤20	200	>200	≤20	50-200	>200
Fermi	1.8%	2.6%	3.3%	4.4%	7.0%	0.2%	4.2%	8.9%	14.4%
Kepler	0.9%	4.7%	13.7%	2.5%	7.3%	17.1%	6.5%	10.4%	22.3%
Titan	2.3%	6.3%	10.4%	6.6%	7.2%	16.7%	10.2%	7.4%	22.9%

Table 3.4: Scenario B.2: Synthetic random graphs; relative improvements on the execution time between GPU Crauser using default configuration parameters, and optimized configuration parameters. Gains above 10% are highlighted in grey.

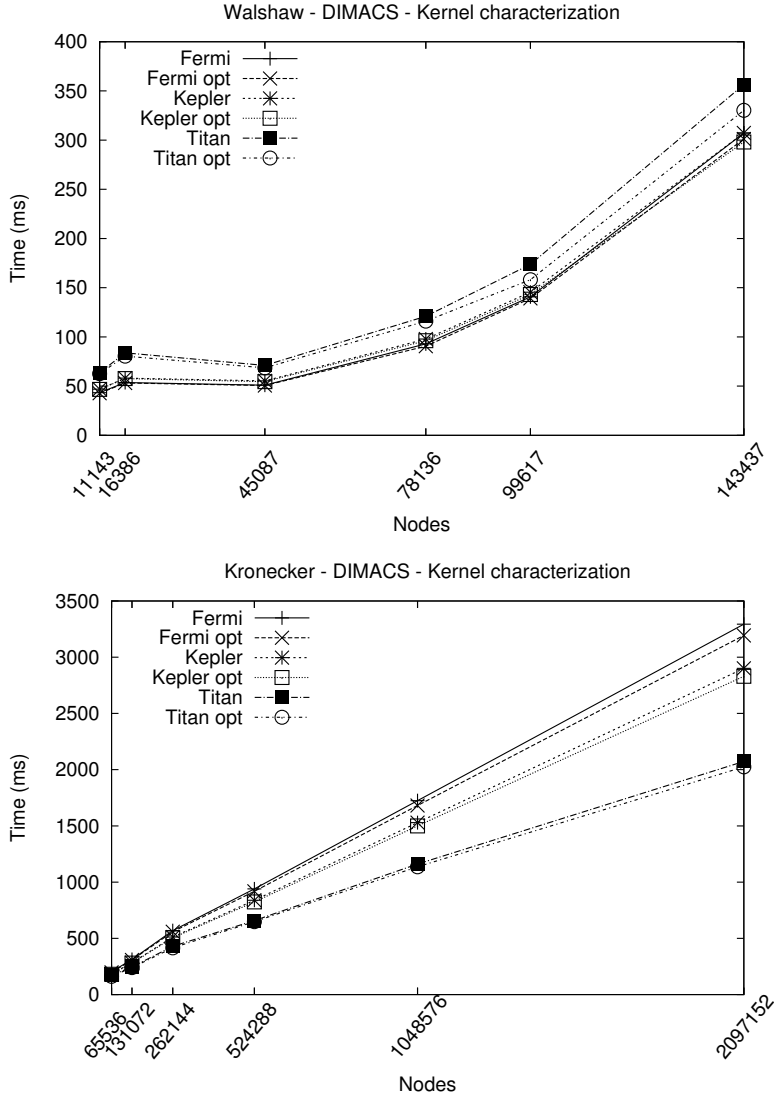


Figure 3.11: Scenario B.3: Social networks and kronecker graphs; execution times of GPU Crauser using default configuration parameters, and the optimized configuration parameters.

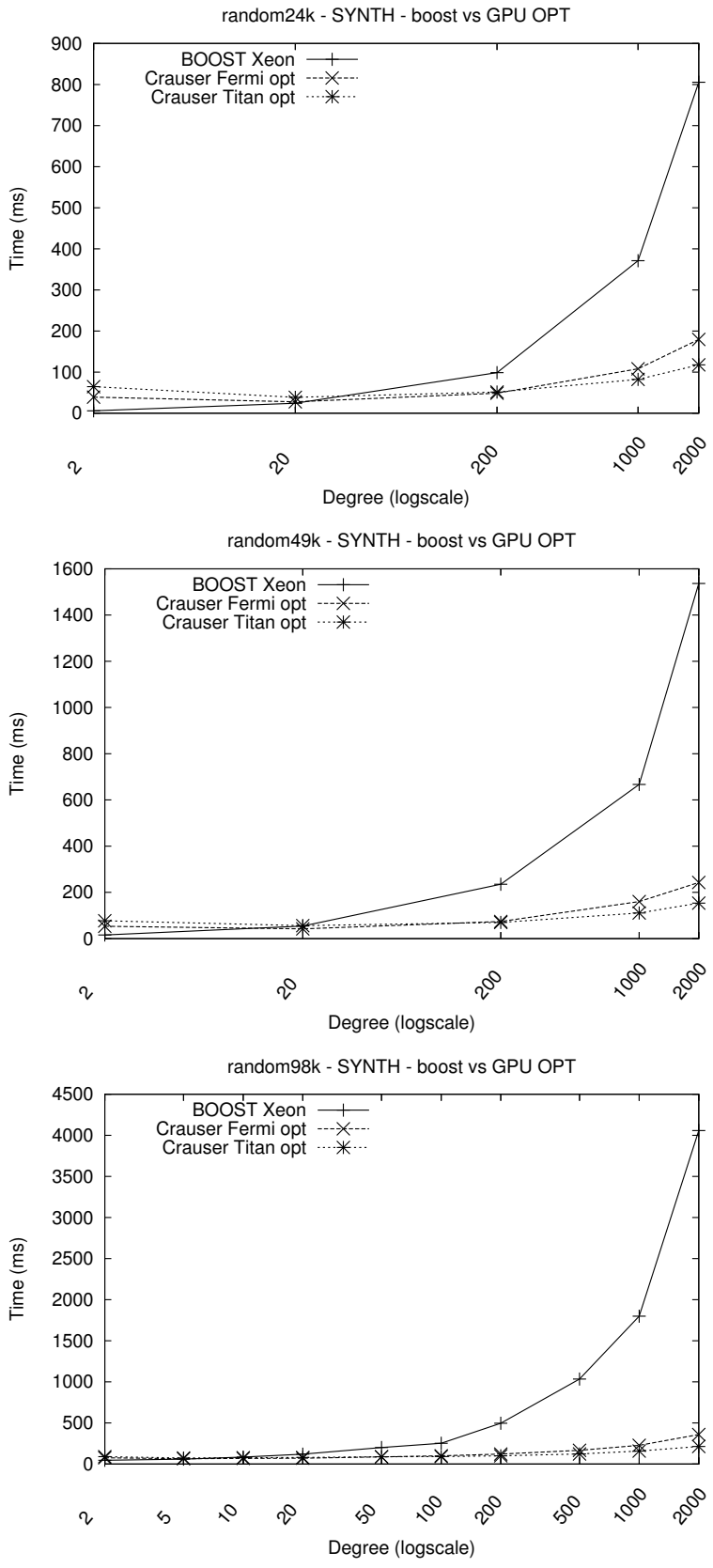


Figure 3.12: Experimental Results II: Execution times of the optimized GPU Crauser implementation, executed on Fermi and Titan boards, versus the optimized sequential Dijkstra's algorithm included in the Boost Graph Library, for synthetic random graphs.

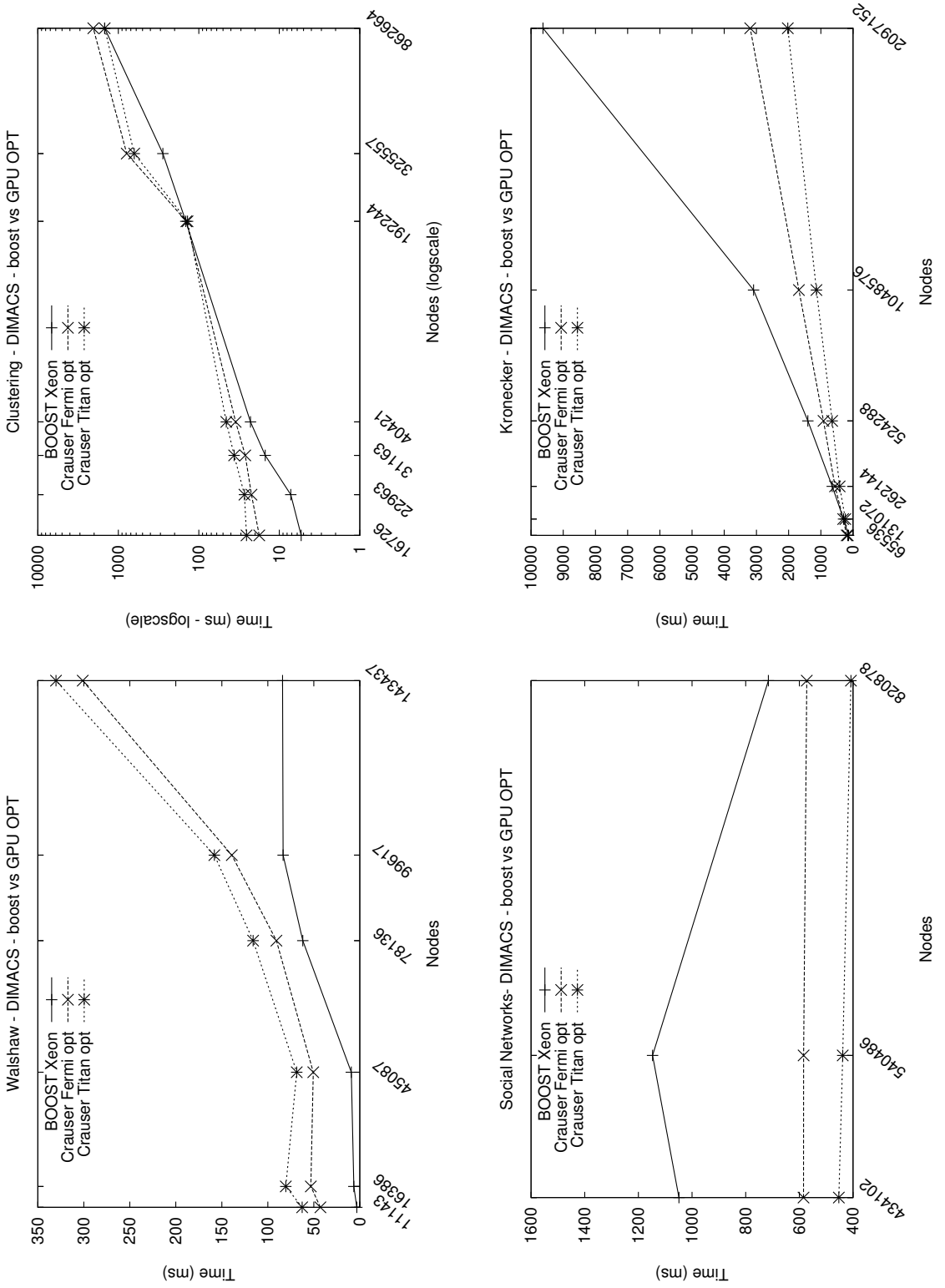


Figure 3.13: Experimental Results II: Execution times of the optimized GPU Crauser implementation, executed on Fermi and Titan boards, versus the optimized sequential Dijkstra's algorithm included in the Boost Graph Library, for real-world graphs.

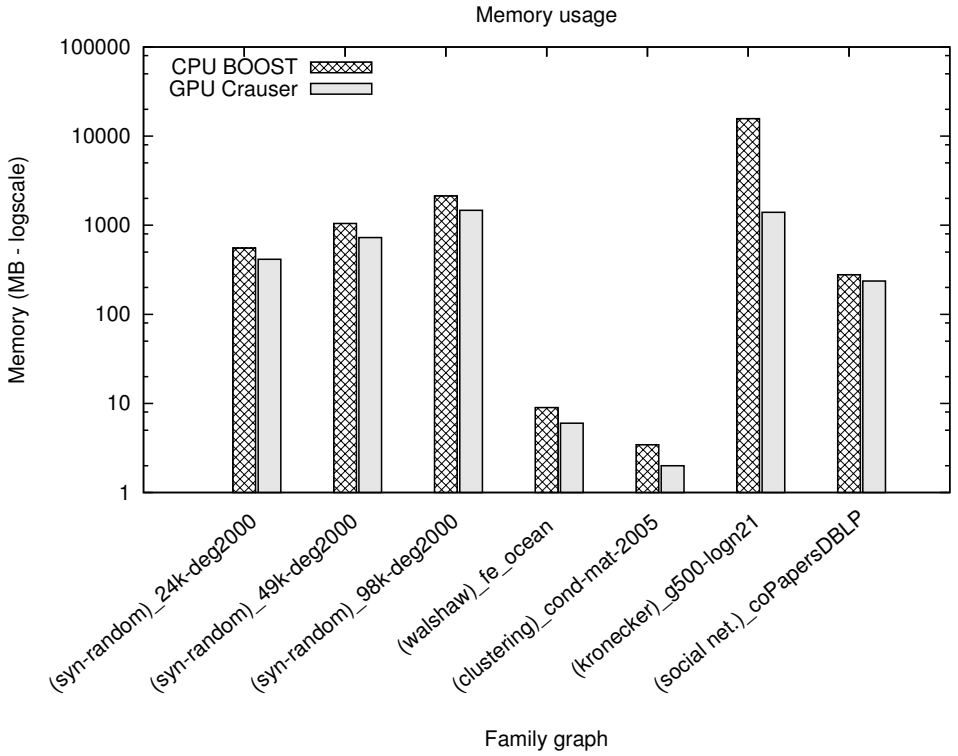


Figure 3.14: Memory usage of Boost library and GPU Crauser for certain graphs belonging to different graph families.

timized solution. We observe that for graphs with low fan-out degree and small size, there is a low level of parallelism associated. Thus, the sequential algorithms work better than the parallel GPU implementation (6.8× faster than Fermi). However, as the complexity of the graph increases, in terms of size and degree, also augmenting the level of parallelism, the difference between the execution times from the Boost library compared with the execution times from our GPU solution becomes bigger.

The greater the size of the graph, the earlier the performance of the GPU solution surpasses the sequential one: (1) deg200 for the 24k-size scenarios, with a speedup of 2.13×; (2) deg20 for the 49k-size scenarios with a speedup of 1.28×; and (3) deg10 for the 98k-size scenarios with a speedup of 1.24×. Finally, our GPU solution reaches a total speedup of 6.9×, 10×, and 19× for the synthetic random graphs with degree 2 000 and 24k, 49k, and 98k nodes, respectively, using the Titan board.

Figure 3.13 shows the same experimental scenario for real-world graphs, where similar conclusions can be obtained. The Boost library implementation performs well in the walshaw graph family, with speedups of up to 13.09× vs our optimized version. However, our GPU approach starts to get closer to it in the clustering family graphs, and finally offers a better performance than the Boost version in social-network and kronecker family graphs (with speedups of up to 4.76×), while requiring lower quantities of memory.

The memory usage of each approach, displayed in Fig. 3.14, is different due to the nature of each algorithm. The sequential implementation uses relaxed heaps as queue data structure to store the reached nodes. When there are many edges per node in a graph,

the space usually needed by this queue increases exponentially, making the computation impractical for cases with big sizes and high fan-out degree. In comparison, our GPU solution only has three additional vectors, in addition to the data structures to store the graph, and their size does not increase during the execution (see Sect. 3.1), leading to a consumption of up to 11.25 times less memory space (kronecker_g500-logn21 graph).

3.3.5 Experimental Results III - Architectural Comparison

Figure 3.15 shows, for the synthetic random graphs, the execution times of the optimized GPU Crauser executed in the different NVIDIA platforms used. We observe that the last released board, NVIDIA Titan, did not always have the best performance for all cases. For the scenario with the lowest size and degree (24k-deg2), the Fermi board obtained the best performance with a difference of up to 39.40%. However, as the degree of the graph increases, these performance distances get closer until they meet at degree 200 in our experimental study. Finally, for more dense graphs, the Titan board reaches the best performance with a gain of up to 40.53%, as compared with Fermi. This behavior also appears for the other random graphs, but the meeting point between both architectures decreases as the graph size increases. This occurs because the Fermi board has a lower number of cores (480) than the other tested boards (1 536 for Kepler and 2 880 for Titan), but a higher clock rate (1.40 Ghz against 1.05 and 0.98 Ghz). Thus, for graphs with a low degree, where the level of parallelism is lower, it is better to use a higher clock-rate GPU with fewer cores instead of a slower one with more processing units. On the other hand, having higher degrees, there are more threads performing useful relaxing operations. Therefore, for graphs with high degree and high size, it is better to use a GPU with many more cores to exploit higher levels of parallelism.

For the real-world and benchmarking graphs (see Fig. 3.16), the GPU boards have returned analogous results to those of the synthetic random graphs, where Fermi is the fastest one for low size and low degree graph instances, and as these features increase, Titan obtains better results.

3.4 Conclusions

In this chapter we have described how we have adapted the Crauser *et al.* SSSP algorithm to exploit GPU architectures. We have compared our GPU-based approach with its sequential version for CPUs, and with the most relevant GPU implementations presented in [23]. This new GPU version obtains significant speedups for all kinds of graphs compared with the previous approaches tested, up to 45× and 130× respectively. It also obtains important speedups for some of the tested graph families compared with the optimized sequential implementation of the Boost library [25].

We have observed that the algorithm due to Martín *et al.* is not as profitable in GPUs as Crauser's for graphs with a small number of nodes and a low fan-out degree, behaving even worse than the sequential CPU Crauser version. This occurs due to the small threshold for converting reached nodes into frontier nodes of their algorithm, and due to the low level of parallelism that can be extracted for these graphs. Our optimized GPU solution

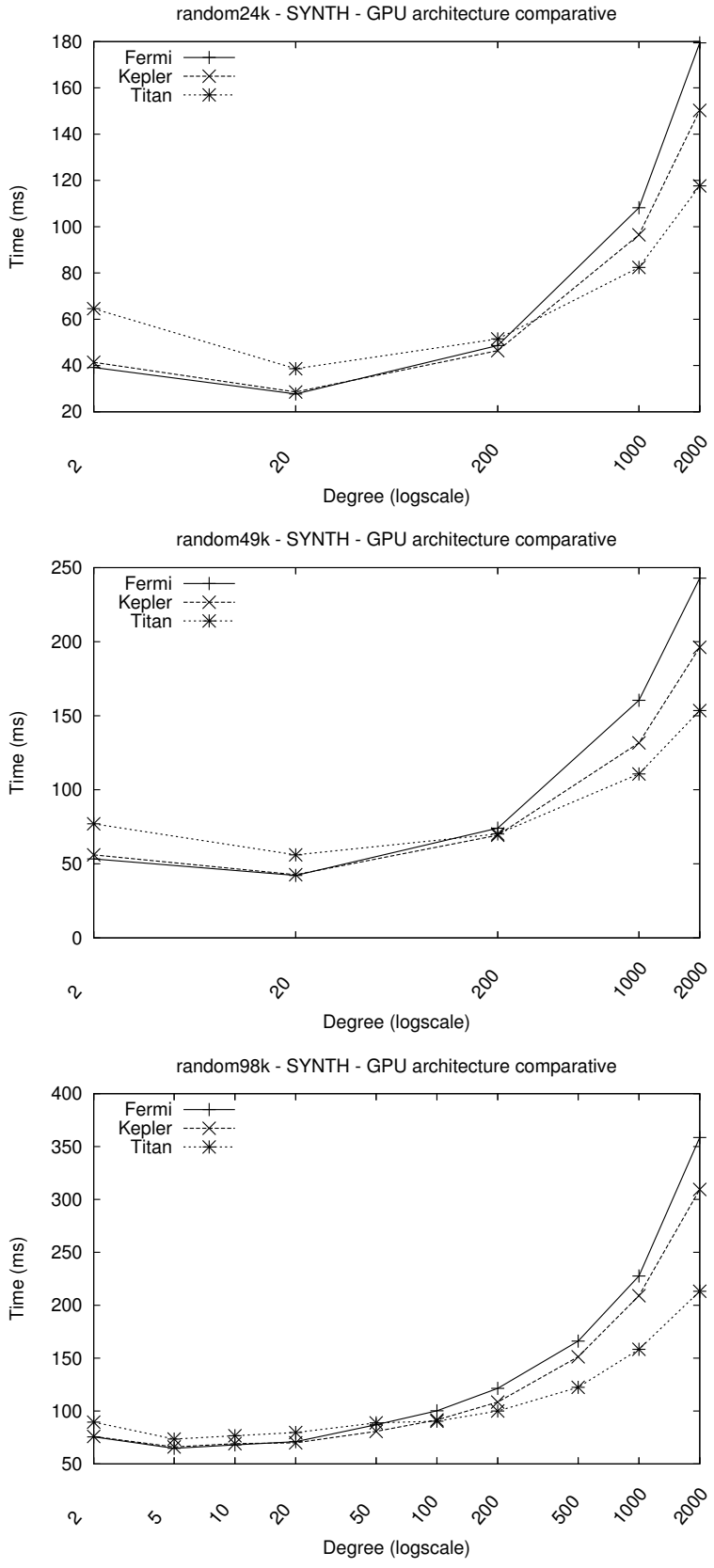


Figure 3.15: Experimental Results III: Comparison of CUDA architectures using the optimized GPU Crauser implementation for synthetic random graphs executed on the considered boards.

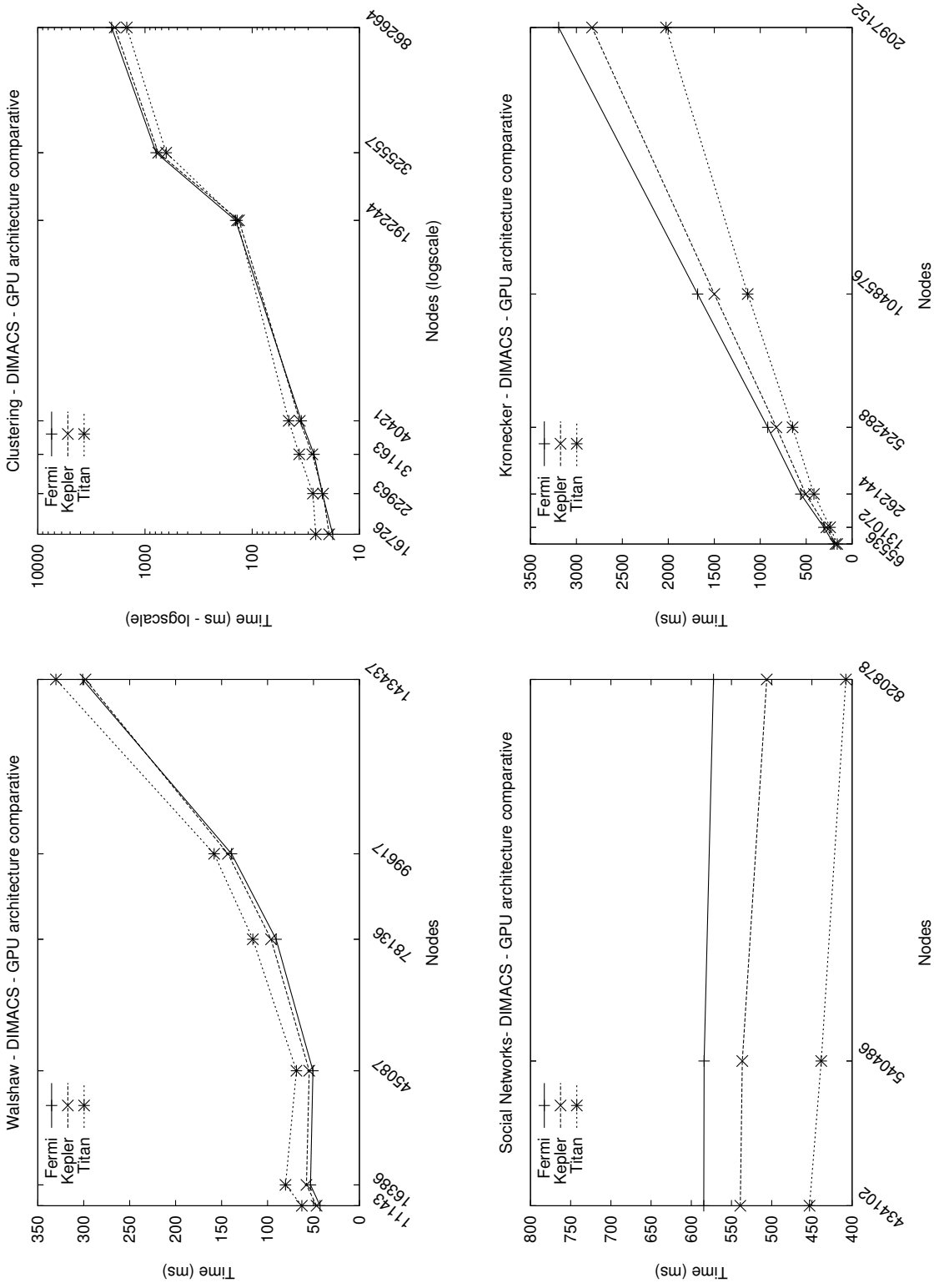


Figure 3.16: Experimental Results III: Comparison of CUDA architectures using the optimized GPU Crauser implementation for real-world graphs executed on the considered boards.

cannot beat the times of the Boost library in graphs with extremely low degrees for the same reason. However, our approach runs faster when the size and degree increases, obtaining a speedup of up to 19× for some graph families, consuming up to 11.25× less memory.

Most recent GPU architectures contain higher amounts of single processors at the cost of reducing clock frequency, in order to take advantage of the huge parallelism levels of the applications. However, as can be seen in our experimental results, there is a threshold, related to the low application parallelism level, where previous CUDA architectures, working with higher clock frequencies, obtain better execution times with a difference of up to 40.5%. Therefore, the faster architecture in this application domain is not always the most modern one, but depends on the features of the corresponding input set.

Finally, we have checked that is possible to obtain significant performance gains, up to 22.43% in our GPU implementation compared with the non-optimized version, by correctly tuning some CUDA configuration parameters. As we have briefly said in the experimental setup description, in Sect. 3.3.1, this optimized configuration can be obtained by trial-and-error or through a prediction process based on a kernel characterization criteria that will be described in the following chapter. This characterization process for the kernels of our GPU SSSP algorithm is also shown there, together with the exhaustive evaluation confirming the usefulness of the predicted values.

The work and conclusions described in this chapter have been published in the following papers:

- "Comprehensive Evaluation of a New GPU-based Approach to the Shortest Path Problem," H. Ortega-Arranz, Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos, *International Journal of Parallel Programming,* p. 1–21, 2015.
  Online, DOI: 10.1007/s10766-015-0351-z

- "A New GPU-based Approach to the Shortest Path Problem," H. Ortega-Arranz, Y. Torres, D. R. Llanos, and A. Gonzalez-Escribano, in *Proceedings of the 11th International Conference on High Performance Computing and Simulation*, ser.(HPCS'13), Helsinky, Finland: IEEE, 2013, pp. 505–511.
  Online, DOI: 10.1109/HPCSim.2013.6641461

Chapter 4

Exhaustive Search for GPU Optimal Parameter Values Within the Shortest-Path Context

Arriving to a parallel implementation of a highly-parallel algorithm is an affordable task. However, the optimization of GPU codes is a challenging activity. The main reason for this difficulty is the high number of parameters, programming choices, and tuning techniques available, many of them related with complex, and sometimes hidden, architecture details. A useful strategy to systematically attack these optimization problems is to characterize the different kernels of the application, and use this knowledge to select appropriate configuration parameters in a systematic way.

In this chapter we use different kernel characterization criteria to tune the execution of our proposed implementation for NVIDIA GPUs to solve the APSP problem. Our experimental results show that the combined use of proper configuration policies, and the concurrent-kernels capability of new CUDA architectures, leads to a performance improvement of up to 62% with respect to a basic configuration among the ones recommended by CUDA, considered as a baseline.

4.1 Problem Description: The Importance of Using Proper Values for Tuning Parameters

Nowadays, GPUs are among the most powerful HPC devices. However, their use with programming models such as CUDA implies to take several decisions in terms of runtime configuration parameters. Some of these CUDA specific configuration parameters, such as the threadblock size, the configuration of the L1-cache size, and the amount of concurrent kernels [21], do not need changes in an application code, turning out the optimization of the application into a simple tuning process. The joint use of the considered techniques can lead to significant performance improvements, but the problem is that there are many possible combinations.

Until now, the only way to ensure which are the best values for these configuration/op-

timization parameters is to carry out an empirical study that explores the complete search space. In order to avoid this costly task, CUDA gives some recommended values, such as the use of threadblock sizes that maximize the occupancy, as a proper choice for a good NVIDIA GPU performance [21]. However, the authors of [16] have shown through synthetic micro-benchmarking that this recommendation does not always correlate with the values that lead to the optimal performance. Instead, they proposed some rules to select the optimal values depending on the kernel features.

Since we have a GPU implementation that we want to optimize using these tuning techniques, we will follow the guidelines proposed by [16]. First we will refine and extend these guidelines to accommodate them for the shortest path context, taking into account some input graph features that modify the computing behavior of the GPU. Later we will characterize the kernels of our GPU implementation in order to obtain predicted values that are meant to be suitable for optimal/near-optimal executions.

In order to evaluate the validity of the new model, it is needed to check the correctness of our predictions. This test is carried out by executing an exhaustive search that evaluates the most relevant values for the configuration parameters.

Additionally, the usefulness of the model should be also evaluated, in terms of performance improvements, by comparing the execution times obtained with the predicted configurations against the execution times of a *baseline configuration*. This *baseline configuration* represents the configuration that a GPU programmer could apply following the recommendations of CUDA guidelines [21], leading to the poorest performance.

Since the second generation of NVIDIA architectures, CUDA supports concurrent kernel execution. With this feature different kernels can be executed concurrently, allowing better utilization of GPU resources. The maximum limit of concurrent kernels that the architectures GF110 and GK104, used for other experiments in this Ph.D. thesis work, support is up to 16 kernels. Note that, this limit can be lower depending on the usage of the underlying hardware resources made by the launched kernels. Moreover, the advantages of concurrent kernel execution are automatically exploited by the CUDA kernel dispatcher, providing a high efficiency without an extensive user intervention.

Using this feature, we aim to solve the APSP problem through the $n \times SSSP$ approach by concurrently executing several instances of the kernels involved in our GPU SSSP implementations (*relax*, *minimum*, and *update*), where the different instances of each one are assigned to different SSSP sources. Finally, the integrity of the model has to be tested when it is used together with a concurrent kernel execution. Along with this experimentation, we will try to obtain some guidelines for the concurrent kernel usage to fit into the previously mentioned extended model for parameter tuning.

4.2 State of the Art: Scarce Models for GPU Configuration Parameter Tuning

There exists multiple code tuning strategies for CUDA programming model [21]. These strategies are currently more complicated to understand and use than those designed for commercial CPUs. Therefore, to squeeze the computational power of a GPU requires much more programming effort. There is a lack of studies focused on the tuning of CUDA

configuration parameters. A correct choice of them is critical to take advantage of GPU capabilities, even when applying previous code optimizations techniques.

Regarding the threadblock size tuning, one of the first works that shows some inklings of squeezing the GPU using these techniques is the work of Wynters [152]. He tested several threadblock sizes in a matrix multiplication implementation using a GPU with 240 cores (pre-Fermi architecture). The comparison between the evaluated sizes shows that the correct choice for this configuration parameter is very important, due to the high difference in the execution times. However, the comparison is not completely fair since threadblocks with very few threads are used (1, 4, 16, 36, 64, 81, 144) against threadblocks of 256 threads. An example of a more recent work is [153], where the authors evaluated all threadblock sizes multiples of the warp size, 32 threads, for their application. However, this study is applied only to GPUs with Fermi's architecture, and their implementation only accepts cubic threadblock sizes ($n \times n \times n$), leaving out many possible combinations.

CUDA programming guidelines [13, 21] highlight the importance of using proper threadblock sizes. Their recommendation is to choose one that maximizes occupancy of the Streaming Multiprocessors of the target GPU, in order to reduce the memory latencies when accessing the global memory of the device. However, there are situations where other non-recommended values present significant better performance than these CUDA values [16]. Results of previous studies made by our research group [15, 154] also show the importance of adapting, not only the configuration parameters such as the threadblock sizes, but also the shared memory vs. the L1 cache memory sizes, according to the specific memory access pattern of the kernel to be tuned.

Regarding the management of the memory hierarchy, there are also few works that studies its importance in terms of performance gains, and they are only focused on Fermi's architecture. The authors of [155] show how the increment of the first cache memory of the hierarchy (L1-cache memory) significantly improves the total performance of an application, by taking advantage of data locality. The CUDA programming guidelines [13, 21] pose that experimentation is required to determine the best combination of L1 cache/shared memory for a given kernel. Some examples of works that experimentally compared their implementations with different managements of the L1 cache in order to find the best configuration are [156, 157].

All works previously mentioned do not systematically explore a wide range of the joint use of these configuration parameters, nor relate their use with the underlying hardware effects, nor give some guidelines to apply proper values in order to optimize other GPU applications, nor even consider more modern NVIDIA CUDA architectures. Only the work presented in [16] covers a complete study on which are the proper values to use according to some characteristics of the GPU kernels. However, this model does not consider some situations where these kernel characterizations could change depending on the application-dependent parameters, nor the joint effect of using the modern feature of launching several kernels concurrently.

The creation of such a model, that can predict parameter values that lead to optimal/near-optimal execution times, will be useful for the final GPU programmers, alleviating them from the burden of performing an empirical search of these proper values, as it was done in [158, 159]. Additional examples where these studies would have been useful can be found in tools such as FLAME [160], and MCUDA [161]. Despite the valuable work

carried out in them, the user still needs to manually determine by trial-and-error the best values of configuration parameters. Such a model can also be used to complement other frameworks that try to automatically tune the GPU kernels. We will cover this point in more detail in Sect. 6.2.

4.3 Kernel Characterization Model: Code-Dependent Parameters

In this section we describe the kernel characterization with respect to a classification criteria based on the work presented in [16]. This criteria is based on three kernel code features that can be obtained by inspection of the source code: *Memory access pattern*, *Computational load ratio*, and *Data sharing*. We have refined the criteria by determining new specific value ranges, and classification methods for each of these three features. These ranges have been determined by experimental measures for the platforms considered [162].

Memory Access Pattern (MAp)

It refers to how each thread accesses the global memory positions at a given moment. Three different kind of patterns are defined:

(a) *Full-coalesced:* Each warp of threads requests only one transaction segment (also known as cache line) at the same time. This means that every thread is requesting data in the same segment, and therefore, the number of memory requests is small. The memory requests are overlapped with the instruction computation or the memory-request latencies of the following warps. This overlapping is optimized when the SM occupancy is maximized [21].

(b) *Medium-coalesced:* Each warp requests between two and four transaction segments at the same time. This means that there are threads of the same warp that request data from different segments. Thus, the overlapping benefits of computation and global memory latencies depend on other kernel features described below.

(c) *Scatter:* Each warp requests more than four transaction segments. Thus, the number of memory requests significantly increases with respect to the full-coalesced pattern needing more warps with high computational load to compensate the memory latencies.

Computational Load Ratio (CLr)

It refers to the mean ratio of logic or arithmetic instructions per thread compared to the memory accesses of the same thread. Note that this metric is related to the *operational intensity* metric of [163], but our metric considers the number of memory transfers independently of their number of bytes, which differs depending on the L1 activation configuration (32 bytes when deactivated, and 128 otherwise).

We define three ranges for this ratio:

- *Low CLr*, ratio values between 0 and 10.
- *Medium CLr*, ratio values between 10 and 100.
- *High Clr*, ratio values over 100.

Note that during the time that one or more warps are computing, other warps blocked due to global memory request can finish their communication (global memory transaction) phase, hiding the memory latencies. Broadly described, the communication-computation overlapping is optimized using maximum-occupancy threadblock sizes for full-coalesced memory access patterns with a low CLr, medium-coalesced memory access patterns with a medium CLr, and scatter access patterns with a high CLr.

Data Sharing Across Blocks Ratio (DSr)

It refers to the ratio of data sharing compared to the number of memory accesses per thread. We name the limit values for this metric as:

- *High DSr*, when all threads of a block re-uses all values fetched by other threads;
- *Low DSr*, when there is no DS (each thread accesses to different data).
- We consider all situations between the limits described as *Medium DSr*.

When there is a high DSr in a kernel, the recommendation in [16] is to increase the number of threads per block in order to take profit of the data present in the cache memories. Moreover, they recommend to augment the L1 cache size in order to store more reused values for Fermi's GF100 architecture, or to increase the L1 local data bandwidth for Kepler's GK104.

4.4 Kernel Characterization Model: Graph-Dependent Parameters

For the problems related with the calculations of the shortest paths within a graph, the kernel characterization criteria not only depends on the kernel programming but also on the graph properties, such as the mean fan-out degree and the size of the graph.

Depending on these two graph parameters, the number of required hardware resources and the memory hierarchy bottlenecks vary for the programmed kernels. Thus, the recommendations for proper values of the configuration parameters depending on the kernel characterization could slightly vary.

Number of Vertices of the Graph

The behavior of the GPU changes when the number of launched threads overpasses the maximum limit of active threads that it can support. In this case we say that the GPU enters in a stressed situation. Let *stressing ratio*, $st(G, GPU)$, being G a graph, and GPU

the particular device used, be the ratio between the number of launched threads and the maximum number of active threads of the GPU device.

In our GPU Crauser solution, the number of launched threads is equal to n, that is the number of vertices of the input graph G. Note that, for the particular case of *minimum kernel*, the number of launched threads is equal to $n/2$, but it still depends on this graph feature. Thus, we can associate this graph property, n, with the previously defined ratio, $st(G, GPU)$, to determine the behavior of a particular GPU:

- *Low stressing ratio* those with $st(G, GPU) \in (0, 1.5]$;
- *Medium stressing ratio* as those with $st(G, GPU) \in (1.5, 3]$; and
- *High stressing ratio* as those with $st(G, GPU) > 3$.

Mean Fan-out Degree

The behavior of the particular *relax kernel* (see Alg. 6) also depends on the number of outgoing connections the nodes of the graph have. This is because every GPU thread assigned to a frontier node has to explore all its adjacent vertices checking their belonging to the corresponding unsettled set U_i, and checking if the new obtained distance is lower than the previously stored value to keep the minimum one. Therefore, a high mean fan-out degree implies more memory accesses for each thread of the *relax kernel*. The augmentation of the size of the L1-cache can alleviate the burden of these overloading accesses. Note that the behavior of the remaining kernels, *minimum* and *udpate*, does not directly depend on this graph feature, because the GPU threads do not check any value regarding the adjacent successor vertices of their assigned nodes.

Let $d(G)$ be the mean fan-out degree of the input graph $G(V, E)$, defined as the mean number of outgoing edges for the graph nodes. We classify the input graphs according to the values of the mean fan-out degree as:

- *Low $d(G)$ graphs* those graphs with $d(G) \in [1, 20)$;
- *Medium $d(G)$ graphs* those graphs with $d(G) \in [20, 200]$; and
- *High $d(G)$ graphs* those graphs with $d(G) > 200$.

This parameter may affect the DSr code-dependent parameter for the same kernel. Thus, the classification of the specific kernel features should be taken into account.

4.5 Characterizing the Kernels of the SSSP Algorithm

A trial-and-error optimization of the GPU performance would imply a very large experimentation space due to the big number of possible combinations. Through the kernel characterization model described above and the guidelines proposed in [16], we can classify the GPU kernels of a program, and therefore predict which configuration values would lead to a good performance for our GPU implementation. Table 4.1 summarizes

the *relax*, *minimum* and *update* kernel properties using the classification criteria described above.

***relax kernel*:** The code of this kernel is shown in Alg. 6.

- ***(MAp)*** The first condition (line 2) performs coalesced memory accesses, whereas the inner instructions carry out low/medium coalesced accesses. Therefore, we consider that it has a medium-coalesced pattern.
- ***(CLr)*** The computational load ratio is low because there are less than 10 logic/arithmetic instructions, and several global memory accesses.
- ***(DSr)*** Finally, the data sharing across blocks will increase as $d(G)$ and n increase.

minimum kernel: The code of this kernel is shown in Alg. 9.

- ***(MAp)*** It has a full-coalesced pattern because contiguous threads access to contiguous memory addresses.
- ***(CLr)*** There are more than 20 logic/arithmetic operations considering the $\min$ operations and the for loop against a pair of values to retrieve from global memory, so it has a medium CLr.
- ***(DSr)*** The data sharing across blocks is not affected by $d(G)$ nor n, and hardly any data sharing is performed, so it has a low DSr.

update kernel: The code of this kernel is shown in Alg. 10.

- ***(MAp)*** All data structures are accessed with a full-coalesced pattern.
- ***(CLr)*** There is just one instruction per memory access (low CLr).
- ***(DSr)*** There is no data sharing present in this kernel (low DSr).

4.5.1 Predictions for the Threadblock-size Values

With the previous characterization of the GPU kernels, we can then predict which would be the proper values for the thread-block size in terms of the input sets characteristics $st(G, GPU)$ and $d(G)$, in order to obtain a optimal/near-optimal execution time for our application (see Table 4.2):

- ***relax kernel:*** A medium-coalesced access pattern with a low CLr suggests the use of the minimum value of the threadblock-sizes that maximize the occupancy. Depending on the st parameter the kernel data sharing across blocks varies. As this reutilization increases, the optimal threadblock size is higher:
  - *low st:* The lowest value that maximizes the occupancy (192 for Fermi, and 128 for Kepler);

Kernel	**d(G)**	**MAp**	**CLr**	**DSr**
Relax	Low	medium-coalesced	low	low
	Medium	medium-coalesced	low	medium
	High	medium-coalesced	low	high
Minimum	Low/Med./High	full-coalesced	medium	low
Update	Low/Med./High	full-coalesced	low	low

Table 4.1: The characterization of the *relax*, *minimum* and *update* kernels depending on the mean fan-out degree, d(G). The characterization parameters are the Memory access pattern (*MAp*), Computational load ratio (*CLr*), and Data sharing across blocks (*DSr*).

Kernel	**st**	**Threadblock-size**		**L1-Cache management**	
		FERMI	KEPLER	FERMI	KEPLER
relax	Low	192	128	Increased	Increased
	Medium	192 / 256	128 / 256	Increased	Increased
	High	192 / 256/ 384	256	Increased	Increased
minimum	Low	128 / 192	96 / 128	Increased	Increased
	Medium	128 / 192	96 / 128	Increased	Increased
	High	128 / 192	96 / 128	Disconnected	Disconnected
update	Low	192	128	Increased	Increased
	Medium	192	128	Increased	Increased
	High	192	128	Disconnected	Disconnected

Table 4.2: Prediction values resulting from the kernel characterization process.

  - *medium st:* The two lowest values that maximize the occupancy (192/256 for Fermi, and 128/256 for Kepler);
  - *high st:* Even higher values than the ones predicted for medium *stressing ratio* due to high reutilization (192/256/384 for Fermi, and 256 for Kepler).

- ***minimum kernel:*** A full-coalesced access pattern with a medium CLr suggests the use of the minimum value of the threadblock sizes that maximize the occupancy, or even slightly lower values, although they do not fully maximize the occupancy (128/192 for Fermi, and 96/128 for Kepler). The behavior of this kernel is not affected by the st parameter.

- ***update kernel:*** A full-coalesced pattern with a low CLr leads to the use of the minimum value of the threadblock sizes that maximize the occupancy (192 for Fermi, and 128 for Kepler). The behavior of this kernel is not affected by the st parameter.

4.5.2 Predictions for L1-cache Management

The characterization of the kernels also allows us to predict which is the proper state that should be chosen for the L1-cache memory (deactivated, normal state, increased) to obtain a optimal/near-optimal execution time of our application.

In the SSSP kernels, the shared memory is not totally filled up, so the increase on the size of the L1-cache memory does not lead to performance degradations, and this augmentation alleviates the memory thrashing effects. However, when there is too much thrashing in the L1 cache, as it happens when the GPU enters in stressed situations, the performance can be improved by deactivating this memory. This occurs because the deactivation of the L1 cache memory implies the reduction of the size of the transaction segments (from 128 to 32 bytes). This size reduction accelerates the transmission of the requested segments. The number of requested transaction segments from global memory is highly increased for graphs that cause a high-stressing situation to the GPU.

Therefore, the predictions for all SSSP kernels are:

- to increase the L1-cache memory for graphs with a low/medium $st(G, GPU)$; and
- to deactivate the L1-cache memory for graphs with a high $st(G, GPU)$.

For the particular case of the *relax kernel*, that has a high DSr (data sharing across blocks), the recommendation is to always increase the size of the L1-cache memory, even when it is executed on high-stressing situations. A high ratio of data sharing across blocks reduces the theoretical thrashing effect of a high-stressing situation.

4.6 Experimental Evaluation

In this section, we describe the methodology used to design and carry out the experiments to validate the kernel characterization model. We also study its usefulness through the different experimental scenarios we considered, the input sets used, and the experimental results.

4.6.1 Methodology

We have performed a complete experiment that allows us to carry out three different studies of the results, checking different premises. This experiment tested a wide range of combinations of the possible configuration values, for the threadblock size, L1-cache memory management, and number of concurrent kernel, across the full search space. Afterwards, each study analyzes the experimental results focusing on different objectives:

- The first study evaluates the influence of launching more kernels concurrently in the predictions, and also tries to find some guidelines to add to the model.
- The second study checks the validity of the model by comparing the values that returned the best execution times with the ones predicted in Sect. 4.5.
- The third study measures the usefulness of these predictions in terms of performance gain against a configuration that follows the recommendations of CUDA programming guidelines.

We issued these experimental studies for the APSP problem, using the $n \times SSSP$ approach, measuring the execution time of the whole program and of each kernel separately. The measures were repeated with different input sets, and for two different architectures, Fermi GF100 and Kepler GK104.

A complete description of the platforms and devices used, and the experiments carried out, is presented below.

Target architectures

The experiments have been carried out using the CUDA Toolkit 4.2, and the GPU devices GTX 480 (Fermi GF100) and GTX 680 (Kepler GK104) with 23 040 and 16 384 maximum concurrent threads respectively. The host machine used is an Intel(R) Core(TM) i7 CPU 960 3.20GHz, with 6 GB of memory with an Ubuntu Desktop 10.10 (64 bits).

Experiment: Exhaustive Evaluation of CUDA Runtime Configuration Parameters

This experiment is carried out by executing an exhaustive search in the workspace that evaluates the most relevant values for the CUDA configuration parameters. The following paragraphs enumerates the considered values in our experimental searching for the optimal execution times (see Table 4.3).

- **Threadblock size:** CUDA recommends the use of threadblock sizes that maximize the SM occupancy of the GPU [21]. These sizes are dependent on the GPU architecture, being 192, 256, 384, 512, and 768 for Fermi, and 128, 256, 512, and 1 024 for Kepler. Nevertheless, following the recommendations of [16], we have also evaluated lower values (occupancy ≥ 0.75). Note that all values should be multiples of 32 (warp size) in order to maximize the core exploitation of the SM. Consequently, the block sizes that lead to a medium ratio of SM occupancy have been also tested (96 and 128 for Fermi, and 96 for Kepler). The grid and blocks use one-row shapes

Configuration parameter	Used values	
	FERMI GF100	**KEPLER** GK104
Threadblock size	96 / 128 / 192 / 256 / 384 / 512 / 768 / 1024	96 / 128 / 256 / 512 / 1024
L1-cache management	Deactivated / Normal / Increased	
#concurrent kernels	1 / 2 / 4 / 8 / 16 / 32	

Table 4.3: Tested values in our experimental scenario for the different configuration parameters.

to couple the thread indexes with the elements of the unidimensional array where the graph is stored.

- **Cache L1 state** We have included in our experimentation the three different states that this cache can adopt for the GPU boards considered: Normal state (16K of cache L1), Augmented state (48K of cache L1), and No-L1 state (cache L1 deactivated).

- **Number of concurrent kernels** We have evaluated the following number of concurrent kernels: 1, 2, 4, 8, 16, and 32. The concurrent kernel technique allows to send several instances of the same kernel simultaneously with a synchronized end. Each of these instances operates in a different memory workspace to solve its corresponding SSSP task. Due to the concurrent kernel synchronization, these memory workspaces are transferred with a single operation.

Experimental Study I: Compatibility with Concurrent Kernels

This study is focused on the influence of executing more than one kernel concurrently in a GPU with the model predictions of threadblock / L1-cache memory sizes. The analysis of the results will determine if the recommendations given by our model can be maintained when applying concurrent kernel execution or not. Finally, if this parameter is independent, the results of this study can be used to create new guidelines to extend the kernel characterization model.

Experimental Study II: Validation of Model Predictions

Assuming that the concurrent-kernel execution does not affect the predicted values, the second study evaluates the validity of the model. This study check the correctness of the model predictions by comparing them with the values that delivered the optimal execution times.

Experimental Study III: Usefulness of Model Predictions

The objective of this study is to measure which is the maximum performance gap between an optimized configuration, applying the kernel characterization model, and a naïve not

so good configuration, that could be obtained by any programmer that follows the CUDA guidelines recommendations [21].

Once we have obtained the best configuration combining the three techniques in the experiment, we compare it with this naïve configuration, that we have called the *baseline configuration*. This ***baseline configuration*** applies the following values:

1. From the threadblock sizes recommended by the CUDA guidelines, that usually offer a good performance, it uses the one that returns the lowest performance;

2. No cache adjustments, default configuration (normal state); and

3. No kernel concurrency (one kernel at a time).

4.6.2 Input set characteristics

The used input set is composed by the collection of synthetic random graphs described in Sect. 3.3.2. In order to reduce the huge amount of results due to the high number of values to combine, we have chosen only the graphs with a mean fan-out degrees $d(G) \in \{2, 20, 200\}$ from the synthetic random suite. The sizes $n \in \{24\,576, 49\,152, 98\,304\}$ of these graphs have been designed to have scenarios with the stressing ratio in the three levels previously defined (24k–low stressing ratio, 49k–medium stressing ratio, 98k–high stressing ratio).

4.6.3 Experimental Results: Exhaustive Evaluation of CUDA Runtime Configuration Parameters

Due to the high number of results returned from combining all possible configuration values, the complete set of figures with all the results of the remaining experimental scenarios are presented in the Appendix A. These figures are the result of formatting the raw data resulted from this experimentation in order to allow an easy comparison between different dimensions at once.

As an example, Figs. 4.1 and 4.2 show the execution times for all the possible value combinations of the configuration parameters, for the 24k-d2 and the 98k-d200 used graphs for the Fermi GF100 architecture. These graphics are organized as follows. A graphic is depicted for each kernel executed using a particular graph set (with a specific number of nodes and fan-out degree). Inside this graphic we find that: The results are grouped in three coupled frames depending on the configuration of the L1 cache memory; The running times obtained using a different number of concurrent kernels executed are depicted with a curve colored depending on the particular threadblock size used. A horizontal line is depicted for the minimum running time inside each frame, with the same color than the one used to depict its corresponding threadblock curve. Each line crosses all three frames in order to ease the comparison of the minimums, being the lowest line of the complete graphic the one that represents the configuration with the fastest running times, for that kernel using this particular graph set.

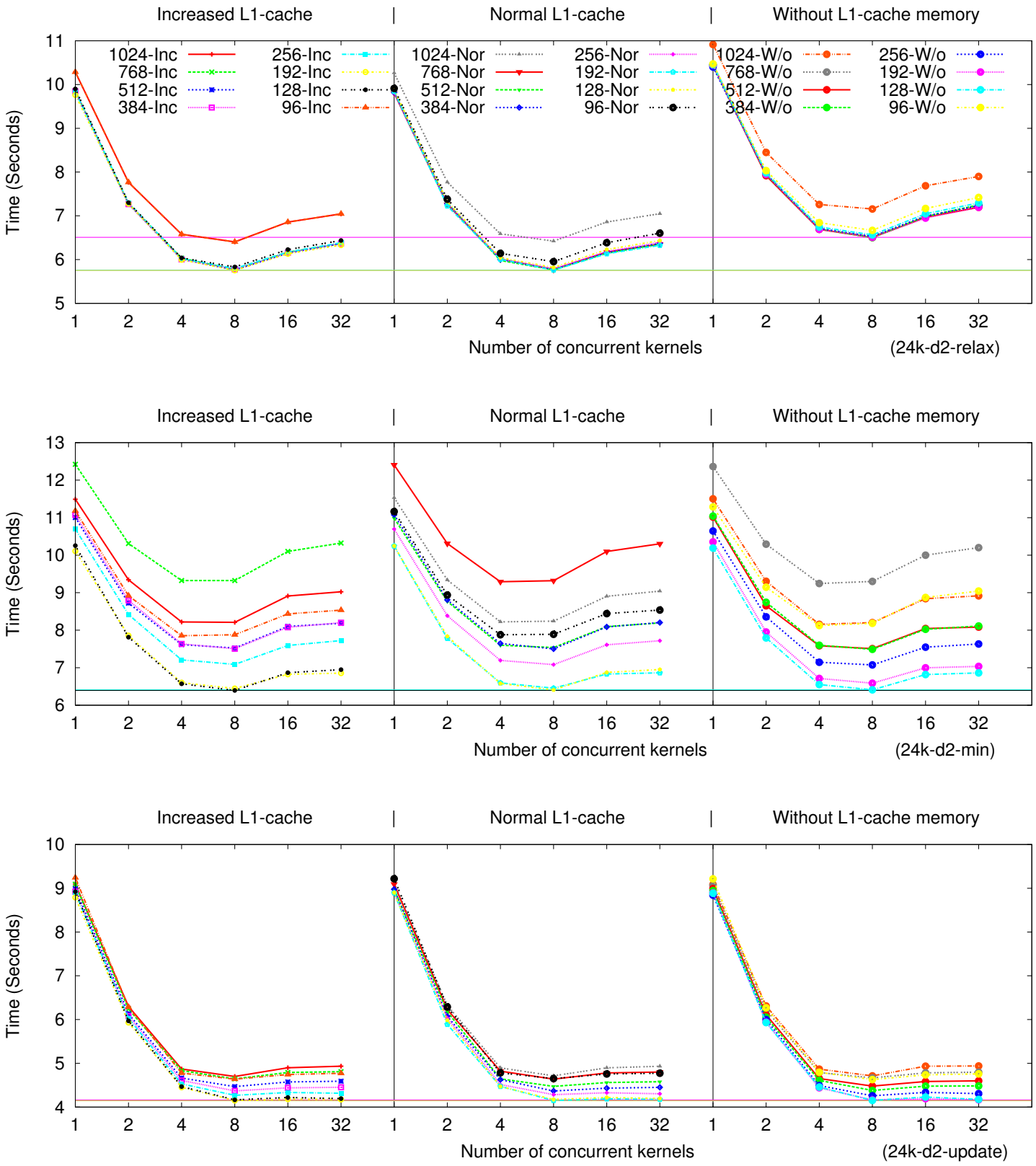


Figure 4.1: Exhaustive search for optimal values in the graph 24k-d2 scenario for the Fermi GF100 architecture. Different threadblock sizes, states of the L1-cache, and number of concurrent kernels have been evaluated for the *relax kernel (top)*, *minimum kernel (middle)*, and *update kernel (bottom)*. Each different key depicted inside the top frame of each state of the L1 cache memory is the same for the remaining frames with equal L1 cache memory configuration.

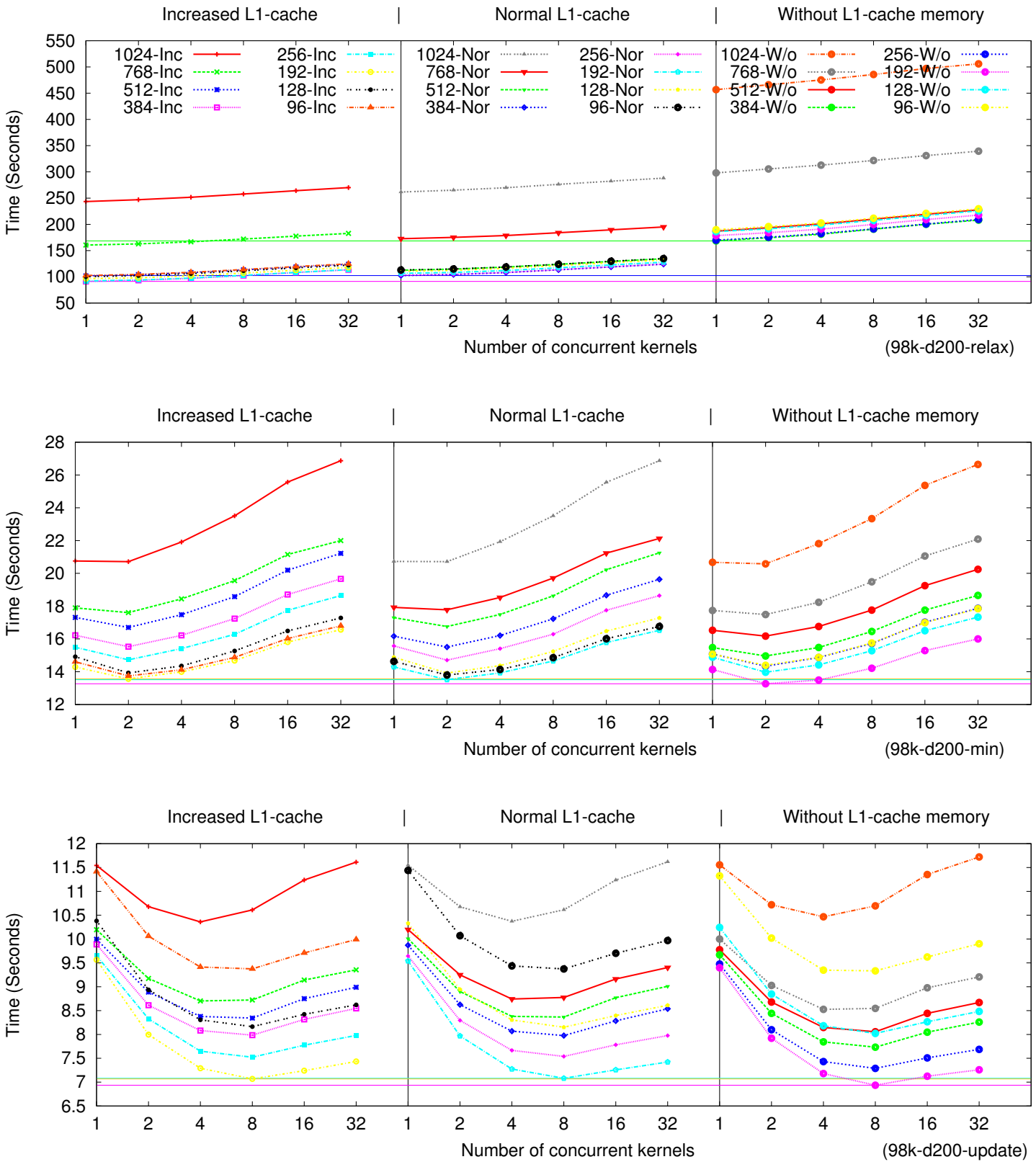


Figure 4.2: Exhaustive search for optimal values in the graph 98k-d200 scenario for the Fermi GF100 architecture. Different threadblock sizes, states of the L1-cache, and number of concurrent kernels have been evaluated for the *relax kernel (top)*, *minimum kernel (middle)*, and *update kernel (bottom)*. Each different key depicted inside the top frame of each state of the L1 cache memory is the same for the remaining frames with equal L1 cache memory configuration.

4.6.4 Study I - Compatibility with Concurrent Kernel Execution

The results suggests that the use of the concurrent-kernel execution with the other two techniques does not interfere with the predictions made. In all cases the best configuration values using only one concurrent kernel matches the same value using more concurrent kernels.

Using this technique, a performance improvement in all scenarios is obtained; up to 52.8% for Fermi, and up to 44.3% for Kepler (both cases for the *update* kernel in 24k-d2), with the exception of the d200 cases for the *relax* kernel. In there, it was better to leave a sequential-kernel execution instead launching concurrent kernels. This occurs because the number of memory accesses significantly increases when each thread is looking for around 200 successors of its corresponding frontier node (see line 3 of Alg. 6). The minimum kernel has also to carry out more memory accesses when calculating the minimum value for graphs with bigger $d(G)$ because there are more reached nodes with tentative-distance values to compute in each iteration. This effect can be seen in the d200 scenarios, where the optimal concurrent-kernel number is lower compared to the other cases. Otherwise, for the *update* kernel, the increment of $d(G)$ of each graph does not lead to more memory accesses, so the optimal number of concurrent kernels is the same for all cases.

Following these results we can include into the programmer guidelines the recommendation of reducing the number of concurrent-kernel deployment for kernels with big number of memory accesses.

4.6.5 Study II - Validation of Model Predictions

Table 4.4 shows the best and the baseline values of CUDA parameters for the APSP kernels. The first columns contain, the $st(G)$, and the graph properties, n and $d(G)$. The remaining ones show the best values found for the studied parameters. These optimal values match almost all predicted values obtained following the optimization guidelines of the kernel characterization model (see Sect. 4.5) for both architectures, Fermi and Kepler. For the unmatched predicted values, the differences between the execution times delivered by them and the optimal performance are lower than the 1%. Therefore, we can also consider them as proper values for optimal performances.

Summary of the applied optimization guidelines

For those kernels of any application that fulfill the same characterizations presented in Table 4.1, we expect that they will present the same performance behavior when using the same configuration values. Thus, we can extrapolate the following optimization guidelines for selecting threadblock size and L1-cache state management:

(1) *Use the minimum size that maximizes the occupancy for kernels with a:*

- i Full-coalesced memory access pattern, a low CLr, and a low DSr (see all scenarios of *update* kernel in Table 4.4 (c)).
- ii Medium-coalesced memory access pattern, a low CLr, and a low DSr (see 24k-scenarios of *relax* kernel in Table 4.4 (a)).

iii Full-coalesced memory access pattern, a medium CLr, a low DSr, and high stressing ratio (see 98k-scenarios of *minimum* kernel in Table 4.4 (b)).

(2) *Use lower sizes than the minimum one that maximize the occupancy for:*

- Full-coalesced memory access patterns with a medium CLr, a low DS, and a low/medium st kernels (see 24k and 49k scenarios of *minimum* kernel in Table 4.4 (b)).

(3) *Jump to higher maximizing occupancy sizes from the minimum one for:*

- Medium-coalesced memory access patterns with a low CLr, and a medium/high DS kernels (see 49k and 98k-scenarios of *relax* kernel in Table 4.4 (a)).

(4) *Increase the L1 cache size for non-high stressing situations for the GPU, or high data sharing situations:*

- As predicted, the use of a bigger cache L1 size speeds up the execution time compared with the normal size configuration, because thrashing effects are alleviated, and compared with a deactivated cache configuration, because the global memory requests are slower than cache ones. That is the case of 24k-scenarios (*low* $st(G)$) and most of the 49k-scenarios (*medium* $st(G)$), and the 98k-scenarios of *relax* kernel (*high DSr*).

(5) *Deactivate the L1 cache size when the GPU enters in a high-stressing state with non-high data sharing.*

- In the 98k-scenarios (*high* $st(G)$) with non-high DS, the number of memory accesses is increased, due to the thrashing effect. As it was predicted, in order to alleviate the memory traffic, it is better to disconnect the L1 cache, due to the reduction of the transaction segment size.

Similar performance behavior for Fermi and Kepler

Although each architecture has its own values that maximize the occupancy, both architectures have presented a similar behavior in our experimentation. We can observe from Table 4.4, that the optimal values for all kernels deployed on both architectures follow the guidelines described in Sect. 4.5. Thus, we conclude that the lessons learned, and described above, can be applied to any Fermi and Kepler board. The experimental results described in Sect. 3.3.3 showed that the predicted values used here to achieve an optimal performance for the Kepler GK104 architecture delivered also the same performance trends for a GPU with a Kepler GK110B architecture.

4.6.6 Study III - Usefulness of Model Predictions

Figure 4.3 shows the execution time breakdown of the different scenarios chosen for Fermi architecture in terms of the times consumed by each kernel, and other operations, such as memory transfers. For each scenario, we present two bars, one for the baseline time, and

st	Graph	Fermi GF100	Kepler GK104
Low	24k-d2	192-8-Increased→ 41.4%	128-4-Increased→ 35.3%
	24k-d20	192-4-Increased→ 16.5%	128-4-Increased→ 28.8%
	24k-d200	192-1-Increased→ 12.0%	128-1-Increased→ 30.8%
Medium	49k-d2	192-4-Increased→ 35.4%	256-4-Increased→ 34.3%
	49k-d20	192/256-2-Incr.→ 28.2%	256-2-Increased→ 38.3%
	49k-d200	256-1-Increased→ 34.0%	256-1-Increased→ 49.4%
High	98k-d2	192-8-Increased→ 39.7%	256-8-Increased→ 40.9%
	98k-d20	256-2-Increased→ 34.8%	256-2-Increased→ 48.9%
	98k-d200	384-1-Increased→ 47.2%	256-1-Increased→ 62.4%
Baseline configuration		768-1-Normal	1024-1-Normal

(a) Relax kernel

st	Graph	Fermi GF100	Kepler GK104
Low	24k-d2	128-8-Increased→ 48.5%	96-8-Increased→ 43.8%
	24k-d20	128-4-Increased→ 41.6%	96-8-Increased→ 38.1%
	24k-d200	128-4-Increased→ 41.1%	96-8-Increased→ 35.5%
Medium	49k-d2	128-4-Increased→ 34.7%	96-8-Increased→ 40.5%
	49k-d20	128-2-Without→ 30.0%	96-4-Increased→ 34.1%
	49k-d200	128-2-Without→ 30.5%	96-4-Increased→ 34.7%
High	98k-d2	192-4-Without→ 30.0%	128-4-Increased→ 36.8%
	98k-d20	192-2-Without→ 25.4%	128-2-Without→ 27.1%
	98k-d200	192-2-Without→ 26.0%	128-2-Without→ 28.1%
Baseline configuration		768-1-Normal	1024-1-Normal

(b) Minimum kernel

st	Graph	Fermi GF100	Kepler GK104
Low	24k-d2	192-8-Increased→ 54.4%	128-8-Increased→ 45.3%
	24k-d20	192-8-Increased→ 45.5%	128-8-Increased→ 36.8%
	24k-d200	192-8-Increased→ 45.0%	128-8-Increased→ 34.5%
Medium	49k-d2	192-8-Increased→ 44.6%	128-8-Increased→ 37.4%
	49k-d20	192-8-Without→ 37.4%	128-8-Without→ 28.6%
	49k-d200	192-8-Without→ 38.1%	128-8-Without→ 29.6%
High	98k-d2	192-8-Without→ 41.0%	128-8-Without→ 40.2%
	98k-d20	192-8-Without→ 29.2%	128-8-Without→ 27.3%
	98k-d200	192-8-Without→ 32.0%	128-8-Without→ 30.4%
Baseline configuration		512/768-1-Normal	1024-1-Normal

(c) Update kernel

Table 4.4: The best values of configuration parameters (threadblock size, number of concurrent kernels, and L1 cache state) obtained experimentally for the *relax* (a), *minimum* (b) and *update kernels* (c) and their percentage of performance gain with respect to the baseline configurations.

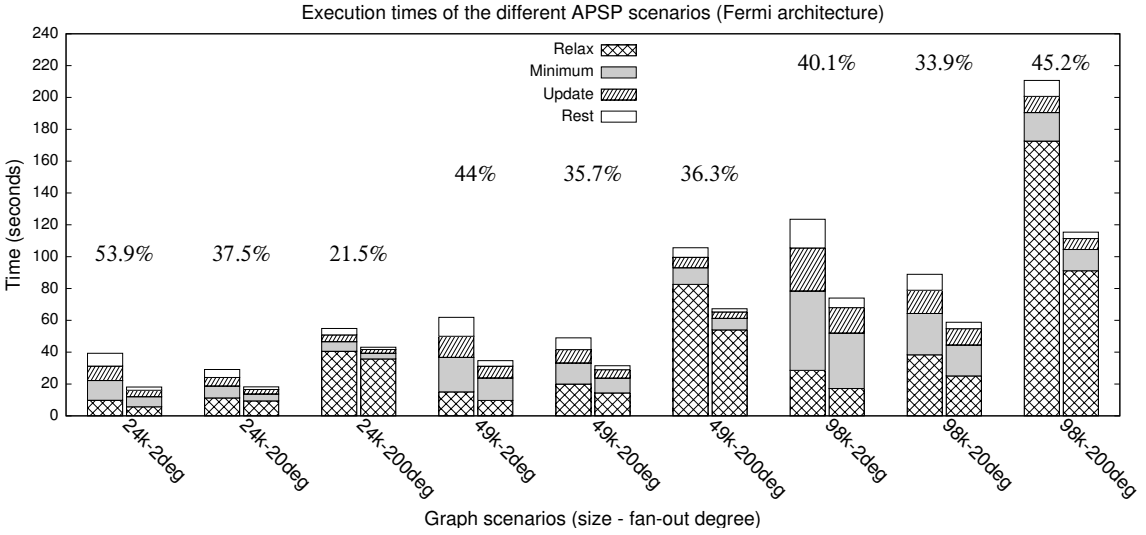


Figure 4.3: Execution time breakdown of the GPU kernels as well as other operations (data transfers) on the different scenarios, and the performance gain percentages for Fermi architecture between the baseline (left column) and the best configurations (right column).

one for the best execution time experimentally obtained for any tested configuration value combination. The performance gain percentages are written above the bars. The execution breakdown observed in Kepler is similar. For Fermi, the use of the considered techniques turns out in a global performance gain from 21.5% to 53.9%, whereas for Kepler they are in the range from 33.75% to 58.53%.

The joint use of these optimization techniques not only returns performance improvements in the total execution time of the GPU implementation, but also it always improves the execution time of each kernel independently (see non-white boxes in Fig. 4.3). Comparing the best values of the configuration parameters with the results for baseline values (see percentages of Table 4.4), we obtain the following performance gains for Fermi:

1. ***relax kernel*** from 12% to 47.2%,
2. ***minimum kernel*** from 25.4% to 48.5%, and
3. ***update kernel*** from 29.2% to 54.4%.

For Kepler, the performance gains are:

1. ***relax kernel*** from 28.8% to 62.4%,
2. ***minimum kernel*** from 27.1% to 43.8%, and
3. ***update kernel*** from 27.3% to 45.3%.

Additionally, the use of the concurrent-kernel technique has significantly reduced the memory transfer times between CPU and GPU (see the white boxes in Fig. 4.3). This transfer time is reduced because of the fact that it is more efficient to transfer bigger data sets than a lot of smaller data sets. Due to the data offloading-transfer management chosen for our implementation, the higher the number of concurrent kernels, the bigger the amount of data in a single transfer, and the less transfer iterations needed.

Unstable evolution for default values in new architectures

The default layouts of the new boards and architectures is changing. As we previously described in Sect. 4.6.1, the default state of the L1 cache memory configures it with a total of 16 KB, we named normal state. Nevertheless, with the advent of new architectures, such as the Kepler GK110B, or Maxwell GM204, NVIDIA changed the default configuration from this normal state to the deactivated state [164, 165]. Although these new releases maintain the possibility of handling all possible states of previous architectures (increased, normal, and deactivated) with equivalent sizes, now the programmers should compile their programs with a specific flag to configure the GPU kernel execution with the increased and normal states. Thus, if we change what we consider the *baseline configuration*, described in Sect. 4.6.1, the performance differences against the optimized version are even more significant for some of the kernels considered, as the three-fold speedup of *relax kernel* in the 98k-d200 scenario shown in Fig. 4.2.

4.7 Conclusions

This chapter have shown how the combined use of different configuration and optimization techniques can significantly enhance the kernel performance of a GPU solution. When applied to our problem case study, the APSP problem, we obtained a global performance improvement up to 62% compared with baseline configurations.

The experimental results have shown how the kernel characterization technique is a useful procedure to predict proper configuration parameter values for the threadblock size and the cache L1 state, leading to significant performance improvement in NVIDIA GPUs. The experimental results described in Sect. 3.3.3 showed that the predicted values used here to achieve an optimal performance for the Kepler GK104 architecture delivered also the same performance trends for a GPU with a Kepler GK110B architecture. Thus, we can conclude that the predicted values obtained from the kernel characterization procedure can be applied to any Fermi and Kepler board.

We have shown that the CUDA recommended values are not always the proper choice, and due to the big search space of possible combinations, we find these predictions and guidelines very helpful for non-expert CUDA programmers, or even for auto-tuning tools that aim to automatically configure the kernel execution for an optimal performance. Finally, the non-stable default configuration for the new boards makes the tuning and portability processes more difficult, turning the usage of the predictions and guidelines in helpful "compulsory" practices if optimal application performance are desired.

Regarding to the concurrent-kernel technique, the experimental results suggest that its use does not interfere with the predictions of the kernel characterization criteria. This technique results more profitable for kernels with few memory accesses (high ratio of operations per memory accesses). Additionally, for the case of the offloading-data transfer management used in our APSP implementation, it does not only reduce the total execution time for a pack of kernels, but also promotes design changes that decrease the transfer times by copying to the GPU a big block of data only once, instead of transferring small

data blocks multiple times.

The work and conclusions described in this chapter have been published in the following papers:

- "Optimizing an APSP Implementation for NVIDIA GPUs Using Kernel Characterization Criteria", H. Ortega-Arranz, Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos, *The Journal of Supercomputing,* vol. 70, no. 2, pp. 786-798, 2014.
  Online, DOI: 10.1007/s11227-014-1212-z

- "A Tuned, Concurrent-Kernel Approach to Speed Up the APSP Problem," H. Ortega-Arranz, Y. Torres, D. R. Llanos, and A. Gonzalez-Escribano, in *Proceedings of the 13th International Conference on Computational and Mathematical Methods in Science and Engineering*, ser.(CMMSE'13), Almería, Spain: eds. I.P. Hamilton and J. Vigo-Aguilar, 2013, vol. 4, pp. 1114-1125.
  Online: CMMSE Proceedings

Chapter 5

Using Heterogeneous Computing to Solve the All-Pair Shortest-Path

This chapter studies solutions to the All-Pair Shortest-Path problem for sparse graphs when combining parallel algorithms and parallel-productivity methods in heterogeneous systems (ΠH-APSP solutions, see Sect. 2.5.2). As we have previously seen, this problem can be divided into independent Single-Source Shortest-Path subproblems, thus distributing the computation space into the different processing units that are usually present in modern shared-memory systems, i.e. CPUs and GPUs. Although the powerful GPUs are significantly faster than the CPUs, we will see that their combined use leads to further reduction on execution times. Furthermore, two different policies are compared to solve the scheduling issue: An equitable scheduling, where the workspace is equitably divided between all computational units independently of its nature, and a work-queue retrieving scheduling, where a computational unit retrieves a new task when it has finished its previous work, generating a dynamic balance.

5.1 Problem Description: Π-APSP Approach

The new generation of high performance computing (HPC) trends to assemble different kinds of multi-core CPU and many-core devices (like GPUs) in the same heterogeneous computing system. The goal of heterogeneous environments is to jointly exploit all computational capabilities of the devices with different hardware-resource configurations.

The different nature of these heterogeneous computational units (HCU) makes necessary to implement the same algorithm in different ways, in order to take the maximum profit of each underlying architecture. However, although each HCU has its own code implementation, usually some of them solve a problem faster than others due to its different computational resources. In order to alleviate this imbalance problem and to maximize the exploitation of the heterogeneous systems, different methods of load balancing can be applied.

Load-balancing is one of the challenging problems which has a tremendous impact on the performance of parallel applications, especially in heterogeneous environments. The objective of load-balancing methods is to distribute the workload proportionally, accord-

ing to the computational power of the devices. In this way, these methods allow to avoid device overloads when others are idle. However, in order to obtain a good performance exploiting heterogeneous systems, the programmer needs to manually implement these load-balancing methods. One example of a basic load-balancing technique is to assign more work to the most powerful HCU, as it could be the GPU, and the remaining work to the conventional CPUs.

In this chapter we are going to solve the APSP problem for sparse graphs combining parallel algorithms and parallel-productivity methods in heterogeneous systems. The first level of parallelism we have used is the parallelization of Dijkstra's algorithm. As we saw in Sect. 2.2.2, the naïve Dijkstra's algorithm is a greedy algorithm whose efficiency is based in the ordering of previously computed results. This feature makes its parallelization a difficult task. However, there are certain situations where parts of this ordering can be permuted without leading to wrong results neither to performance losses. As long as the APSP problem can be divided into independent SSSP subproblems, we use this feature to distribute the computation space into different processing units. Thus, a second level of parallelism is exploited by executing n times in parallel an SSSP solution, using a parallel algorithm (Π-SSSP) for the GPUs, and a sequential algorithm (SSSP) in the CPU cores.

5.2 State of the Art: Towards Heterogeneous Computing

Heterogeneous computing [18] is a computing paradigm that tries to jointly exploit different kind of computational units, not only the traditional CPU cores, but also graphical accelerators (GPUs) [166], multi-core chip coprocessor (XeonPhi) [167], or customized integrated circuits (FPGAs) [168], among others. During the last decade, the interest of the scientific community has significantly raised for these heterogeneous environments, although their programming is a tedious task that has a long learning curve. Compared to the classic homogeneous computing that involved only symmetric CPUs, this new computing paradigm offers higher peak performance in certain kind of parallel problems while being both energy and cost efficient [18, 169].

Load-balancing methods for heterogeneous systems try to properly distribute the workload among different computing units according to some criteria, such as their computational capabilities or its available hardware resources, with the aim to increase the total performance of the heterogeneous system. Depending on the moment when the workload is assigned to the different computational units, the load-balancing methods can be classified in static or dynamic. The rest of this section presents a brief discussion of different works that includes these load-balancing techniques.

The static load-balancing techniques defines the task distribution before starting the computation. The methods included in this category can be used for in those problems where the required workload information is available at that point. As example of very simple approaches, having different sizes of portions of the workload already fixed before computing, the authors of the works [170, 171] assign the bigger portions to the most powerful devices. Another approach is presented in [172, 173], that divided all the workload in tasks of the same size instead, but the number of tasks assigned to each device

depends on its computational capabilities or hardware resources, respectively.

Although there are situations where this information is not directly available, there are indicators that could be used to indirectly make an estimation or an approximation of it. For example, the authors of [174] created a model to estimate the execution time of each task, based on the number of instructions and input data size. Thus, determining the size of each single task before starting the computation, they have enough information to decide which hardware would be the most suitable to efficiently compute it, depending on this estimated temporal cost returned by the model. Similar works that also apply the estimations of a specific prediction model are [175] and [176]. The model of the first work studies the costs of data transfer between the CPU and GPU, and the returned scheduling distributes the workload with the aim to reduce these communications. Analogously, the model of the second work assigns the workload in order to reduce the memory bottlenecks.

On the other hand, when the information of the tasks is not available until the computation has significantly advanced, or it is completely unknown, usually it is better to apply dynamic load-balancing techniques. These methods schedule the workload while the algorithm is computing. The authors of [177, 178] dynamically create a complete dependence-graph in order to classify the application tasks as dependent or independent. Only independent tasks are launched to GPU devices in order to reduce the costly data transfers through inter-GPU communications. The work presented in [179] obtains a good load balance between CPUs and GPUs, by dynamically assigning, when available, big enough workload portions to the GPUs in order to keep these devices busy. The previously described work of [171] implements a dynamic load-balancing mechanism after the static distribution of the tasks into queues by performing task-stealing and task-donation techniques.

5.3 Load-balancing Techniques

For the hybrid approaches that simultaneously exploits different computational units, that are GPUs and CPU cores in our case, we define two different types of threads responsible of the computation in each of them. The first type of threads are executed on a CPU core with the aim to govern the logic of the algorithms designed for one GPU device: Control the data transferences from host memory to device memory, and launch the corresponding GPU kernels. If there are more GPU devices in the systems, more threads executed on different CPU cores are needed. The second type of threads are also executed on CPU cores but their purpose is to carry out the computations and logic of the algorithm designed for CPUs. Due to the huge differences between both computational units when processing each work unit, defined as a task, it is needed to balance the complete workload in order to obtain a good performance.

This section describes the different load-balancing policies we have implemented to distribute the different tasks among the computational units: Equitable Scheduling, and Work-queue retrieving Scheduling.

```
00:  #parallel                          /* Parallel region */
01:  if (idThread < numGPUs){           /* For GPUs */
02:      selectGPU(idThread);           /* GPU selection */
03:      atomic{ taskid = retrieve_work(taskQueue) };
04:      while( taskid != NULL ){
05:         launch_GPU_Kernel(taskid);
06:         atomic{ taskid = retrieve_work(taskQueue) };
07:      }
08:  }else{                             /* For CPU cores */
09:      atomic{ taskid = retrieve_work(taskQueue) };
10:      while( taskid != NULL ){
11:         launch_CPU_Kernel(taskid);
12:         atomic{ taskid = retrieve_work(taskQueue) };
13:      }
14:  }
15:  #end parallel
```

Figure 5.1: Work-queue retrieving technique implementation.

5.3.1 Equitable Scheduling

A simple way to apply load-balancing to a heterogeneous system is to equitably distribute the work without taking into account the computational capabilities of the devices. This kind of techniques usually lead to easy implementations, but at the expense of having a temporal cost equal to the time that the worst device needs to compute its work. Equitable Scheduling can be classified as a static load-balancing technique.

Our Equitable Scheduling (ES) approach statically divides the workspace between the computing threads giving to each one the same quantity of tasks. If nc represents the number of computing threads, id the thread identifier, and $nt = n/nc$ the number of tasks per thread, this approach makes each thread responsible for computing the tasks from $id \cdot nt$ to $id \cdot nt + nt - 1$. If this task division is not exact, the first threads takes one of the remaining tasks until there is no more work to do.

5.3.2 Work-queue retrieving Scheduling

The work-queue retrieving scheduling is commonly employed to accomplish a dynamic distribution of the work between any kind of hardware device during the execution of the program. The threads responsible of handling hardware accelerators of the heterogeneous system can retrieve a task from the global task queue. Note that the access to the global task queue must be implemented with some kind of synchronization in order to avoid that two or more devices retrieve the same task. Usually, this synchronization involves a bottleneck in the execution times. Work-queue retrieving scheduling can be classified as a runtime, dynamic load-balancing technique.

Our Work-queue retrieving Scheduling (WS) approach allows an idle thread that has finished its previous work to retrieve the following task t_i. This task is immediately eliminated from the queue at the moment it is taken. Then, the thread computes the corre-

sponding SSSP problem with node i as source. Finally, when the thread ends its work, it comes back to the global task queue in order to take another one, repeating the process till there is no more pending work. The synchronization of the task retrieving has been implemented using an atomic region. That means that only one thread can be retrieving the next task at any moment.

The algorithm shown in Fig. 5.1 is the pseudocode of work-queue retrieving technique to solve the APSP problem. The lines *01* and *08* indicate that the first threads are assigned to GPU devices and the rest to CPU cores. The *taskQueue* stores the list of all tasks, and the *atomic{}* primitive creates a mutual exclusion region to avoid a simultaneous retrieving of the same task from different idle threads.

5.4 Experimental Evaluation on a Heterogeneous Shared-Memory System

We will first describe the methodology used for our experiments, as well as the load-balancing techniques evaluated, the input set problems used, and the experimental results.

5.4.1 Methodology

In order to evaluate the efficiency and potential of the proposed solution for the APSP problem in heterogeneous systems, we have conducted two experiments with different objectives and input sets, but using the same described load-balancing techniques. The first one checks the efficiency of working in heterogeneous systems against a *reference implementation*, that uses just one GPU. This experiment also studies which load-balancing technique presents the best results using synthetic graphs with an irregular but known distribution. The second one compares the same techniques using bigger graphs and taking random sources, with the aim to evaluate their scalability computing arbitrary nodes whose required times to be solved is unknown.

A description of the platforms and devices used, the load-balancing involved to speedup the solution of the APSP, and the experiments carried out is presented below.

Target Architectures

The evaluated heterogeneous system is composed by a single shared-memory system with two different computational units, CPUs and two GPUs, with the following characteristics:

- The CPU of the host machine is an Intel(R) Core(TM) i7 CPU 960 3.20 GHz with 4 cores, and active hyperthreading.
- the first GPU is a NVIDIA GeForce GTX 680 (Kepler GK104).
- the second GPU is a NVIDIA GeForce GTX 480 (Fermi GF100).

Heterogeneous instances	Description
G_1	Single GPU thread (Kepler) (*reference implementation*)
E_2 / W_2	2 GPU threads (Fermi & Kepler)
E_3 / W_3	2 GPU threads + 1 CPU threads
E_4 / W_4	2 GPU threads + 2 CPU threads
E_6 / W_6	2 GPU threads + 4 CPU threads
E_8 / W_8	2 GPU threads + 6 CPU threads
E_{14} / W_{14}	2 GPU threads + 12 CPU threads
E_{16} / W_{16}	2 GPU threads + 14 CPU threads

Table 5.1: Experimental instances used on the shared-memory system. "E" instances use the equitable scheduling whereas the "W" instance use the work-queue retrieving load-balancing technique.

The shared-memory host machine used has 6 GB DDR3 of memory with an Ubuntu Desktop 10.10 (64 bits). The experiments have been carried out using the CUDA Toolkit 4.2.

Experiment I: Heterogeneous Approach for the Complete APSP

The objective of this experiment is to determine whether the use of a heterogeneous system, combining traditional CPU cores with modern GPUs, turns out to be a relevant approach to speed up computationally-costly problems such as the APSP. In order to check this hypothesis, we have compared the execution times of several instances of our heterogeneous implementation against the execution times of the *reference implementation*. This *reference implementation*, instance "G_1", is executed in the same shared-memory host machine of the previously described heterogeneous system, but it only uses the most modern GPU device as computational unit, the GeForce GTX 680 (architecture Kepler GK104), governed by one OpenMP thread.

Several instances with different number of OpenMP threads for both equitable and work-queue retrieving methods, have been executed in order to determine which load-balancing technique/configuration returns the best performance in different scenarios. We have tagged each instance, depending which load-balancing technique implements, with the label "E" for equitable scheduling instances, and "W" for work-queue retrieving scheduling instances, followed by a number that represents the number of OpenMP threads used (see Table 5.1). Thus, the evaluated instances "E_3" and "W_8" are a implementation of Equitable Scheduling with 3 threads and a implementation of Work-queue retrieving Scheduling with 8 threads respectively. The first two threads are always assigned to the two GPU hardware devices, one for each graphic accelerator. The rest of the threads are executed in the CPU cores. Therefore, the instances "E_2" and "W_2" only use the GPUs resources with the corresponding load-balancing technique.

All the instances have to resolve the complete APSP problem in synthetic graphs with 2^{20} nodes. As it is described in the following sections, these graphs have a known specific

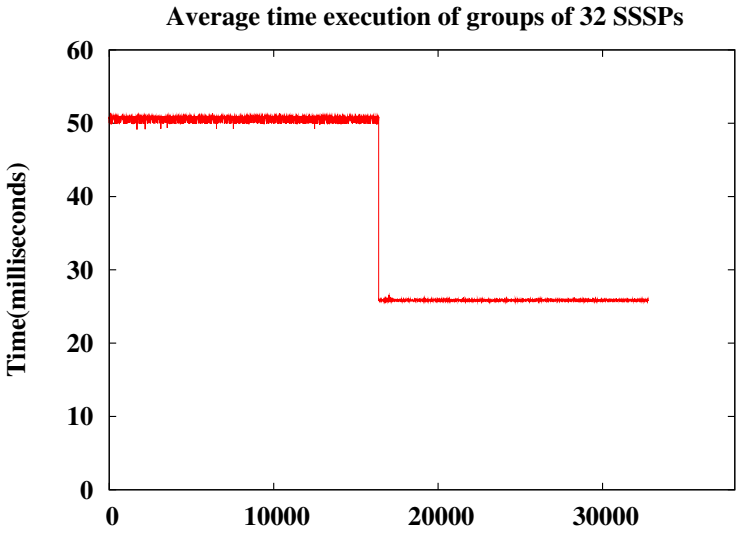


Figure 5.2: SSSP Execution time for different source nodes of the Martín graphs using the GeForce GTX680 (Kepler GK104).

node distribution where the cost of solving an SSSP problem for source nodes is correlated with the source node index. It allows to do predictions of the temporal costs of a task for a given computational unit. This is very useful in order to discover which policy suits better this kind of graphs, and under which certain situations.

Experiment II: Random Scalability for the Heterogeneous Approach

The objective of this second experiment is to study the behavior of the implemented load-balancing techniques in a scenario where it is not known the cost of the following task to be computed, and also to check the scalability of the instances in larger graphs within these conditions. Thus, instead of traversing the SSSP tasks to be computed in order, from source node 0 to n, the order in which tasks are assigned to devices is chosen randomly with a uniform distribution. In this way, the distribution made when using the equitable distribution will contain nodes costly nodes, and light nodes with the same probability. For the selection of these source nodes we have used the random function *srand48()* from the C standard library.

The bigger graphs computed in this experiment also belong to the same synthetic suite used in the previous scenario, but here the number of nodes of these graphs ranges from 2^{20} to $11 \cdot 2^{20}$. However, due to the large amount of computational load needed to solve the APSP in these graphs, we have bounded the problem to 512 random *source-nodes-to-all* in order to reduce the global execution time.

5.4.2 Input Set Characteristics

The input set used is composed by the collection of Martín's synthetic graphs, described in Sect. 3.3.2. The irregular distribution of this kind of synthetic graphs leads to differentiate two types of source nodes, in terms of the cost of solving each SSSP subproblem.

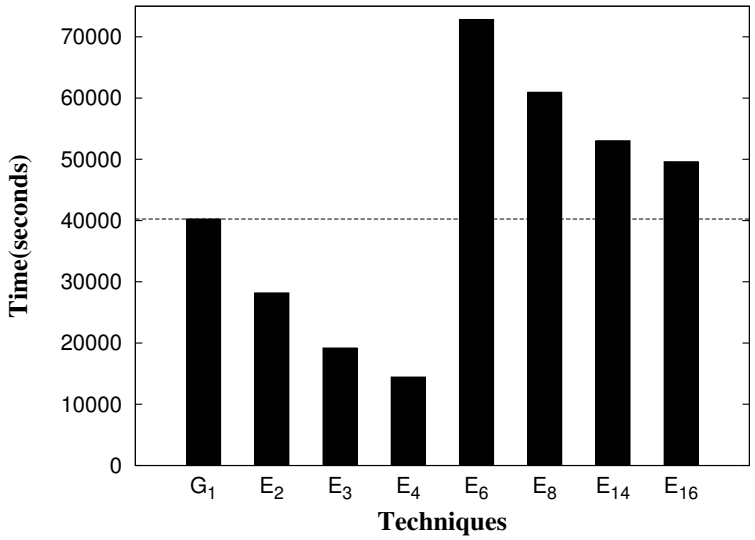


Figure 5.3: Execution times of different instances using the Equitable Scheduling policy for Martín's graphs with $|V| = 2^{20}$.

Figure 5.2 shows the execution time needed to solve each SSSP from source node 0 to source node n, grouping them in intervals of 32 nodes, using the GPU of the *reference implementation*, the Kepler GK104. It is possible to appreciate that the first half of nodes of the graph are more costly than the second half, where the required time to compute a SSSP is considerably smaller. This set with these particular characteristics and known distribution is very interesting for our designed experiments because we will be able to observe the different behavior of using this information in the Equitable Scheduling, and the versatility of the Work-queue retrieving Scheduling.

5.4.3 Experimental Results I - Complete APSP Evaluation

In this section we present the experimental results obtained for the execution of the complete APSP, using the Martín graphs with size $n = 2^{20}$, for the two considered load-balancing techniques.

Equitable Scheduling

Figure 5.3 presents the execution times of the equitable scheduling technique for instances with different number of OpenMP threads. The performance of the *reference* approach (G_1) is significantly improved when a second GPU device is used (E_2). However, a two-fold speedup is not reached because the architectures of the two GPUs are different. This means that the total execution time corresponds to the execution time of the less powerful GPU device. Nonetheless, E_2 presents a 30% performance improvement with respect to the *reference*.

The use of one and two additional CPU cores (E_3 and E_4) helps to decrease this critical execution time because the number of subproblems (SSSP problems) that the critical GPU has to resolve is reduced. The instance E_4 shows a 65% performance improvement

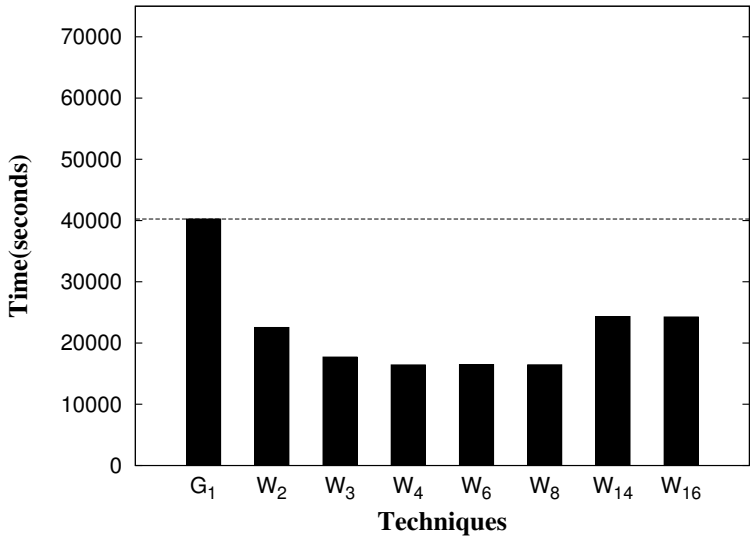


Figure 5.4: Execution times of different instances using the Work-queue retrieving Scheduling policy for Martín's graphs with $|V| = 2^{20}$.

compared with the *reference*. The more threads launched, the less the computation load assigned to each device. Nevertheless, due to the irregular nature of the graph (see the distribution time in Fig. 3.4), there is a threshold where the equitable partition approach overloads too much the CPU cores. This occurs when the most costly tasks, that were assigned to GPUs in previous equitable instances, are now assigned to CPU cores. For this reason, the total execution time of the approach E_6 is significantly increased, even surpassing the *reference* time. Furthermore, when more than six threads are launched, the total time execution is reduced again. This occurs because the number of tasks assigned per computational unit is less and all devices are used. However, their times still overpasses the *reference* times because the low number of assigned tasks to GPU devices is quickly computed, becoming idle whereas the CPU cores are still solving their assigned costly tasks.

Work-queue retrieving Scheduling

Figure 5.4 shows the execution time results of the work-queue retrieving scheduling technique for instances with different number of OpenMP threads. The performance of the *reference* approach (G_1) is significantly improved by any instance that uses the work-queue retrieving method (W_i). The instance that uses only two GPUs has a 44% performance improvement with respect to the *reference*. As we increase the number of OpenMP threads, more hardware devices are used, thus reducing the execution times. Although the most costly tasks are also taken by the CPU cores, while they are computing their subproblem, the GPUs are continuously retrieving tasks.

The instance with the fastest execution times is W_4, leading to a 60% performance improvement. However, when the number of launched threads exceeds the number of total computational units (W_{14} and W_{16}), the execution of threads that belong to the same CPU core is concurrent. This behavior leads to slight penalties, reaching to only a performance

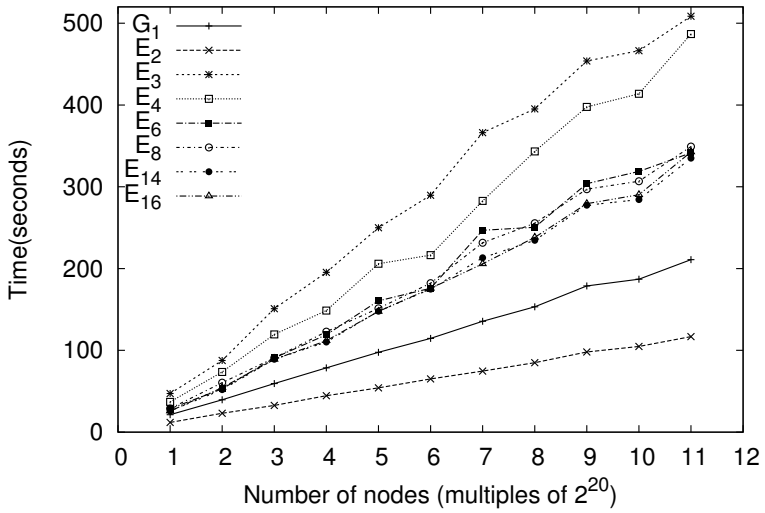


Figure 5.5: 512 nodes execution times using the Equitable Scheduling policy for Martín's graphs.

improvement of 40% with respect to the *reference*.

Experimental Conclusions

The best execution time for the complete APSP scenario is achieved with an equitable scheduling implementation, E_4, leading to an 65% of performance improvement compared with the *reference* G_1. However, the next approaches that closely follows this improvement are those that use a work-queue retrieving implementation ($W_{\{3...8\}}$), instead of other equitable scheduling instances with similar thread configurations.

5.4.4 Experimental Results II - Random Scalability Evaluation

In this section we present the experimental results obtained for the execution of 512 different random SSSP, using the complete graph suite due to Martín, with graphs which number of nodes range from 2^{20} to $11 \cdot 2^{20}$.

Equitable Scheduling

Figure 5.5 presents the execution times for instances for the equitable scheduling implementation, and different OpenMP launched threads for the different number of nodes of the graph considered. The execution times grow proportionally for bigger graphs as more paths to more nodes should be computed for a single source. The overhead of the scheduling technique is too low to be noticeable in this trends. The best performance is obtained with the E_2 configuration, leading to a 45% performance improvement with respect to the *reference*.

The heterogeneous approaches with CPU cores ($E_{\{3...16\}}$) have worse execution times than the *reference* because the CPUs are also computing costly tasks due to the random

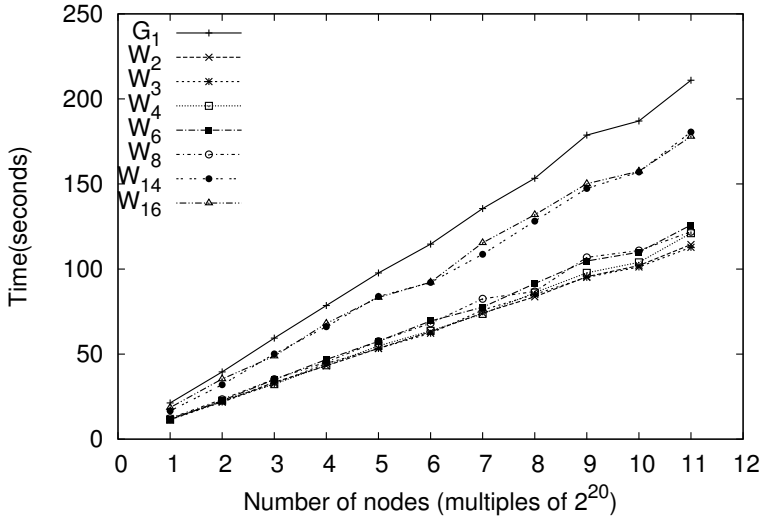


Figure 5.6: 512 nodes execution times using the Work-queue retrieving Scheduling policy for Martín's graphs.

nature of the node selection. However, as it happened in the Complete APSP scenario, this time is reduced when more threads are launched.

Work-queue retrieving Scheduling

Figure 5.6 shows the results when using the work-queue retrieving implementation for different number of OpenMP threads launched. As it happens in the Complete APSP scenario, the execution time of any work-queue retrieving instance ($W_{\{2...16\}}$) is better than the *reference* (G_1). The instance of two threads that only uses GPU devices, W_2, has a very good performance with respect to the *reference* (46% better). Inserting an additionally CPU core to the heterogeneous system, W_3, leads to an even better performance improvement of 47%. However, adding more than one CPU core to the heterogeneous system, $W_{\{4...16\}}$, leads to slightly worse execution times compared with the best one.

Experimental Conclusions

For the 512-source-to-all scenario, the best results are reached with a work-queue retrieving implementation, W_3, with a 47% improvement compared to G_1. All equitable scheduling instances involving CPU cores lose performance with respect to the *reference implementation*. The version that only uses GPUs, E_2, delivers a performance improvement of 45%.

5.5 Conclusions

We have presented solutions of the APSP problem for heterogeneous systems composed by GPUs and CPU cores using equitable and work-queue retrieving load-balancing techniques. The experimental results has shown that:

(1) The equitable scheduling can be tuned up using information of the graph features information to achieve the best performance times, avoiding critical code regions, but being very sensible to changes of the input graph.

(2) The work-queue retrieving implementation has a more consistent performance behavior than the equitable scheduling because it is more independent from the graph nature.

In particular, the best solution in the first scenario is achieved using a very specific equitable scheduling, with a performance improvement up to 65% compared with the *reference* single-GPU solution. However, the same configuration has not shown the same efficiency in the second experiment. Additionally, the remaining equitable instances that involve CPU cores have not shown a significantly performance improvement if the nature of the graph is not taken into account. On the other hand, most of the work-queue retrieving implementations have returned good performances for both tested scenarios.

Our first conclusion is that the joint use of very different computational power devices is useful to improve the total execution time compared with the use of a single-GPU implementation. The second conclusion is that a previous study of the nature of the input problem is very important, because it allows the programmer to better map the most costly tasks to the most powerful devices. In our case, the equitable scheduling that maps all costly tasks to the GPUs and leaves light ones to the CPU cores is the implementation that has delivered the best performance. Finally, the application of work-queue retrieving techniques results in a more versatile implementation with respect to the equitable scheduling because it is less sensitive to the nature of the input problem.

The work and conclusions described in this chapter have been published in the following paper:

- "The All-Pair Shortest-Path Problem in Shared-Memory Heterogeneous Systems," H. Ortega-Arranz, Y. Torres, D. R. Llanos and A. Gonzalez-Escribano, in book *High-Performance Computing on Complex Environments,* ser. Series on Parallel and Distributed Computing. John Wiley & Sons, Inc., 2014, pp. 283-299. Online, DOI: 10.1002/9781118711897.ch15

Chapter 6

TuCCompi Programming Model

During the last decade, parallel processing architectures have become a powerful tool to deal with massively-parallel problems, such as the All-Pair Shortest-Path problem, that benefits from High Performance Computing (HPC) techniques. As it is shown in the previous chapter, the use of heterogeneous environments, combining different computational processing devices, such as CPU cores and GPUs, turns out to be the most promising solution for HPC. However, as we have also described in Chapter 4, maximizing the performance of any GPU parallel implementation of an algorithm requires an in-depth knowledge about the GPU underlying architecture, becoming a tedious manual effort only suited for experienced programmers.

This chapter presents TuCCompi, a multi-layer abstract model that simplifies the programming on heterogeneous systems including hardware accelerators, by hiding the details of synchronization, deployment, and tuning. TuCCompi chooses optimal values for the configuration parameters using the kernel characterization criteria described in Chapter 4. As we will see, this programming model is very useful to tackle problems characterized by independent, high computational-load independent tasks, such as *embarrassingly-parallel* problems. We also present the description of a prototype implementation of the model, and its evaluation in different heterogeneous environments, using the All-Pair Shortest-Path problem, described in Sect. 2.4, as a case study.

6.1 Problem Description: The Need for Speed and the Lack of an Unified Solution

There are many computing-intensive problems that can be solved dividing them into many independent tasks that can be executed in parallel, without requiring any communication among them. They are called *embarrassingly-parallel* problems [180]. Many real problems are included in this category, such as index processing in web search [181], bag-of-tasks applications [182], traffic simulations [183], Bitcoin mining [184], volume rendering [185], some molecular physics computations [186], biomedical-domain data processing [187], or computational geometry problems [188]. Although the parallelization of embarrassingly-parallel problems does not require a very complex algorithm to

take advantage of parallel computing environments, their high amount of computational work requires High Performance Computing (HPC). Deployment, load balancing, and tasks synchronization details should be tackled by the programmer in a specific way for different applications, and different execution environments. In order to give support to the massive demand of HPC, the last trends focus on the use of heterogeneous environments including computational units of different nature, such as CPU cores, graphics processing units (GPUs) and other hardware accelerators. The exploitation of these environments offers a higher peak performance and a better efficiency compared to classical homogeneous cluster systems [18]. Due to these advantages, and since the cost of building heterogeneous systems is low, they are being incorporated into many different computational environments, from academic research clusters to supercomputing centers.

Despite the wide use of heterogeneous environments to execute massively-parallel problems, there are two issues that limit the usability of these systems. The first one is the lack of computing frameworks that can easily schedule the workload in such complex environments. Some works have been presented to integrate the use of different programming languages or tools [189, 190]. However, the programmer still needs to tackle different design and implementation problems related with each level of parallelism. These problems are specially more complex when integrating GPU programming techniques. The second limitation is the lack of a tuning methodology that efficiently unleashes all the power of GPU devices. Although there are languages, such as CUDA, that aim to reduce the programmer's burden in writing parallel applications, it is a difficult exercise to correctly tune the runtime configuration parameters in order to efficiently exploit all underlying GPU resources. As it was shown in Chapter 4 of this dissertation, the CUDA recommendations do not always lead to the optimum performance, leaving to the programmers the responsibility of searching for the best values. This search usually implies to carry out several time-consuming trial-and-error tests. Until now, there was not a parallel model that automatically selects the optimal values for CUDA configuration parameters, such as the threadblock size-shape, or the state of L1 cache memory, for each kernel. These optimization techniques significantly enhance the GPU performance.

6.2 State of the Art: Looking for One Tool to Rule All Parallel Levels

There are several works integrating different parallel languages or models in tools to consider different levels of parallelism. We can find the following contributions as examples of works that combine two different levels of parallelism, such as OpenMP with MPI, or with CUDA in multiple GPUs. The work presented in [191] is a framework for the development of hybrid MPI+OpenMP programs, generating parallel code depending on the features extracted when compiling the input sequential functions. However, it does not support conversions for CUDA. Similar to the previous approach, the tool llCoMP [189] is another source-to-source compiler that translates C annotated code to MPI + OpenMP or CUDA code, that is only focused in parallel-loop problems. Additionally, this compiler does not support the joint use of CUDA with the other parallel models, leaving its suitability for heterogeneous environments limited. Besides this, the llCoMP compiler does

not easily support the use of new GPU architectures or other kind of accelerators. Other programming library for hybrid architectures supporting GPUs is SkelCL [192]. It tries to enhance the OpenCL interface in order to coordinate different GPUs of the same shared-memory machine. The limitations of this library are similar to the previously described works, it does not support load distribution between GPUs of different machines, or even, other computational units of different nature, such as the CPU cores.

Going further, the following paragraphs describe approaches that combines all traditional parallelism levels combining the well-known OpenMP, MPI, and CUDA models together. A parallel programming approach using hybrid CUDA, MPI and OpenMP programming is presented in [193]. The authors focus on a model to solve iterative problems. However, they do not take into account any generic CUDA optimization technique, and their model does not support any mechanism to include new load distribution policies. The authors in [194] have created a hybrid tool, that includes the same parallel models used by the previous mentioned work, to solve a ray-casting volume rendering problem. They test the system scalability when the input data size is increased. The limitations of this tool are that it is only focused in a single parallel application, not even including any CUDA optimization technique, nor any automatic mechanism to efficiently exploit heterogeneous environments.

A more general approach is the work presented in [190], where the authors proposed a framework called OMPICUDA used for developing parallel applications on hybrid CPU/GPU clusters by mixing OpenMP, MPI and CUDA programming models. This framework presents some limitations: it does not support recursive functions, nor GPUs with 1.x architecture. Additionally, it cannot be easily modified to support a new parallel model. The task programming library StarPU [195] does not presents such limitations and has different optimized heterogeneous scheduling techniques for CPU/GPU clusters. However, none of these approaches provides or considers any tuning techniques for better exploiting GPU capabilities, or policies to select proper values of CUDA configuration parameters. A skeleton programming framework that uses StarPU is SkePU [196]. SkePU tries to find the optimal threadblock size by automatically checking all possibilities using trial-and-error executions, but it does not provide a model for tuning this parameter, nor a easy system to change proper values for new architectures. Finally, all presented works do not support the automatic exploitation of the concurrent-kernels feature of modern GPUs.

6.3 TuCCompi: The Distributed Heterogeneous Computing Model

TuCCompi (Tuned, Concurrent Cuda, OpenMP and MPI), is a multi-layer, skeleton-based abstract model, that transparently exploits heterogeneous systems and squeezes the GPU capabilities by automatically choosing the optimal values for important runtime configuration parameters. Each layer represents a level of parallelism. The first layer handles the distributed-memory environment, coordinating different shared-memory systems (nodes), whereas the second layer manages the computational units that are inside the nodes. The third layer automatically deploys the execution in the hardware accelerators, such as the GPUs, whereas the fourth layer automatically handles concurrent works in-

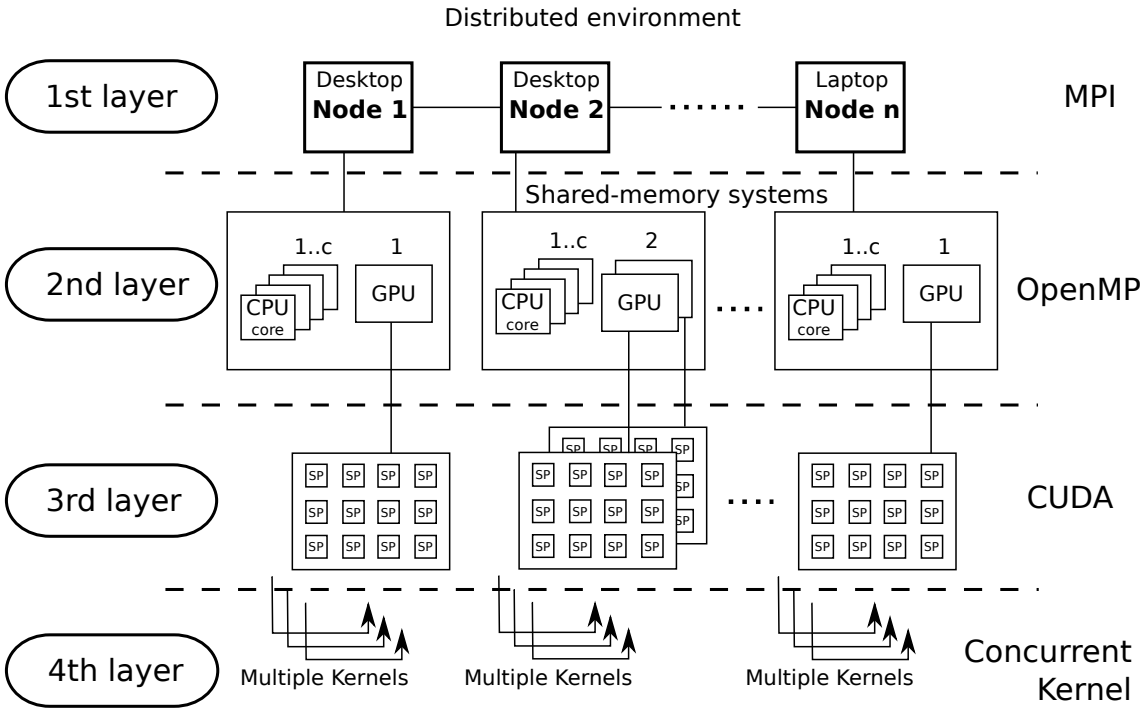


Figure 6.1: Layer deployment of TuCCompi model in a heterogeneous cluster.

side these GPUs. Finally, an internal tuning mechanism automatically selects the optimal values for GPU configuration parameters for each kernel, and each GPU architecture.

These execution layers are integrated with different coordination mechanisms, that are all abstracted to provide the programmer an unified view of the computing in heterogeneous systems. She has to program her applications in two *programming levels*: (1) a *coordination programming level*, that abstracts the work distribution across the computational units inside the distributed shared-memory nodes (1st and 2nd Layers); and (2) a *deployment programming level*, that abstracts the management of computational unit of different nature (3rd, 4th and Tuning Layers).

6.3.1 The Multi-Layer Architecture

This section gives a description of the different layers defined in our model. A graphical representation is depicted in Fig. 6.1.

The 1st Layer (Distributed Environment)

Nowadays, one of the most economic ways to assemble a heterogeneous system is to interconnect a set of different individual machines, also called nodes, such as personal computers, laptops, virtual host machines, or even other supercomputing systems composed in turn by other machines. It is necessary to apply communication and synchronization mechanisms in order to coordinate these nodes. The first layer of TuCCompi (see Fig. 6.1) is responsible of managing this node coordination without taking into account the hardware details and features of each machine. This layer is abstracted at the *coordination*

programming level, allowing the programmer to skip thinking in terms of more complex message-passing models.

The 2nd Layer (Shared-Memory Systems)

Nodes are nowadays composed by several processing units that share a global address space. Additionally, there are other accelerator devices, such as GPUs, and Xeon Phi, that are usually controlled by a host system (CPU) and are capable of executing kernels independently. In this layer of TuCCompi we use the concept of "computational unit" for any CPU core or device hosted in a node. This second layer is responsible of the coordination of all computational units inside the node. For the programmer's point of view, this layer is also encapsulated in the abstraction of the *coordination programming level*. It also hides the fact that each special device is controlled by a dedicated thread that executes a different code. The programmer sees all devices and CPU cores in an homogeneous form.

The 3rd Layer (GPU Devices)

This layer implements the abstraction used at the *deployment programming level*. It is the responsible of the coordination and deployment actions needed for special devices, such as GPUs, or Xeon Phis, in an homogeneous form. This is done by hiding the details needed to manage different address spaces, offloading codes, etc.

The 4th Layer (Concurrent GPU Kernel Execution)

The most recent NVDIA GPUs support concurrent-kernel execution [13], where different kernels of the same application context can be executed on a GPU at the same time. This feature is very helpful when kernels that use just few resources are launched, allowing a concurrent execution of other kernels, and thus, exploiting at the same time all resources of the device. Although at first glance this feature seems to be profitable only when low resource-consuming kernels are launched, the concurrent execution of higher resource-consuming kernels also gives performance gains. This occurs because several kernels of the same application context work on the same memory areas taking advantage of the data-caches, originating less number of cache-misses and therefore alleviating the global memory bottlenecks. The programmer provides a parameter to define the number of tasks that will be concurrently deployed in a single GPU for each application. This layer internally takes care of the synchronization of the concurrent kernel launching. It contributes to the functionalities encapsulated in the *deployment programming level*.

The Tuning layer

While correctness of an NVIDIA CUDA program is easy to achieve, the optimal exploitation of the GPU computational capabilities is much more complicated than in traditional CPU cores. Usually, it requires an extensive CUDA programming experience. Some examples of code tuning strategies are the choice of an appropriate threadblock size and shape, the maximization of the coalescing of the memory accesses, or the occupancy

optimization of the Streaming Multiprocessors, among others. Moreover, the resource differences between each GPU architecture and release, such as the number of computational units, cache-sizes, and other features, make it even more difficult to find the optimal configuration for a given GPU. Besides this, the optimal values also depend on the memory access pattern and the characteristics of the code of each executed kernel. This layer allow the programmer to supply to the *deployment programming level* with an abstract characterization of the CUDA kernel codes in terms of human-understandable features. With these values, the model internally chooses proper values for the execution parameters. This solution also opens the possibility to integrate techniques to automatically analyze and characterize the CUDA kernel codes for specific GPU devices.

6.3.2 TuCCompi Model Usage

To build a program using TuCCompi, a programmer should provide the following elements (see Fig. 6.2): (1) *Coordination programming level*, implemented as a main C language program using TuCCompi primitives and macros, and (2) *Deployment programming level*, including the sequential-CPU and the parallel-GPU specific codes for each application, named as `PLUG-IN_CPU` and `PLUG-IN_GPU` respectively, and characterizations of the accelerator kernel codes.

In this way, the application programmer does not have to provide: (a) the values of GPU configuration parameters for an optimal execution on each different GPU, (b) the code implementation for concurrent kernel deployment, (c) the code implementation for the management of the distributed and shared computational-units, nor (d) the communication between all involved nodes.

Coordination Programming Level - TuCCompi Main Program Implementation

Figure 6.3 shows an example of the code that the user has to implement in order to start and control the execution. The primitive `TuCCompi_COMM` in Line M01 initializes the system. Afterwards, the user can introduce his code, including variable declarations, initializations and the sequential code needed for the application. Line M03 shows the primitive needed to set the number of kernels that the GPU devices will execute concurrently (information for the 4th execution layer). Line M04 shows the primitive used to initialize and execute the functions implemented in the corresponding plug-ins. This synchronization expression transparently executes the CPU-plugin code for the CPU cores, or the specialized GPU-plugin code for the GPU devices, using the same semantics, across a whole heterogeneous cluster. The first parameter of this macro represents the kind of scheduling policy desired by the user (described below). It is used internally by the 1st and 2nd execution layers to balance the workload across the different computational units. Line M05 shows the primitive needed to make the process wait until all computational units of all nodes have finished. The user is free to insert more code to execute other kernels, before the finalization of the heterogeneous cluster communication, shown in line M07.

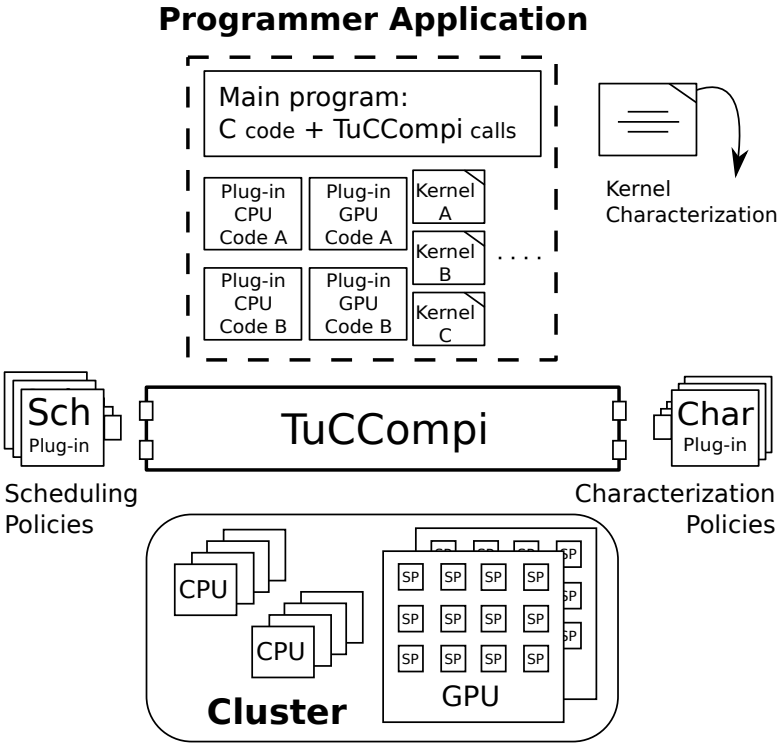


Figure 6.2: TuCCompi model usage. Elements in the dashed box are provided by the programmer. Note that the user can develop different versions of each plug-in (Code A, Code B, ...) but only one at a time will be deployed into TuCCompi framework.

```
M00: main( ){
M01:   TuCCompi_COMM( );
M02:   (main user code)
M03:   TuCCompi_SETCK( number );
M04:   TuCCompi_PARALLEL(MS, plugin_Cpu(..), plugin_Gpu(..));
M05:   TuCCompi_SYN( );
M06:   (main user code)
M07:   TuCCompi_ENDCOMM( );
M08: }//main
```

Figure 6.3: User implementation of the TuCCompi main-program. The programmer has to add to his code the boxed primitives.

Coordination Programming Level - Workload Scheduling

The TuCCompi model includes three different policies to distribute the workload between all available cluster resources through the first parameter of the primitive shown in line M04 of Fig. 6.3.

The first one, `EQ1`, is an equitable policy that schedules the same number of tasks to each node of the 1st layer (distributed memory environment), independently of the number of CPU cores, GPUs, or other accelerators that the nodes have inside. Later, each

```
C00:  plugin_Cpu(user_vars ...){
C01:    (Cpu user code)
C02:  }//pluginCPU
```

```
G00:  plugin_Gpu(user_vars ...){
G01:    (Gpu user code)
G02:    TuCCompi_GPULAUNCH(kernelname1, total_numthreads,
                  TuCCompi_PARLLCK(vector1, type, lng), ...);
G03:    TuCCompi_GPUSYN( );
G04:    TuCCompi_GPULAUNCH(kernelname2, total_numthreads2,
                  TuCCompi_PARLLCK(vector2, type, lng), ...);
G05:    TuCCompi_GPUSYN( );
G06:  }//pluginGPU
```

Figure 6.4: Plugin_Cpu (top) and Plugin_Gpu (down) interfaces. The programmer adds to his code the boxed pieces of code to deploy the CPU plugin in TuCCompi, and he has to replace the CUDA kernel launch primitives for the boxed TuCCompi macros for the GPU plugin.

node equally divides the assigned workload between all its own computational units (CPU core/Accel.), also in a balanced way.

The second one, `EQ2`, is also an equitable policy, but it divides the workspace directly between the computational units of the whole cluster at the 2nd layer. The workspace division does not consider the computational unit nature.

The third one, `MS`, follows a master-slave model. One computational unit is sacrificed to act as the master, and the rest of the computational units work as slaves. The slaves enter into a working loop, requesting tasks from the master when they become idle, until the master sends a termination signal to them. Thus, the more powerful units will ask for more work, and therefore they will process more tasks than the less powerful units. As the master can be located at any cluster node, these asking-for-tasks requests are issued through distributed-environment communications.

Additionally, TuCCompi also offers the possibility of including a scheduling policy programmed by the user through the *Scheduling plug-in* (see Sect. 6.4).

Deployment Programming Level - User-code Plug-ins

Figure 6.4 (top) shows the interface of the sequential code that will be executed in a CPU core computational unit. The user is responsible for inserting the code to implement the algorithm that solves a single task (line C01, `Cpu user code`).

Figure 6.4 (bottom) shows the code that will be executed in a CPU thread assigned by TuCCompi to manage one or more associated GPUs. The control of the GPU often involves active waits. In this case, a CPU core should be sacrificed to execute this GPU-controller thread. The user should define the code that handles the logic control of the algorithm that comprises the use of one or several GPU kernels. This code will be responsible of launching the corresponding kernels. Line G02 shows the TuCCompi macro that

carries out a kernel launch, with the name of the kernel as first parameter, and followed by other user variables that have been previously allocated in the GPU. The model transparently executes as many kernel instances as indicated by the programmer in the main control program (CK value) (see line M03 of Fig. 6.3). Every concurrent kernel launched will need its own workspace to compute its results. The second primitive of line G02 gives to the kernel one memory pointer for each data structure needed. The needed parameters are: The variable name; the native type of the elements that it contains; and the number of elements that compounds it. As we said before, the algorithm implementation can require the execution of different kernels that should be sequentially launched for a single task computation (line G04). The TuCCompi primitive of line G03 forces the CPU to wait for the finalization of an executing kernel, or kernels concurrently launched, providing a synchronization mechanism.

***Deployment Programming Level* - Kernel Characterization**

The user has to provide a general characterization of the kernels along with its definition. This information is easily expressed in our prototype implementation through the `TuCCompi_KERNELCHAR( kernel_name, num_dims, A, B, C, D)` primitive. The values for parameters *A*, *B*, *C* and *D* have to be chosen from the kernel-characterization classification shown in Table 6.1. TuCCompi model will automatically optimize the use of the underlying hardware of any kind of GPU found in the platform, for each possible combination of these parameters. The current prototype implementation includes support for any GPU with Fermi or Kepler CUDA architecture, following the guidelines and optimizations proposed in the work of [16], and extended in this Ph.D. thesis (see Chapter 4).

Figure 6.5 shows some examples of the code used to characterize the kernels. Lines K00 and K04 describes the characterization of kernels *k1* and *k2* respectively, indicating the kernel name, the number of dimensions of the threadblock, and the classes chosen from the classification criteria described in Table. 6.1. In the case that the user does not know how to classify the kernels, he can use the default values provided by the model. The primitive used for this default case is `TuCCompi_KERNELCHAR(kernel_name, num_dim, def, def, def, def)`. Note that this `def` word used to configure the *default configuration* cannot be combined with any other word that represents a kernel characterization.

6.3.3 The External-Work Attachable to TuCCompi

In order to facilitate to the programmer the porting from sequential code to parallel code, and vice-versa, the functionality of TuCCompi could be complemented with other works related with this kind of code transformation, that are already present in the scientific community. For example, accULL [197] receives a sequential code and automatically transforms it to parallel GPU code. Another example of code transformation is Ocelot [198], that works in the opposite way. Given a GPU implementation, Ocelot transforms it to sequential code. MCUDA [161] is another tool/framework that also makes GPU-to-CPU code conversion. TuCCompi model does not aim to deal with code transformations, but these works can be easily attached as previous functional modules to our multilayer model (see Fig. 6.6). Another attachable module could be the work of elastic kernels

Parameter	Description	Choice
A	Global memory-access pattern	`scatter`/ `mediumcoalesced`/ `coalesced`/ `def`
B	Ratio of arithmetic instructions per thread compared to the global-memory accesses	`high`/ `medium`/ `low`/ `def`
C	Ratio of L1 cache memory lines evictions compared to the size of this memory	`high`/ `medium`/ `low`/ `def`
D	Ratio of memory data reutilization compared to the number of arithmetic instruction per thread	`high`/ `medium`/ `low`/ `def`

Table 6.1: TuCCompi kernel-characterization classification. The `def` choice can be used when the user does not know the kernel characterization.

```
K00: TuCCompi_KERNELCHAR(kernelname1, 2, scatter, none, high, low);
K01: __global__ void kernelname1 (...){
K02:   (kernel implementation)
K03: }

K04: TuCCompi_KERNELCHAR(kernelname2, 1, coalesced, low, low, high);
K05: __global__ void kernelname2 (...){
K06:   (kernel implementation)
K07: }
```

Figure 6.5: Example of kernel characterizations and implementations. The programmer adds the boxed primitive before the kernel implementation to characterize it.

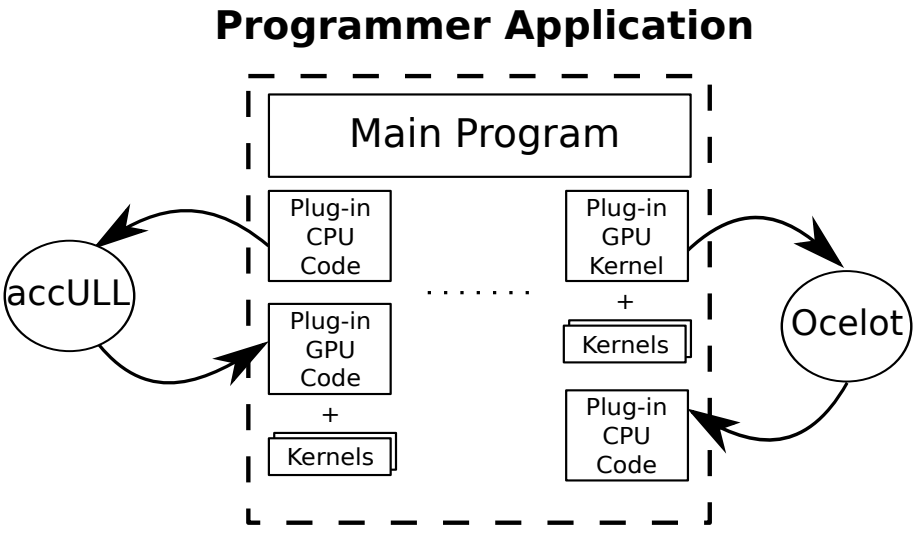


Figure 6.6: Usage of TuCCompi with attachable code-transformation modules.

presented in [199]. They do manual source-to-source code transformations in order to obtain GPU kernels that exploit more the multikernel feature of the GPU devices.

6.4 The Prototype Internals

In this section we will discuss the internals of the TuCCompi prototype we have developed to test the model functionality. The functions and primitives described here have a correspondence with the model layers described in Sect. 6.3.1.

Cluster Inter-Node Communication (1st Layer): `TuCCompi_COMM`

Once a TuCCompi program is in execution, each process initializes its MPI-identification variables, and enters into a global communication step carried out by exchanging a few MPI messages. An arbitrary process is the coordination handler, that we name as parent process. It receives from the remaining processes the number of the computational resources they are able to manage. Then, the parent process sends to each process a global identification number for each resource inside the whole heterogeneous cluster. Additionally, the parent process sends more information about the heterogeneous cluster, such as the total number of computational units and the numeration per node.

Fig. 6.7 shows the implementation of this first phase. We will now review the data structures involved. The `v_cu` vector stores the number of computational units from each process. The `v_id` vector stores the number from which the numeration of computational units should start for the process `i`. The `total_cu` variable stores the total number of computational units. The `id_mpi` variable stores the identifier of the MPI process. The `n_proc` variable stores the total number of MPI processes. Finally, the `PARENT` constant is the identifier of the MPI process that coordinates the communication. In this first phase, lines 02-04 initialize some values and ask to the second layer how many computational units has the machine. Lines 05-09 receive information from the rest of processes. Lines 10-14 perform the heterogeneous-environment information shipping. Lines 15-21 correspond to the behavior of the rest of process, that looks up for the available resources, sends this value to parent process and receives the cluster information. If the programmer needs to modify the information exchanged during this phase, she just has to slightly modify some code lines of this function only, that was designed to gather the behavior of both kind of processes, the coordinator and the rest.

Cluster In-Node Synchronization (2nd Layer): `TuCCompi_PARALLEL`

Once the TuCCompi model has been initialized and the user variables have been declared, the `TuCCompi_PARALLEL` primitive automatically creates as many OpenMP threads as the number of CPU cores that will perform the parallel execution on each node. Figure 6.8 shows the code that is executed when the programmer uses the `TuCCompi_PARALLEL` primitive for the master-slave scheduling policy, (`EQ1` and `EQ2` policies are not shown due to space restrictions). The master-slave implementation just divides the workload between the cluster nodes and the computational units. The slaves execute each task without

```
00:  comm(v_cu, v_id, total_cu, id_mpi, n_proc){
01:    if ( id_mpi == PARENT){
02:      v_id [PARENT] = 0;
03:      v_cu [PARENT] = second_layer_resources()
04:      total_cu = v_cu[PARENT];
05:      for (int i=1; i<n_proc; i++){
06:        v_id [i] = total_cu;
07:        RECV( v_cu [i], i);
08:        total_cu += v_cu [i];
09:      }
10:      for(int i=1; i<n_proc; i++){
11:        SEND(v_id, i);
12:        SEND(v_hilos, i);
13:        SEND(total_cu, i);
14:      }
15:    }else{
16:      cu_local = second_layer_resources()
17:      SEND(cu_local, PARENT_process);
18:      RECV(v_id, PARENT_process);
19:      RECV(v_cu, PARENT_process);
20:      RECV(total_cu, PARENT_process);
21:    }
22:  }
```

Figure 6.7: Implementation of the `comm()` cluster-information gathering function, called from `TuCCompi_COMM()`.

needing any more communication with the master. Lines 05-07 initialize the intra-node computational units identifiers. Lines 08-09 check whether any of the current OpenMP threads should act as the master, executing the default master function. If there are GPUs, each one is governed by its corresponding OpenMP thread. Therefore, lines 10-15 obtain the device properties, entering into the ask-for-tasks working loop and executing the parallel GPU code provided by the pluginGPU (see Sect. 6.3.2). The normal CPU cores also enter into the ask-for-tasks working loop but executing the code of the pluginCPU (see Sect. 6.3.2) (lines 16-18). Lines 14 and 17 show the code responsible for obtaining the following task to be executed by invoking the pluginSLAVE macro. When all tasks have been already computed, the function behind this macro returns a value higher than the total number of tasks in order to leave the ask-for-tasks working loop.

Kernel Launch and Concurrent Kernel Execution (3rd and 4th Layers): `TuCCompi_GPULAUNCH`

Before the task-threads spawn (Line 03 of Fig. 6.8), the second layer (shared-memory process) detects how many GPUs are available in the shared-memory node (Line 01 of Fig. 6.8). Once in the parallel region, an OpenMP thread is assigned to one CPU

```
00: #define TuCCompi_PARALLEL(MS, pluginCPU, pluginGPU)\
01:   cudaGetDeviceCount(&TuCCompi_gpuCount);\
02:   omp_set_num_threads(omp_get_num_procs());\
03:   #pragma omp parallel\
04:   {\
05:     int task;\
06:     int TuCCompi_local_id  = omp_get_thread_num();\
07:     int TuCCompi_global_id = v_id[id_mpi] + TuCCompi_local_id
;\
08:     if( TuCCompi_global_id == TuCCompi_master) {\
09:       pluginMASTER;\
10:     } else if( TuCCompi_local_id < TuCCompi_gpuCount ){\
11:      cudaDeviceProp props;\
12:      cudaGetDeviceProperties(&prop,TuCCompi_local_id);\
13:      int gpu_arch = props.major;\
14:      while( (task = pluginSLAVE) < total_tasks)\
15:        pluginGPU;\
16:     } else\
17:      while( (task = pluginSLAVE) < total_tasks)\
18:         pluginCPU;\
19:   }#pragma

20: #define TuCCompi_SYN( )\
21:   #pragma omp barrier\
22:   MPI_Barrier(MPI_COMM_WORLD)

23: #define TuCCompi_END( )\
24:   MPI_Finalize();
```

Figure 6.8: `TuCCompi_PARALLEL()` and other macro-definition codes.

core in order to govern each hardware accelerator, also storing some relevant properties of the GPU, such as its architecture (Lines 11-13 of Fig. 6.8). Afterwards, this thread is the responsible of handling the logic control of the algorithm implemented in pluginGPU, actually launching the different kernels invoked through the primitive `TuCCompi_GPULAUNCH(`*`kernel_name, total_numthreads, kernel_vars`*`)`, whose internal definition is shown in Fig. 6.9.

The model automatically detects if the concurrent execution of several kernels (the multikernel feature) is supported by the GPU using the properties previously retrieved. Otherwise, the model always launches only one kernel at the same time. The multikernel feature is also embedded in the GPU launching primitive (Line 01 of Fig. 6.9). Additionally, in order to make possible that each kernel works in a different workspace, the `PARLLCK(`*`variable_name, variable_type, variable_length`*`)` macro automatically computes the memory offset allocation of the corresponding variables that are task-dependent (Lines 04-05 of Fig. 6.9).

```
00: #define TuCCompi_GPULAUNCH(k_name,total_numthreads,uservars)\
01:   for( int parll = 0; parll < CK; parll++)\
02:     k_name<<<t_grid(k_name, arch, total_numthreads),\
03:              t_threads(k_name, arch)>>>(uservars)\

04: #define TuCCompi_PARLLCK(var_name,var_type,var_length)\
05:   var_name + parll * sizeof(var_type) * var_length

06: #define TuCCompi_GPUSYN( )\
07:   cudaThreadSynchronize()
```

Figure 6.9: Declarations for the automatic kernel launch and multikernel support.

Automatic Kernel Tuning (Tuning Layer): `TuCCompi_KERNELCHAR`

The optimization layer automatically configures the kernel parameters depending on:

(1) The GPU architecture where it is going to be launched.

(2) The kernel characteristics provided by the user.

As long as the model recognizes the architecture of the GPUs that are present in each cluster node, it only needs to know the characterization of each user-defined kernel. This characterization is indicated by the programmer before the kernel definition (see previous example of Fig. 6.5). It is automatically mapped to a structure that contains the optimal values for all classified architectures (see Fig. 6.10). As can be seen in lines 02-03 of Fig. 6.9, these values are already embedded in the primitive of kernel launching as a call to the `t_grid()` function, that returns the optimal number of blocks, and `t_threads()`, that returns the optimal number of threads per block. In this way, TuCCompi automatically selects the optimal configuration of the grid, and the threadblock size-shape. Note that, currently, our prototype only supports one dimension for the threadblock shape, and the `t_grid()` function returns correct values if the size value, that is the total number of threads, is multiple of the corresponding optimal threadblock size. The modifications to support non-multiple total sizes can be done straightforward applying the ceil function.

If the user does not know how to characterize his kernel, the default values can be used. These values are the ones recommended by CUDA [21], to maximize the SM Occupancy. Although these recommended values sometimes work well, we have seen in Chapter 4 that there could be performance differences compared to the proper values proposed in TuCCompi (as we stated in Sect. 6.3.2).

Advanced TuCCompi Model Features

TuCCompi model has additional functionalities and features, such as the possibility of executing a more complex workload scheduling policy created by the user, or the possibility of changing the optimal values for each kernel and GPU. We will now describe two plugin systems that help to introduce these functionalities.

```
00: #define TuCCompi_KERNELCHAR(name, numDim, A, B, C, D)\
01:     int k_##name[4] = k_##A##B##C##D

02: #define t_threads(name,arch)   k_##name[arch]
03: #define t_grid(name,arch,size) size/k_##name[arch]

04: #define k_defdefdefdef {256, 256, 256, 256}
05: #define k_scatterlowhighlow {256, 256, 96, 64}
06: #define k_coalescedlowlowmedium {256, 128, 192, 128}
07: #define ...
```

Figure 6.10: Some declaration examples for the automatic GPU kernel optimizations.

Scheduling plug-in system

The current master-slave policy involved in the prototype gives a simple implementation where only one task is scheduled to each slave independently of its computational power. The master and the slaves execute, respectively, the master-function and slave-function codes provided in the scheduling plug-in. Additionally, if the problem or the user need a different granularity or a particular load distribution that follows a special pattern or policy, the model allows the programmer to use his own scheduling implementation. This is done by injecting new distribution policies through a *scheduling plug-in* system, using an extended primitive `TuCCompi_PARALLEL(MS, pluginCPU, pluginGPU, pluginMASTER, pluginSLAVE)`.

This is very useful if the user has in the heterogeneous environment some devices that work very fast compared with the rest. In this case, it may be a good choice that the master gives them a pack of tasks instead of a single one. When an OpenMP thread responsible of a GPU device asks for tasks, it can retrieve the corresponding device information that could be sent to the master in the requesting message. With this information, the master could give a pack of tasks to the most powerful devices and a single one to the less powerful computational units. Thus, the master can produce a more complex distribution depending on the capabilities of the computational units that are asking for work. Figure 6.11 shows a customized implementation of the *scheduling plug-in* created for the case study.

Characterization plug-in system

The optimal values for GPU configurations used by the Characterization plug-in are stored in a file. These values can be easily updated if new devices with different architectures or resources are added to the heterogeneous environment. Moreover, it is also easy to modify these values if the user wants to experiment with new combinations of parameters.

```
00:void master_scheduler(task_ini,total_tasks){
01: int next_task = task_ini;
02: while( next_task < total_tasks ){
03:  RECV(id_slave, any_slave, slave_info);
04:  if( slave_info == (FERMI or KEPLER) ){
05:   if( (next_task + CK) <= total_tasks){
06:     SEND(next_task, id_slave);
07:     next_task = next_task + CK;
08:   }else{
09:     SEND(END_SIGNAL, id_slave);
10:     token++;
11:   }
12:  }else{
13:   SEND(next_task, id_slave);
14:   next_task++;
15:  }
16: }
17: while( token < total_cu-1 ){
18:  RECV(id_slave, any_slave);
19:  SEND(END_SIGNAL, id_slave);
20:  token++;
21: }
22:}
```

```
23:  int slave(id_slave, mpi_master, tag){
24:    SEND(id_slave, mpi_master, tag);
25:    RECV(task, mpi_master, id_slave);
26:    return task;
27:  }
```

Figure 6.11: Our case-study implementation for the functions, master (top) and slave (bottom), of the distribution plug-in.

6.5 Porting the SSSP Implementation to TuCCompi

In the previous chapters of this dissertation we have presented two different ways to solve the APSP problem through the $n \times SSSP$ approach. The first one, described in Chapter 4, uses one single GPU, where several tuned kernels are launched concurrently, and each of them takes care of one SSSP task at a time. The second one, described in Chapter 5, uses a complete heterogeneous shared-memory system, combining 8 CPUs and 2 GPUs. Each of these computational units (CPU/GPU) takes care of one SSSP task at a time, so the concurrent kernel feature of the GPU and the tuning techniques were not exploited.

In order to illustrate the capabilities of the TuCCompi framework prototype, we will use the APSP problem as our case study. This problem is a representative example with good characteristics to evaluate the model features, carrying out, at the same time, heterogeneous computing in more than one shared-memory system and a tuned, concurrent

```
00: TuCCompi_KERNELCHAR(relax, 1, mediumcoalesced, low, high, high);
01: __global__ void relax (...){
02:   (relax kernel implementation)
03: }

04: TuCCompi_KERNELCHAR(minimum, 1, coalesced, medium, low, low);
05: __global__ void minimum (...){
06:   (minimum kernel implementation)
07: }

08: TuCCompi_KERNELCHAR(update, 1, coalesced, low, low, low);
09: __global__ void update (...){
010:   (update kernel implementation)
011: }
```

Figure 6.12: Inserting information for TuCCompi prototype related with the characterization of the SSSP kernels.

```
00:  SSSP_pluginGPU(...){
01:    user code
02:    while( ){
03:      TuCCompi_GPULAUNCH(relax,num_v,v_d,a_d,w_d,
07:          PARLLCK(p_d, bool, num_v),
08:          PARLLCK(f_d, bool, num_v),
09:          PARLLCK(c_d, int, num_v) )
11:      TuCCompi_GPUSYN( )
12:      TuCCompi_GPULAUNCH(min,num_v,v_d,a_d,w_d,
16:          PARLLCK(p_d, bool, num_v),
17:          PARLLCK(f_d, bool, num_v),
18:          PARLLCK(c_d, int, num_v) )
20:      TuCCompi_GPUSYN( )
21:      TuCCompi_GPULAUNCH(update,num_v,v_d,a_d,w_d,
25:          PARLLCK(p_d, bool, num_v),
26:          PARLLCK(f_d, bool, num_v),
27:          PARLLCK(c_d, int, num_v) )
29:      TuCCompi_GPUSYN( )
30:    }
31:    user code
32:  }//SSSP_pluginGPU
```

Figure 6.13: TuCCompi pseudo-code for the pluginGPU.

kernel execution.

Additionally, being an embarrassingly parallel problem, it suits perfectly with TuCCompi's approach for the first three layers. Besides, the GPU solution for this problem involves three kernels of very different nature and characterization. This variety allows us to check the behavior of the fourth and tuning layers.

In this section we will explain in detail the corresponding plug-ins for the scheduling policies and the porting of the algorithm to TuCCompi's model.

Scheduling plug-ins

Each SSSP computation is a single independent task. We have slightly modified the naïve master-slave behavior in order to show how easily is to customize the scheduling plug-in (see Fig. 6.11). The master distinguishes the nature of the slave that is requesting a task. Depending on the slaves computational power, the master will send more or less tasks. The TuCCompi model is better exploited if the master gives more tasks to the modern GPUs (Fermi, Kepler and so on) due to their multi-kernel execution feature. This implementation sends CK tasks to each modern GPU, and only one for the Pre-Fermi architectures, or the CPU cores.

The lines 00-22 of Fig. 6.11 show the master implementation, whereas the lines 23-27 show the slave implementation. The master will manage the task distribution while there are task to be executed (lines 01-16). To do so, the master waits for a task request from any slave (line 3). If the slave is a modern GPU (Fermi or Kepler) (line 04), the master checks if there are CK available tasks to be sent. In this case, it sends the identifier of the first task of the pack to the corresponding slave, and updates the task counter (lines 05-07). However, if there are not enough tasks for this type of slave, the master sends to it the termination signal and updates the counter of slaves that have already finished (lines 08-11). If the requesting slave is an old GPU (pre-Fermi) or a CPU core, the master only sends a single task to the slave (lines 12-15), thus, the task counter is simply incremented. When all tasks have been scheduled and carried out, the master sends the termination signal to the rest of active slaves when they request more tasks (lines 17-21).

Regarding the slave implementation, it first notifies the master that it is idle (line 24). Then the slave receives the identifier of the task pack to be executed, one task for CPU cores and Pre-Fermi GPUs, and CK tasks for the modern GPUs in this prototype (line 25).

SSSP plug-ins

Figure 6.12 shows the primitives used for the insertion of the characterization of our SSSP kernels (*relax*, *minimum* and *update*) using our TuCCompi prototype. These characterizations have been obtained through manual inspection of the codes (see Sect 4.5), choosing values in the classification criteria used by TuCCompi, described in Table 6.1. Figure 6.13 shows the adaptation made to the code in order to introduce the deployment primitives of TuCCompi for the pluginGPU.

Small Heterogeneous Cluster (Small HC)			
Node	**CPUInfo**	**#CPU cores**	**GPU details**
pegaso	IC2 i7 960 3.20GHz	8	GeForce GTX 480 + GeForce GTX 680
nodoyuna	IC2 Q8200 2.33GHz	4	-
trasgo/apolo	IC2 Q6600 2.40GHz	4/4	-
geopar	IX E7310 1.6GHz	16	-
patan	IC2 E6550 2.33GHz	2	-
atc01/02	IC2 6300 1.86GHz	2/2	GeForce 9600GT / -
atc03	AMD AtX2 3600+	2	GeForce 8500GT
atc09	IC Q8299 2.33GHz	4	-

Big Heterogeneous Cluster (Big HC): Small HC plus the following machines			
Node	**CPUInfo**	**#CPU cores**	**GPU details**
titan01/02/05	IX E5-2620 2.00GHz	4/4/12+12	-
titan03/04	IX E5645 2.40GHz	8+8/8+8	-
atc05/06	IX E5630 2.53GHz	8+8/4	-
atc07	IX X-5675 3.07GHz	12+12	-
atc08	IX E5-2620 2.00GHz	12+12	-

Table 6.2: Description of the nodes that compound the Heterogeneous Clusters (HCs).

6.6 Experimental Evaluation of TuCCompi Prototype

This section presents the methodology used to test the TuCCompi prototype, the characteristics of the input sets used, and a discussion of the experimental results.

6.6.1 Methodology

We have conducted two experiments with different purposes. The main objective of the first experiment is to test the correct functionality of TuCCompi's ideas, by checking that the joint use of the different layers leads to benefits in terms of performance using our implemented prototype. A sequence of scenarios where each one involves more layers than the previous one is used to prove the scalability of the model in terms of adding more parallelism and coordination levels. The objective of the second experiment is to evaluate the performance gain offered by the two innovative layers introduced by TuCCompi's model not present in other state-of-the-art works, the concurrent-kernel execution and tuning layers.

A description of the platforms and devices used, and the experiments carried out is presented below.

Target architectures

The heterogeneous environment where we have carried out the experiments described in the following sections is compound by several different computing nodes. A description of these cluster components is presented in Table 6.2, where it is indicated the number of CPU cores and GPUs that each one has. In the cluster we can find CPUs ranging from 1.6 GHz to 3.2 GHz, in different kinds of processors, such as Intel Xeon, Intel QuadCore, and AMD. Even the number of computational CPUs that belong to each shared-memory system significantly varies between them. Finally, we can also find computing nodes that are virtual machines running in a virtualization centralized server with several blades, such as the Titan01-05, and the Atc05-08. All these nodes, normal and virtualized, run an Ubuntu Desktop 10.04 operating system.

Three of these machines contains also GPUs. These hardware accelerators are of different architectures (pre-Fermi, Fermi, and Kepler). The first node, pegaso, handles the NVIDIA GPUs GeForce GTX 480 (Fermi GF100) and GeForce GTX680 (Kepler GK104). The nodes atc01 and atc03, handle pre-Fermi boards, GeForce 9600GT and GeForce 8500GT, respectively. All these host machines run the CUDA Toolkit 4.2 and driver 295.41.

The complete heterogeneous cluster contains a total of 180 CPU cores and 4 GPUs. However, each different GPU device is governed by a single CPU core, that is only dedicated to this handling task, not to compute. Thus, they are not counted as CPU computational units. Therefore, the total number of real computational units is 180 (176 CPU cores plus 4 GPUs).

Experiment I - Checking TuCCompi's Layers

In order to evaluate the functionality of the different layers of TuCCompi's model in heterogeneous environments, we have tested the complete APSP problem in different scenarios. The workload scheduling used for the scenarios, described below, is the customized master-slave policy presented in Sect. 6.5. Note that the behavior of the equitable policies, for our heterogeneous scenarios would result in a bottleneck generated at the slowest node, whereas the rest would become idle much earlier.

Each scenario was designed with the aim to check the use of the layers involved in an incremental fashion (see Table 6.2 for architecture details of the machines and devices involved):

1. *A single GPU scenario*, that involves the 3rd, 4th, and the tuning layer. The GPU selected is the GeForce GTX 480 (Fermi architecture). The parameter value for the concurrent kernel execution is set to four, CK=4. This is the only scenario that does not use a scheduling policy because there is only one computational unit.

2. *A multi-GPU scenario*, that involves the 2nd layer in addition to the previous ones (3rd, 4th and tuning layers). We use as slaves the two GPUs present in the shared-memory machine called pegaso (GeForce GTX 480 and GTX 680), and a CPU core doing the master work.

3. *A heterogeneous shared-memory system (complete pegaso)*, with two GPUs and eight CPU cores, in order to test the 2nd layer by mixing two different kinds of computational units. Note that two of the eight CPU cores are responsible of handling the two GPUs, so this scenario handles the execution of 2 GPUs and 6 computing CPU cores.

4. *Small HC*: Small Heterogeneous Cluster, that involves all layers of TuCCompi, using 10 distributed nodes with a total of 48 computational units, 44 computing CPU cores, and 4 GPUs.

5. *Big HC*: Big Heterogeneous Cluster, that also involves all layers of TuCCompi, but with a significant increment of computational resources (180 comprising 176 CPU cores plus 4 GPUs), in order to evaluate the scalability of the model when adding more distributed nodes.

Experiment II - Checking the Innovative 4th and Tunning Layers

This experiment tests the performance gain offered by the joint use of the proposed 4th and Tuning layers. We have compared the execution times of a single GPU connecting or disconnecting the optimizations introduced by these layers. For the non-automatically optimized versions (without 4th and Tuning layers), we have chosen the *default values* offered by TuCCompi. The *default values* selected have proven to be competitive for our case study in several situations of the exhaustive evaluation carried out in the experiment described in Sect. 4.6. This *default configuration* involves the use of 256 threads per block, a increased L1 cache memory size, and no concurrent kernel execution.

The experiments have been carried out just computing enough task sets (1 024, 2 048, 4 096, 8 102, 16 204, and 32 408) to produce sufficient computational load to keep scalability.

6.6.2 Input Set Characteristics

The used input set is composed by the collection of Martín's synthetic graphs, described in Sect. 3.3.2. The graph generation method used to create these input sets leads to irregular loads when applying individual SSSP searches. We have used this kind of graph instead of others with an uniform distribution because we will also check in this way the specific master-slave policy implemented for the study case. For the first experiment, we have used four different graph-sizes, whose number of vertices are 1 049 088, 1 509 888, 2 001 408 and 2 539 008. For the second experiment, the used graphs were the ones with 2 539 008 nodes.

6.6.3 Experimental Results I - Checking TuCCompi's Layers

Figure 6.14 shows the execution times obtained for the single GPU, the multi-GPU system, the heterogeneous shared-memory host, and the two heterogeneous cluster scenarios. Although the GPUs are the most powerful devices, and their combined use has returned a very competitive ratio *speedup/computational unit* difficult to beat (1.98× speedup), the

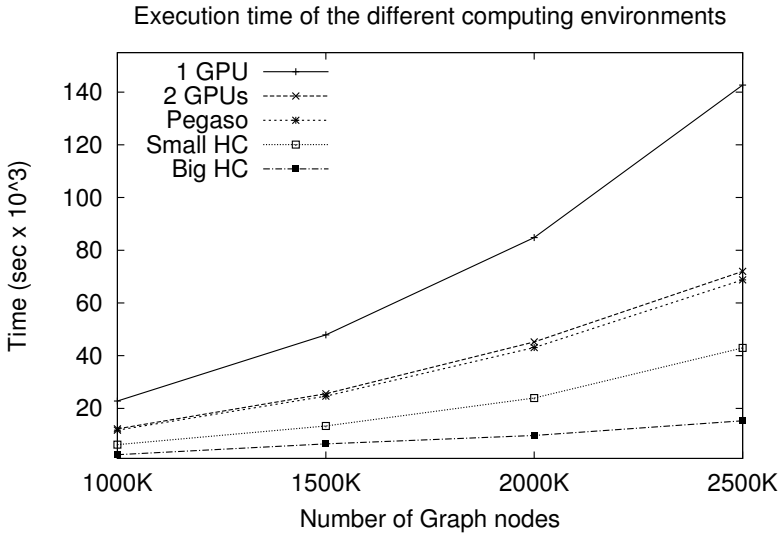


Figure 6.14: Execution times of the tested scenarios for different graph-sizes.

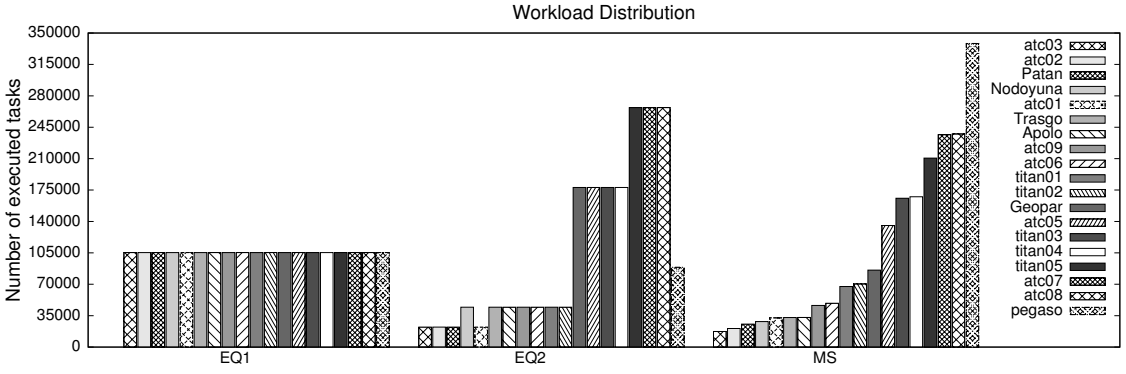


Figure 6.15: Number of executed tasks per node of the Big HC with the three scheduling policies.

advantage of using many less-powerful computational units has enhanced even more the total performance gain (a speedup up to 3.7× in the Small HC, and a up to 9.4× in the Big HC), compared with the single GPU scenario.

Therefore, the addition of more computing resources has always reduced the global execution time needed to compute the complete APSP. This is possible because the communication overhead across nodes of TuCCompi prototype was lower than 1% of the total execution time. In particular, the overhead in the Small-HC platform, has never surpassed 0.589% of the total execution time.

Figure 6.15 shows the experimental distribution of tasks per cluster node using the `MS` scheduling policy, compared with the theoretical values that `EQ1` and `EQ2` static policies would obtain. We can observe how the `MS` dynamic scheduling policy favors a proportioned distribution of the task according to the processing capabilities of the devices, for the used study case.

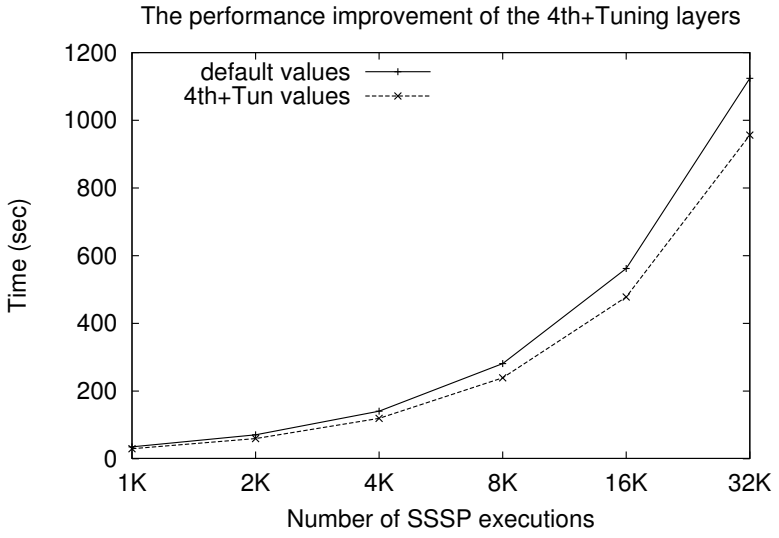


Figure 6.16: Performance improvements obtained by the automatic 4th and Tuning layers, with respect to the competitive default configuration values.

6.6.4 Experimental Results II - The Innovative 4th + Tuning Layers

Figure 6.16 shows the comparison of the concurrent kernel execution, with CK=4, combined with the values proposed in Chapter. 4, with respect to the competitive *default configuration* on the GPU GeForce GTX 480. The use of the concurrent kernel layer and the optimization tuning reduces the execution time for our test case up to 12%.

6.7 Conclusions

TuCCompi is a multilayer abstract model that helps the programmer to easily obtain flexible and portable programs that automatically detect at run-time the available computational resources and exploits hybrid clusters with heterogeneous devices. This model offers to the programmer a transparent and useful mechanism to select the optimal values of GPU configuration parameters just characterizing the nature of the kernels. Any parallel application that can be devised as a collection of non-dependent tasks working on shared data-structures can be exploited with the TuCCompi model.

Compared with previous works, TuCCompi adds a novel parallel layer to the traditional parallel ones, with the automatic execution of concurrent kernels in a single GPU. Additionally, it squeezes even more the computational power of the GPUs by applying optimal values for runtime configuration parameters, such as the threadblock size, and the management of the L1 cache memory state. For our test case, the use of these both new layers leads to performance improvements of up to 12%. Thus, these new layers turn out to be very convenient for exploiting heterogeneous clusters with GPUs.

The model is designed to provide a mechanism of plug-ins, in order to easily change: (1) The algorithms to be deployed; (2) the scheduling policies of the tasks; and (3) the parameter values for optimal configurations on different GPU architectures, without making any change in the model. The use of this model takes advantage of even the less powerful

devices of a heterogeneous cluster, and it correctly scales if more computational units are added to the environment, with a communication overhead less than one percent of the total execution time.

The work and conclusions described in this chapter have been published in the following papers:

- "TuCCompi: A Multi-Layer Model for Distributed Heterogeneous Computing with Tuning Capabilities," H. Ortega-Arranz, Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos, *International Journal of Parallel Programming,* p. 1–22, 2015.
  Online, DOI: 10.1007/s10766-015-0349-6

- "TuCCompi: A Multi-Layer Programing Model for Heterogeneous Systems with Auto-Tuning Capabilities," H. Ortega-Arranz, Y. Torres, D. R. Llanos, and A. Gonzalez-Escribano, in *Proceedings of Workshop on High-level Programming for Heterogeneous and Hierarchical Parallel Systems*, ser.(HLPGPU'14), Vienna, Austria: HiPEAC 2014, pp. 18-25.
  Online: HLPGPU Proceedings

Chapter 7

Conclusions

The developing of solutions for computing shortest path distances has always been an interesting issue for the scientific community, due to its wide applicability to numerous real-world fields. During the course of the history, significant improvements have been made to this solutions by using data structures designed for graphs with particular features, and thus, creating efficient algorithms for specific applications.

The incorporation of new parallel programming models together with modern powerful hardware accelerators to path-finding solutions has open the possibility of creating more study new and more efficient parallel approaches. In some cases, the parallelized solutions of general naïve algorithms outperforms the optimized particular sequential ones. Therefore, the parallel improvements made to these general algorithms may lead to better performance for the general case.

Furthermore, the emerging of heterogeneous parallel computing combining the powerful hardware accelerators with the classical and increasingly powerful CPUs, provides a perfect environment to face the most costly shortest-path problems in the context of High Performance Computing (HPC). However, the programming of hardware accelerators, the optimization of their running times, and also, the coordination of these devices with other computational units of different nature, are still very complex tasks for non-expert programmers.

This Ph.D. thesis has addressed both mentioned problems, the algorithmic GPU programming and the heterogeneous parallel coordination in the context of: Developing new GPU-based approaches to the shortest path problem; the study of the tuning of the GPU configuration parameters; and also, designing solutions where both sequential and parallel algorithms are deployed concurrently in heterogeneous environments.

7.1 Answer to the Research Question

Is it possible to develop techniques and tools to derive more efficient parallel implementations to solve Shortest Path problems using: (1) The new modern Graphics Processor Units (GPUs) and their corresponding optimization tuning techniques, (2) heterogeneous environments composed by these hardware accelerators together with the use of traditional CPUs?

The work presented in this Ph.D. thesis allow us to conclude that both research questions have an affirmative response.

(1) We were able to develop a new parallel solution for GPUs that takes advantage of their powerful capabilities, following some ideas proposed by Crauser *et al.* [24], found through our study of the literature about Shortest Path algorithms. Our implementation improves the performance of the previous state-of-the-art due to Martín *et al.* [23]. Following the guidelines proposed in [28], we have proposed a refined method to systematically obtain good GPU configuration parameters in terms of GPU code characteristics. The application of this methodology has led to performance improvements of our shortest paths program solutions compared with the use of configuration parameter values suggested by CUDA programming guidelines [21].

(2) Combining different parallel programming models and languages, we were able to develop implementations and novel studies for some formulations, yet unexplored, inside the parallel productivity-based solutions of All-Pair Shortest-Path problem. These implementations include the following approaches. Using the new concurrent-kernel feature of modern GPUs, to concurrently execute independent streams of kernels, solving several SSSP subproblems in parallel. Systematical selection of GPU configuration parameters by modeling simple kernel code features. And combining CUDA and OpenMP, exploiting both GPUs and CPU cores present in the same shared-memory system, testing different load-balancing techniques. Finally, we have proposed a multilayer programming model, developing a prototype, where the described approaches are combined to transparently coordinate the use of heterogeneous clusters, internally using several technologies such as CUDA, OpenMP, and MPI. All the experiments carried out for these novel parallel heterogeneous formulations showed that the new combined approach leads to significant speedups when compared to parallel homogeneous baseline versions.

7.2 Summary of Contributions

This section summarizes the contributions of this Ph.D. thesis and the related papers that have been published. The contributions include both surveys related to our study of the state of the art, and results obtained during the development of the subgoals proposed in Sect 1.2.2 (see Fig. 7.1).

7.2.1 Surveys and classification studies for the algorithms involved in Shortest Path problems

We have reviewed both sequential and parallel approaches that solve two different shortest-path problems: The Single-Source Shortest-Path problem, and the All-Pair Shortest-Path problem. The Single-Source Shortest-Path problem addresses the computation of shortest paths, and their distances, between a source node of the graph and the remaining vertices

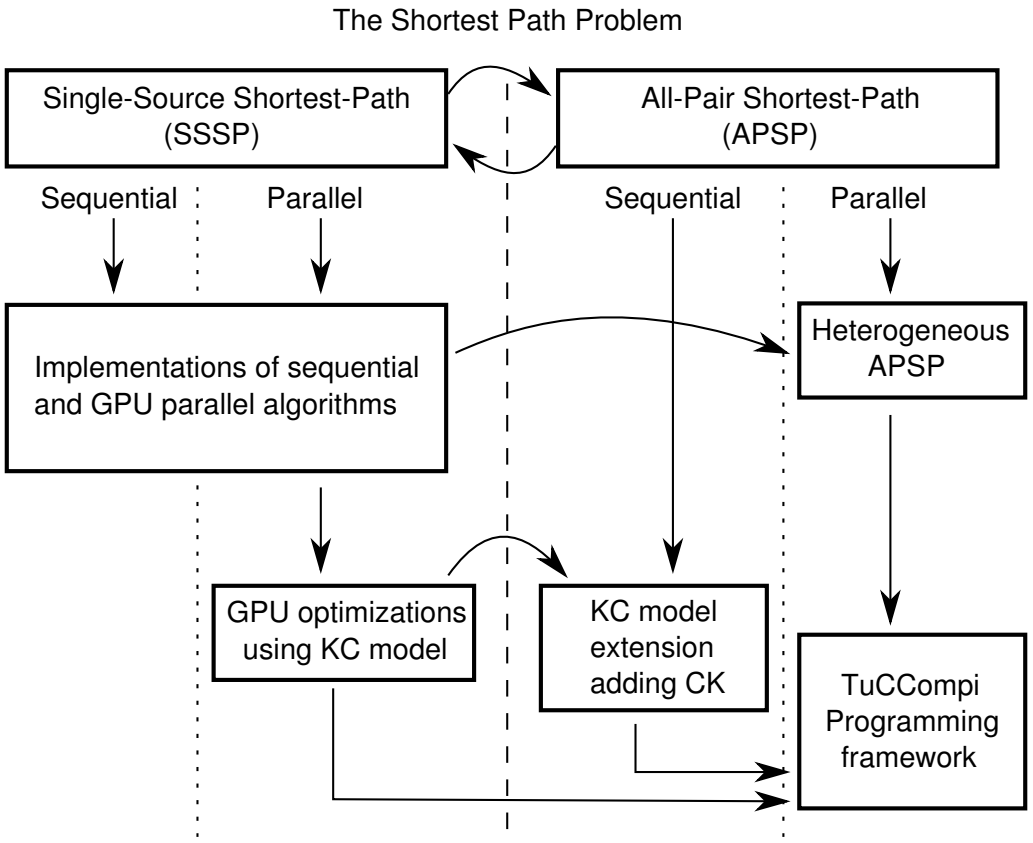


Figure 7.1: Subgoals accomplished in this Ph.D. thesis.

of the graph. The All-Pair Shortest-Path problem addresses the computation of shortest paths between all possible pairs of nodes in the graph.

We have presented new classifications for the parallel approaches according to their features, and we have located some unexplored solutions that have been addressed in this Ph.D. thesis. Additionally, it has been also studied a particular real-world application where the use of parallel computing is required to maintain updated valuable precomputed data.

These contributions have been published in the following papers:

1. "Parallel Approaches to the Shortest Path Problem - A Survey," H. Ortega-Arranz, Y. Torres, D. R. Llanos, and A. Gonzalez-Escribano, *To be submitted to ACM Computing Surveys*

2. "The Shortest Path Problem: Analysis and Comparison of Methods," H. Ortega-Arranz, D. R. Llanos, A. Gonzalez-Escribano, *Book*, 1st edition, ser.(Synthesis Lectures on Theoretical Computer Science series), Morgan & Claypool. Online, DOI: 0.2200/S00618ED1V01Y201412TCS001 [20]

7.2.2 Development of a new GPU-based algorithm outperforming a previous state-of-the-art GPU SSSP solution

We have developed a new parallel approach for GPUs to solve the SSSP problem. A previous state-of-the-art solution, due to Martín *et al.* [23], has been outperformed in all graph families tested, including their graph suite used in their own experimentation, a wide variety of synthetic random graphs, and real-world application networks. Comparing our implementation with Martín's one, we obtained speedups of up to 45×, and even 130× for a particular graph. The values used for the runtime execution parameters were exactly the same as the ones used by Martín *et al.* for a fair comparison.

We have tuned our GPU implementation using values for the runtime parameters which are specifically appropriate for NVIDIA devices, obtaining up to 22.43% performance gain when compared to the non-tuned version. This non-tuned version uses the same values for GPU configuration parameters chosen by Martín *et al.* in their study.

We have compared our tuned implementation with the optimized sequential implementation of the Boost library [25], obtaining significant speedups of up to 19× for some of the tested graph families, and also consuming up to 11.25 less memory.

We have made an architectural study of different modern GPU architectures of NVIDIA devices. The results suggest that the use of a particular NVIDIA architecture for graphs with specific features leads to better execution times compared with other architectures. Fermi GF100 architecture, with a lower number of computing cores but with higher clock frequencies, obtains better execution times with a difference of up to 39.4% with both Kepler's architectures tested, when computing graphs with low number of nodes and low mean fan-out degree. On the other hand, as the values of these two features increase, Kepler GK110B architecture obtains better execution times than Fermi's GF100 with a total difference of up to 40.5%.

The work described has been published in the following papers:

3. "Comprehensive Evaluation of a New GPU-based Approach to the Shortest Path Problem," H. Ortega-Arranz, Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos, *International Journal of Parallel Programming,* Springer, p. 1–21, 2015.
   Online, DOI: 10.1007/s10766-015-0351-z [26]

4. "A New GPU-based Approach to the Shortest Path Problem," H. Ortega-Arranz, Y. Torres, D. R. Llanos, and A. Gonzalez-Escribano, in *Proceedings of the 11th International Conference on High Performance Computing and Simulation*, ser.(HPCS'13), Helsinky, Finland: IEEE, 2013, pp. 505–511.
   Online, DOI: 10.1109/HPCSim.2013.6641461 [27].

7.2.3 Extension of the kernel characterization model

We have extended an existing model, that predicts proper values for NVIDIA GPU runtime parameters only using information related to the characterization of the GPU kernels. The extensions include considering information related to application-dependent parameters of input graphs, such as the number of nodes, or the mean fan-out degree. Depending

on these values the behavior of the GPU kernels, and their characterization, can change leading to different proper values for an optimal performance.

We have checked the validity of the model by performing an exhaustive search in the solution space evaluating the most relevant values for the CUDA configuration parameters, and graph features, with a positive result.

We have also measured the usefulness of the values predicted by the model. We have calculated the maximum gap between an optimized configuration, applying the knowledge behind the kernel characterization model, and a naïve configuration obtained by following the recommendations of CUDA guidelines. There is a total performance gain of up to 62% for our case study.

We have also studied the influence of using different numbers of concurrent kernels on the rest of studied parameters. The results suggest that the use of these techniques does not affect the current predictions of the model.

In order to conduct the complete study, we have adapted our GPU solution to the $n \times SSSP$ approach in order to solve the APSP problem, using the new concurrent-kernel feature of the modern NVIDIA GPUs.

The work described has been published in the following papers:

5. "Optimizing an APSP Implementation for NVIDIA GPUs Using Kernel Characterization Criteria", H. Ortega-Arranz, Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos, *The Journal of Supercomputing,* Springer, vol. 70, no. 2, pp. 786-798, 2014.
   Online, DOI: 10.1007/s11227-014-1212-z [28]

6. "A Tuned, Concurrent-Kernel Approach to Speed Up the APSP Problem," H. Ortega-Arranz, Y. Torres, D. R. Llanos, and A. Gonzalez-Escribano, in *Proceedings of the 13th International Conference on Computational and Mathematical Methods in Science and Engineering*, ser.(CMMSE'13), Almería, Spain: eds. I.P. Hamilton and J. Vigo-Aguilar, 2013, vol. 4, pp. 1114-1125.
   Online: CMMSE Proceedings [29]

7.2.4 Studies of novel heterogeneous approaches for the APSP problem

We have implemented a heterogeneous $n \times SSSP$ approach to solve the APSP problem, combining traditional CPU cores with modern GPUs in the same shared-memory system, and using two different load-balancing techniques, equitable scheduling and work-queue retrieving scheduling.

We have studied the relevance of this approach, when computing the APSP problem in two different situations with irregular graphs. We have compared the execution times of several instances of our heterogeneous implementation against the execution times of using just a single GPU. Performance gains of up to 65% are obtained when computing the complete APSP problem, and up to 47% when computing random APSP subproblems.

We have also concluded that a previous study of the nature of the input problem is very important, because it allows to the programmer to better map the most costly tasks

to the most powerful devices. The equitable scheduling can be tuned up using information of input graph characteristics to achieve better performances. The results are very sensible to changes in the input graph. The work-queue retrieving implementations have a more stable performance behavior than the equitable scheduling because they are more independent from the graph nature.

The work done has been published in the following paper:

7. "The All-Pair Shortest-Path Problem in Shared-Memory Heterogeneous Systems," H. Ortega-Arranz, Y. Torres, D. R. Llanos and A. Gonzalez-Escribano, in book *High-Performance Computing on Complex Environments,* ser. Series on Parallel and Distributed Computing. John Wiley & Sons, Inc., 2014, ch. 15, pp. 283-299. Online, DOI: 10.1002/9781118711897.ch15 [30]

7.2.5 Development of a multilayer programming model: TuCCompi

We have developed TuCCompi, a multilayer abstract model that helps the programmer to easily obtain flexible and portable programs that automatically detect at run-time the available computational resources and exploit hybrid clusters with heterogeneous devices. This model offers to the programmer a transparent and useful mechanism to select the optimal values of GPU configuration parameters just characterizing the nature of the kernels.

We have included in the design of TuCCompi's model: (1) A novel parallel layer that exploits the automatic execution of concurrent kernels in a single GPU; and (2) automatic tuning techniques, with the application of proper values for runtime configuration parameters depending on the kernel characterization provided by the programmer.

We have implemented a prototype of the model, and we have adapted our APSP solution in order to check the functionality of the programming framework.

We have also evaluated the usefulness of the novel layers in a competitive situation. The performance obtained by executing the program using the novel layers and tuning techniques are compared with the performance of the same program using the competitive values delivered by TuCCompi's default configuration.

These new layers turn out to be very convenient for heterogeneous clusters with GPUs offering a performance gain of up to 12%.

The work done has been published in the following papers:

8. "TuCCompi: A Multi-Layer Model for Distributed Heterogeneous Computing with Tuning Capabilities," H. Ortega-Arranz, Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos, *International Journal of Parallel Programming,* Springer, p. 1–22, 2015. Online, DOI: 10.1007/s10766-015-0349-6 [31]

9. "TuCCompi: A Multi-Layer Programing Model for Heterogeneous Systems with Auto-Tuning Capabilities," H. Ortega-Arranz, Y. Torres, D. R. Llanos, and A. Gonzalez-Escribano, in *Proceedings of Workshop on High-level Programming for Heterogeneous and Hierarchical Parallel Systems*, ser.(HLPGPU'14), Vienna, Austria: HiPEAC 2014, pp. 18-25.
Online: HLPGPU Proceedings [32]

7.3 Future Directions

The complete picture of the Shortest Path context is an enormous world with still some corners unexplored, that with the advent of new parallel approaches and new modern devices, becomes even bigger and unexpected. New domains of application arise presenting specific features for which the optimal approach is not yet studied. Finally, the developing of new features in modern hardware accelerators, with new parallel computing capabilities, also opens a new transversal dimension in the search for optimality. The current best-performance algorithms may be modified for a particular computational device in order to take advantage of its optimized internal mechanisms.

Therefore, this work lets the door open to some issues which define the future directions for this research:

Shortest Path context

- Algorithm modifications to take advantage of the new parallel capabilities of the GPUs, or for the emerging XeonPhi devices.
- A comparison with other parallel solutions implemented with other parallel programming models, such as OpenMP or MPI, in order to see what are the thresholds for each kind of solution where their performance is optimal depending on the graph characteristics.

GPU Tuning context

- An extension of the value-predicting model for other more complex characteristics of the input graphs, such as the diameter or betweenness centrality.
- The adaptation of existing code analyzers in order to automatically obtain the kernel characterizations needed to apply our prediction values.

TuCCompi context

- The addition of an optional auto-tuning behavior that allows to find the optimal number of concurrent kernels to be deployed during the execution.
- The implementation and testing of new scheduling plug-ins for new kinds of applications, also including problems with data-dependencies, and for specific data partition and data distribution schemes, needed in problems with larger input data sets.

Appendix A

Graphical Results from the Exhaustive Search for GPU Optimal Parameters Values

The exhaustive search for proper/optimal values of the GPU tuning parameters, carried out in the experimentation shown in Sect. 4.6, resulted in many results that we analyzed. The high amount of data is due to in the experimentation was taken into account not only the corresponding GPU tuning techniques and features, such as the use of different Threadblock Sizes (TS), the management of the L1 cache memory (L1), or the number of Concurrent Kernels that were executed (CK), but also some characteristics of the input set graphs, such as the number of vertices, or the mean of the fan-out degree of these vertices. All the obtained results represents running times for the different kernels that belongs to our used SSSP solution, presented in Sect. 3.2. Therefore, the proper values for the GPU optimal configurations are the ones that represent the minimum (MIN) running time between all tested layouts.

This appendix shows the raw data resulted from this experimentation is formatted following a compressed graphical fashion that allows us to compare different dimensions at once. Table A.1 shows the links to the different eighteen figures created representing the nine graph scenarios considered for both GPU architectures, Fermi and Kepler. Note that a graph scenario is composed by a set of graph with the same characteristics: number of nodes of the graph, and fan-out degree mean.

	FERMI			KEPLER		
	24k	49k	98k	24k	49k	98k
d2	A.1	A.4	A.7	A.10	A.13	A.16
d20	A.2	A.5	A.8	A.11	A.14	A.17
d200	A.3	A.6	A.9	A.12	A.15	A.18

Table A.1: Figure links for all graph scenarios considered.

These graphics are organized as follows:

- A graphic is depicted for each kernel executed using a particular graph set (with a specific number of nodes and fan-out degree). Inside this graphic:
  - **(L1)** The results are grouped in three coupled frames depending on the configuration of the L1 cache memory (Increased L1, Normal configuration, or Deactivated).
  - **(CK)** For a particular threadblock size used, a curve is depicted depending on the number of concurrent kernels executed, from 1 kernel to 32 kernels following a exponential fashion power of 2.
  - **(TS)** As eight different threadblock sizes were tested (96, 128, 192, 256, 384, 512, 768, and 1024), eight lines with different colors are depicted in each different L1-configuration frame.
  - **(MIN)** A line is depicted for the minimum running time inside each frame, with the same color that the threadblock size used. Each line crosses all three frames in order to ease the comparison of the minimums, being the lowest line of the complete graphic the one that represents the configuration with the fastest running times, for the kernel in a particular graph set.
- The three different kernels (relax, minimum, and update), using the same graph input set, are grouped in the same page.
- Each different key depicted for each state of the L1 cache memory is the same for the remaining frames with equal L1 state configuration. The key is removed from these frames in order to show a clean graphic.
- Finally, all the graphics are sorted following the criterion of the size of the graph sets (24k, 49k, and 98k), and inside each of them, following the fan-out degree mean (d2, d20, and d200), resulting in a total of nine different pages of results for each board evaluated.
- The first set of graphics belongs to the results of the Fermi GF100 architecture, whereas the last set belongs to the Kepler GK104 architecture.

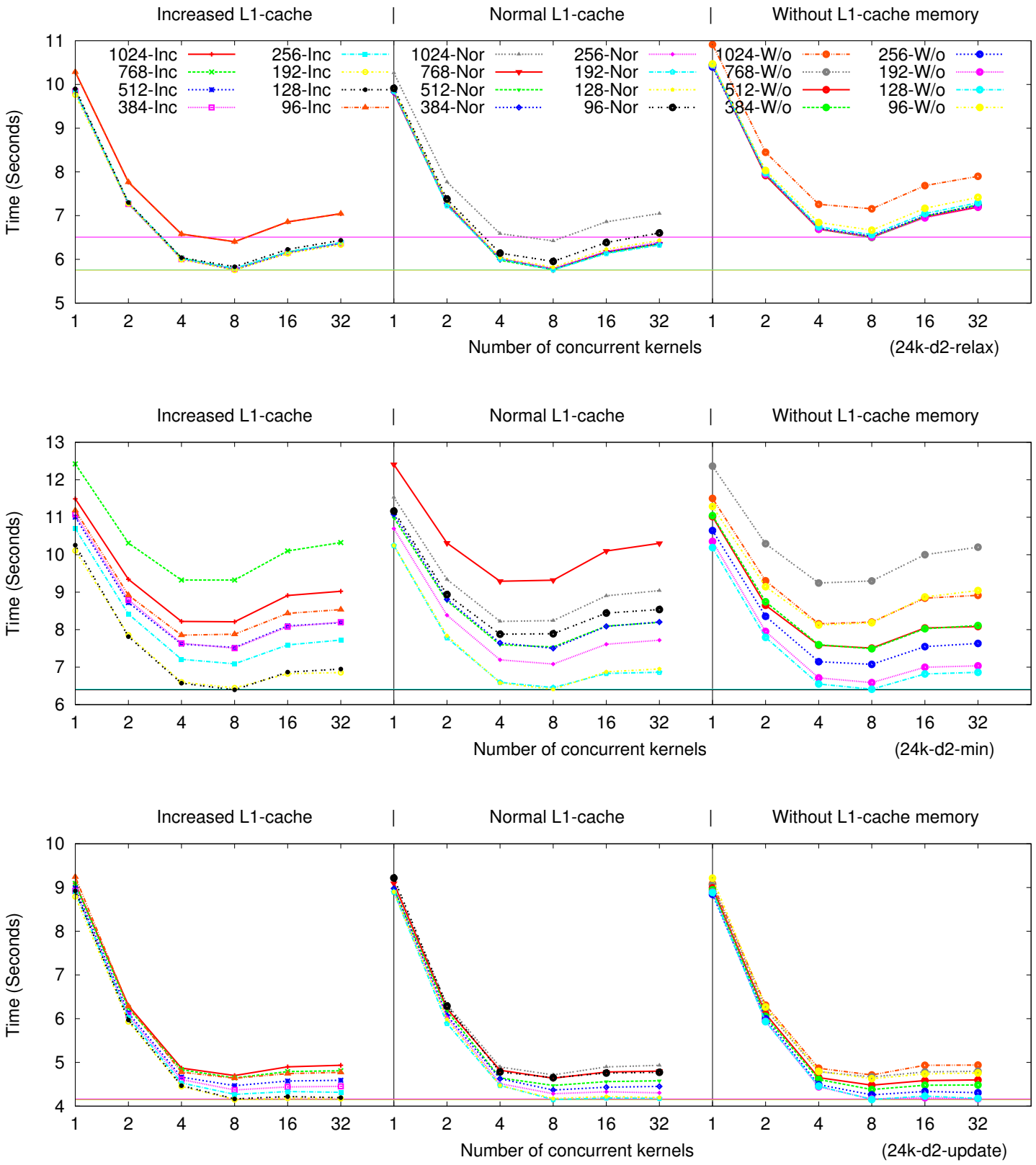


Figure A.1: Exhaustive search for optimal values in the graph **24k-d2** scenario for the **Fermi GF100** architecture. Different threadblock sizes, managements of the L1-cache, and number of concurrent kernels have been evaluated for the *relax kernel (top)*, *minimum kernel (middle)*, and *update kernel (bottom)*.

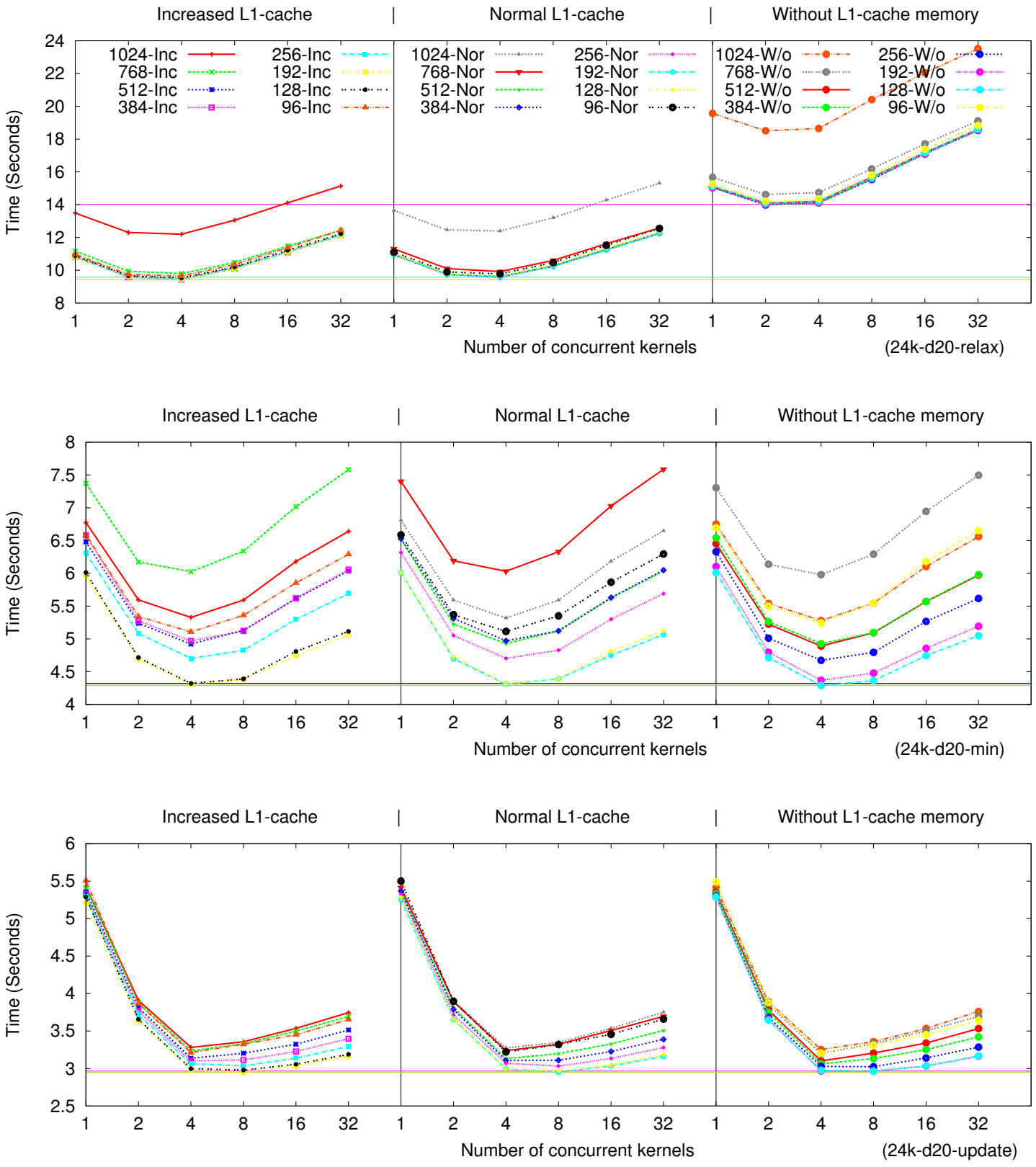


Figure A.2: Exhaustive search for optimal values in the graph **24k-d20** scenario for the **Fermi GF100** architecture. Different threadblock sizes, managements of the L1-cache, and number of concurrent kernels have been evaluated for the *relax kernel (top)*, *minimum kernel (middle)*, and *update kernel (bottom)*.

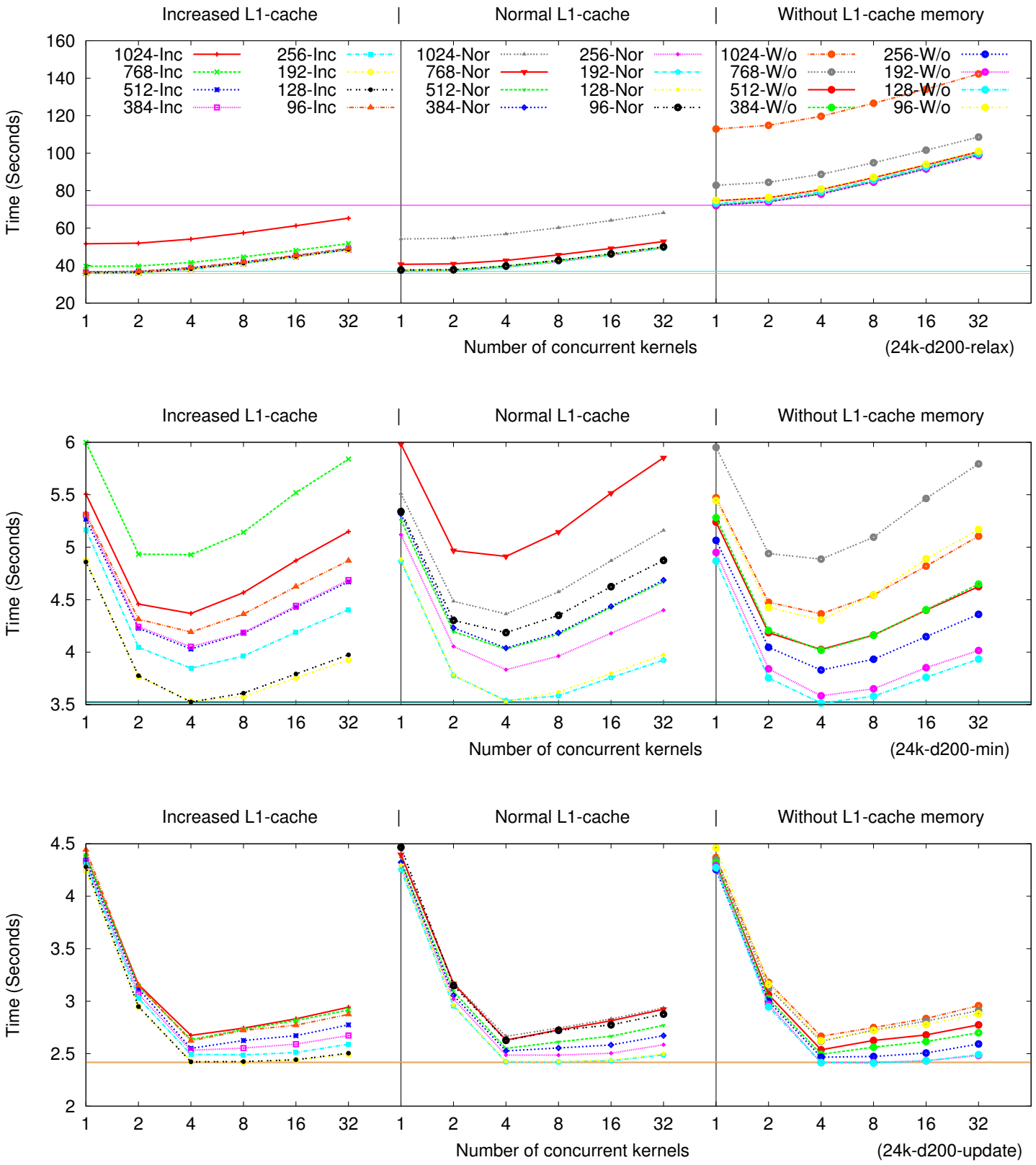


Figure A.3: Exhaustive search for optimal values in the graph **24k-d200** scenario for the **Fermi GF100** architecture. Different threadblock sizes, managements of the L1-cache, and number of concurrent kernels have been evaluated for the *relax kernel (top)*, *minimum kernel (middle)*, and *update kernel (bottom)*.

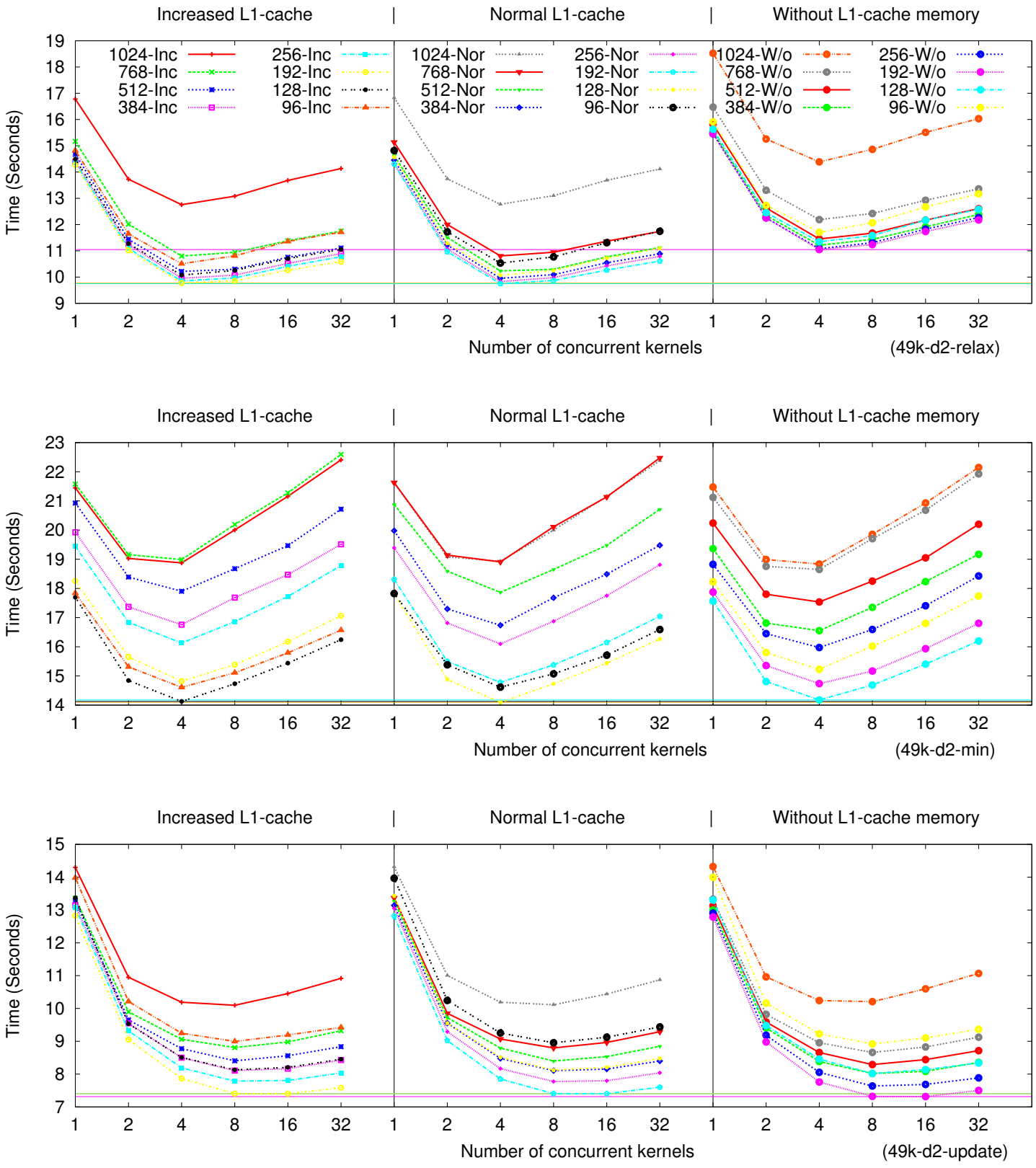


Figure A.4: Exhaustive search for optimal values in the graph **49k-d2** scenario for the **Fermi GF100** architecture. Different threadblock sizes, managements of the L1-cache, and number of concurrent kernels have been evaluated for the *relax kernel (top)*, *minimum kernel (middle)*, and *update kernel (bottom)*.

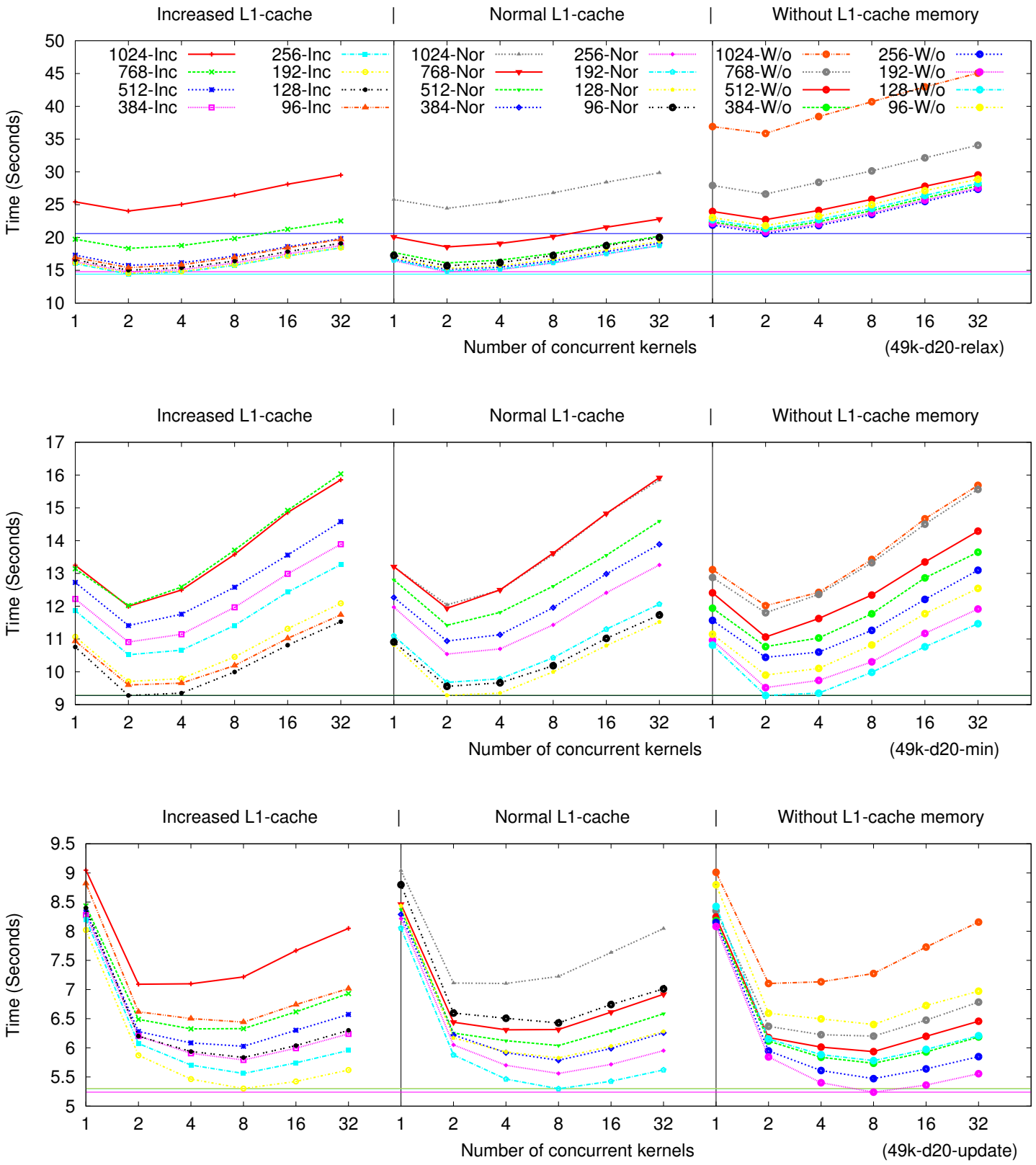


Figure A.5: Exhaustive search for optimal values in the graph **49k-d20** scenario for the **Fermi GF100** architecture. Different threadblock sizes, managements of the L1-cache, and number of concurrent kernels have been evaluated for the *relax kernel (top)*, *minimum kernel (middle)*, and *update kernel (bottom)*.

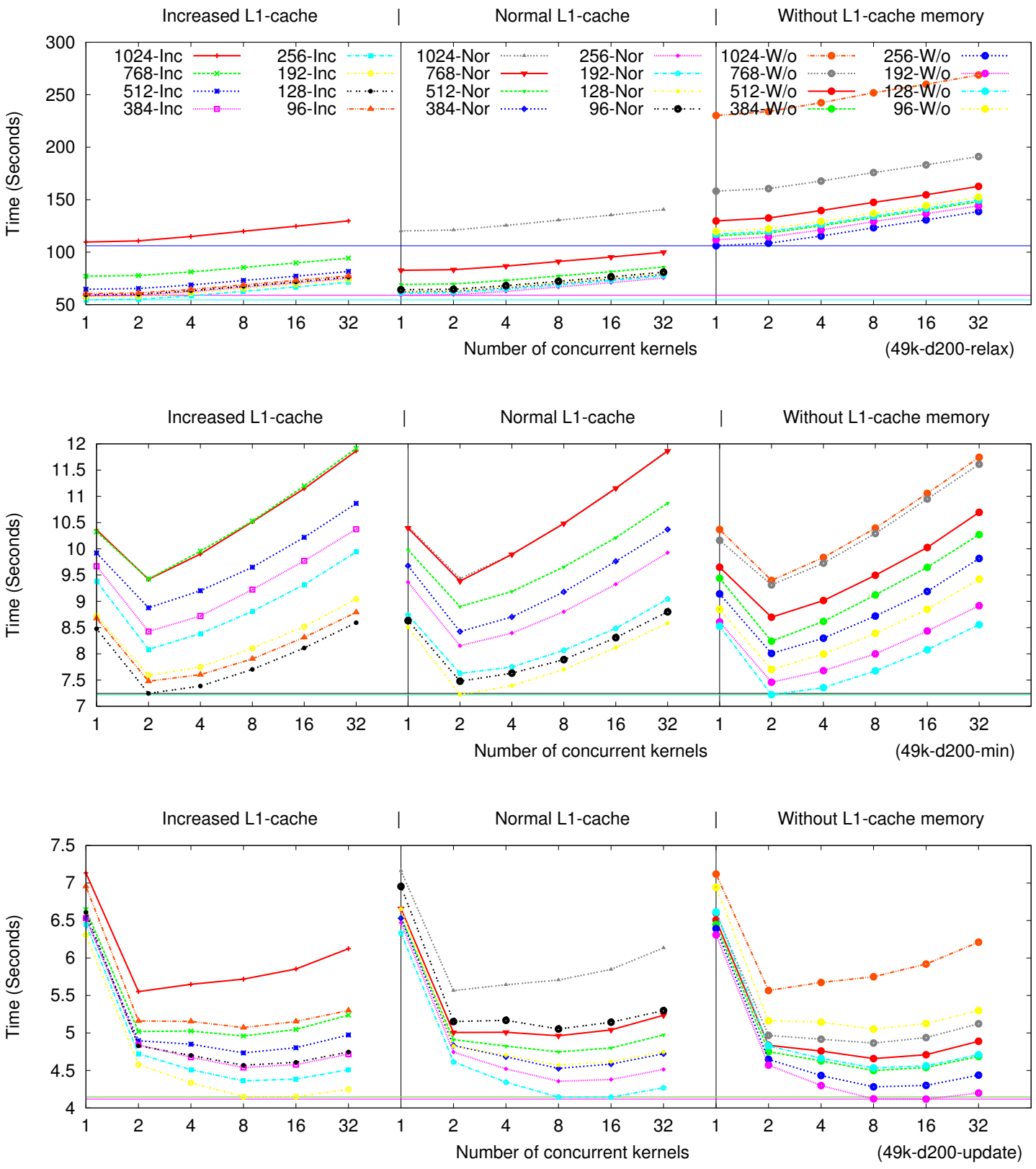


Figure A.6: Exhaustive search for optimal values in the graph **49k-d200** scenario for the **Fermi GF100** architecture. Different threadblock sizes, managements of the L1-cache, and number of concurrent kernels have been evaluated for the *relax kernel (top)*, *minimum kernel (middle)*, and *update kernel (bottom)*.

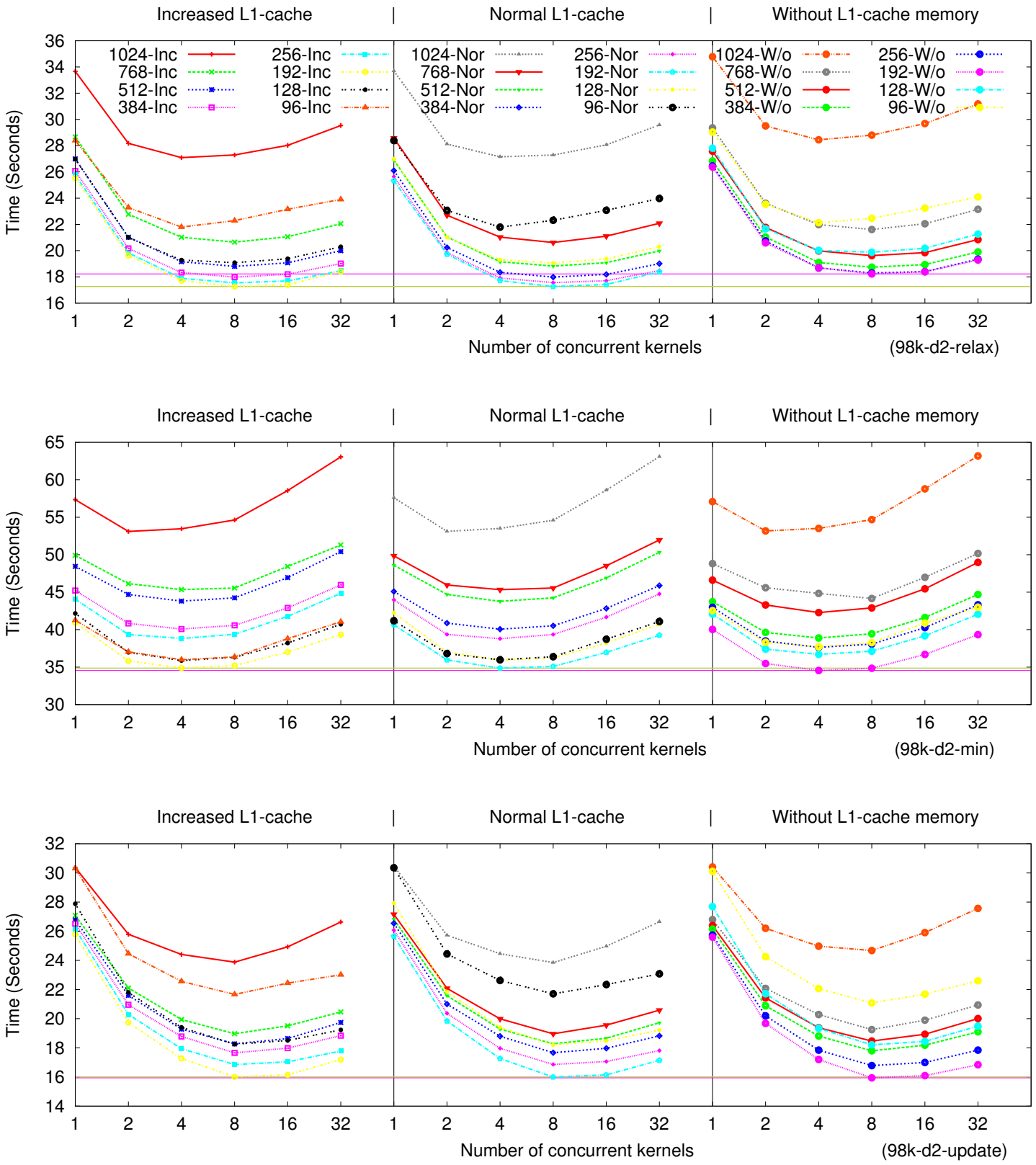


Figure A.7: Exhaustive search for optimal values in the graph **98k-d2** scenario for the **Fermi GF100** architecture. Different threadblock sizes, managements of the L1-cache, and number of concurrent kernels have been evaluated for the *relax kernel (top)*, *minimum kernel (middle)*, and *update kernel (bottom)*.

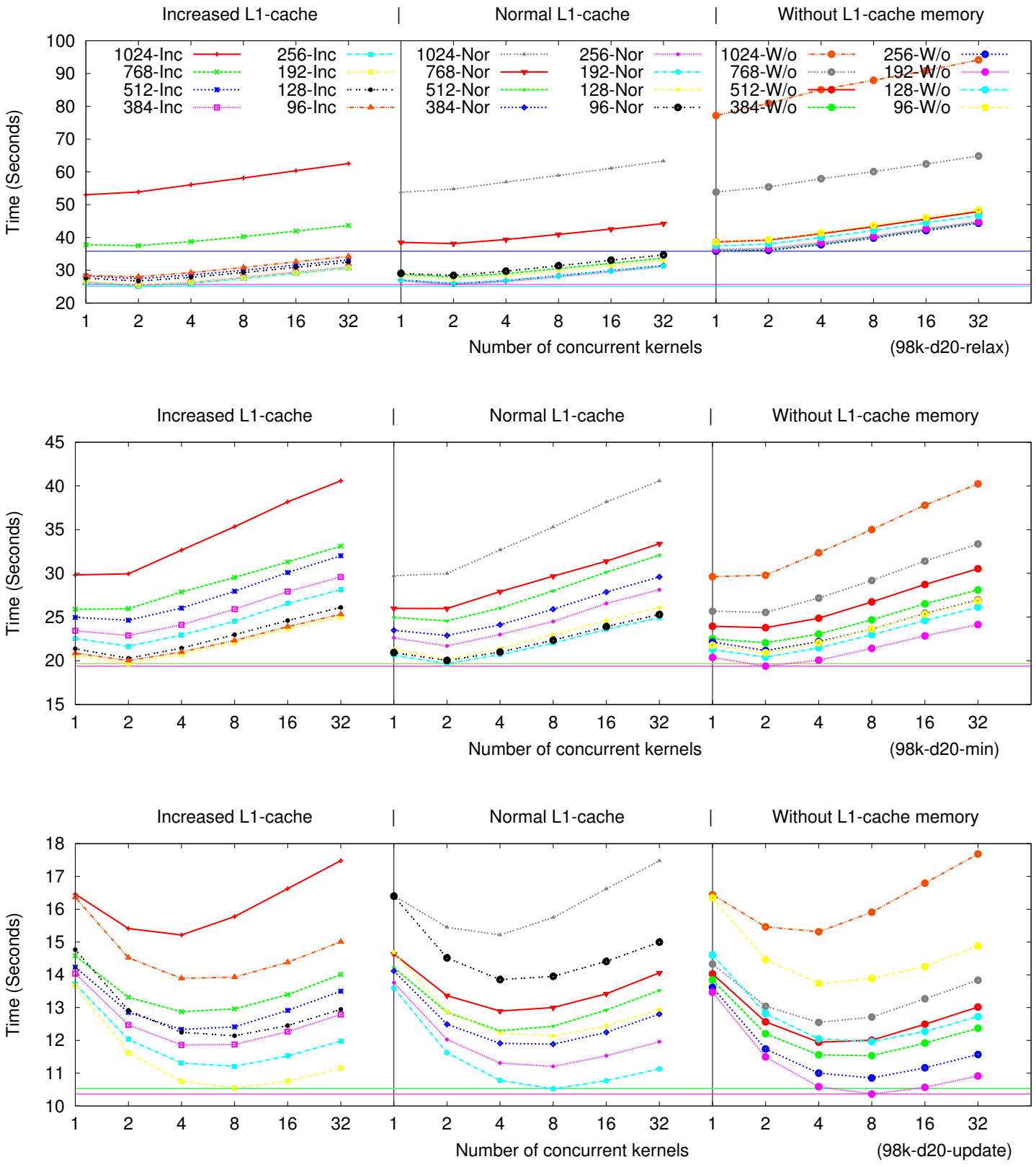


Figure A.8: Exhaustive search for optimal values in the graph **98k-d20** scenario for the **Fermi GF100** architecture. Different threadblock sizes, managements of the L1-cache, and number of concurrent kernels have been evaluated for the *relax kernel (top)*, *minimum kernel (middle)*, and *update kernel (bottom)*.

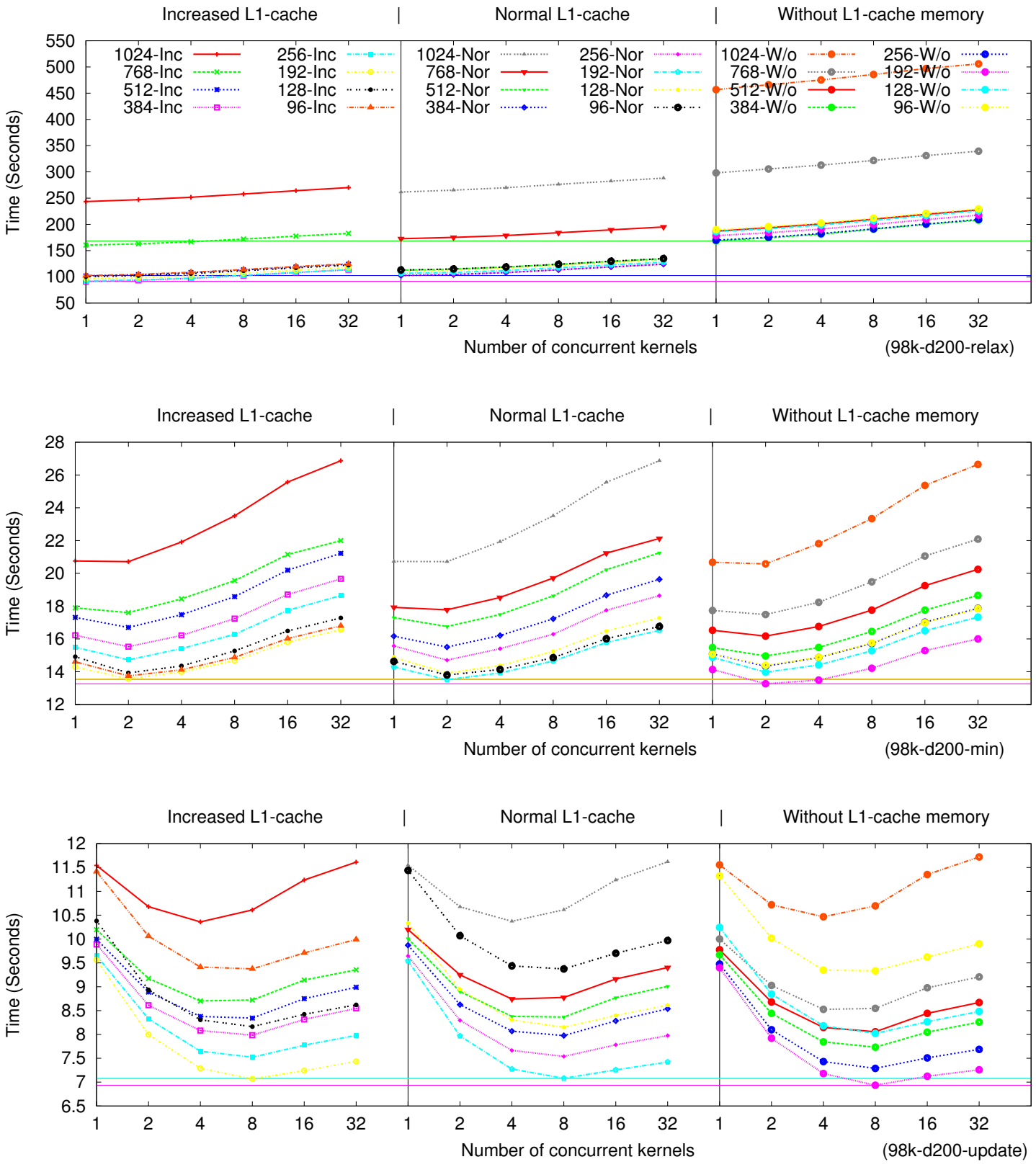


Figure A.9: Exhaustive search for optimal values in the graph **98k-d200** scenario for the **Fermi GF100** architecture. Different threadblock sizes, managements of the L1-cache, and number of concurrent kernels have been evaluated for the *relax kernel (top)*, *minimum kernel (middle)*, and *update kernel (bottom)*.

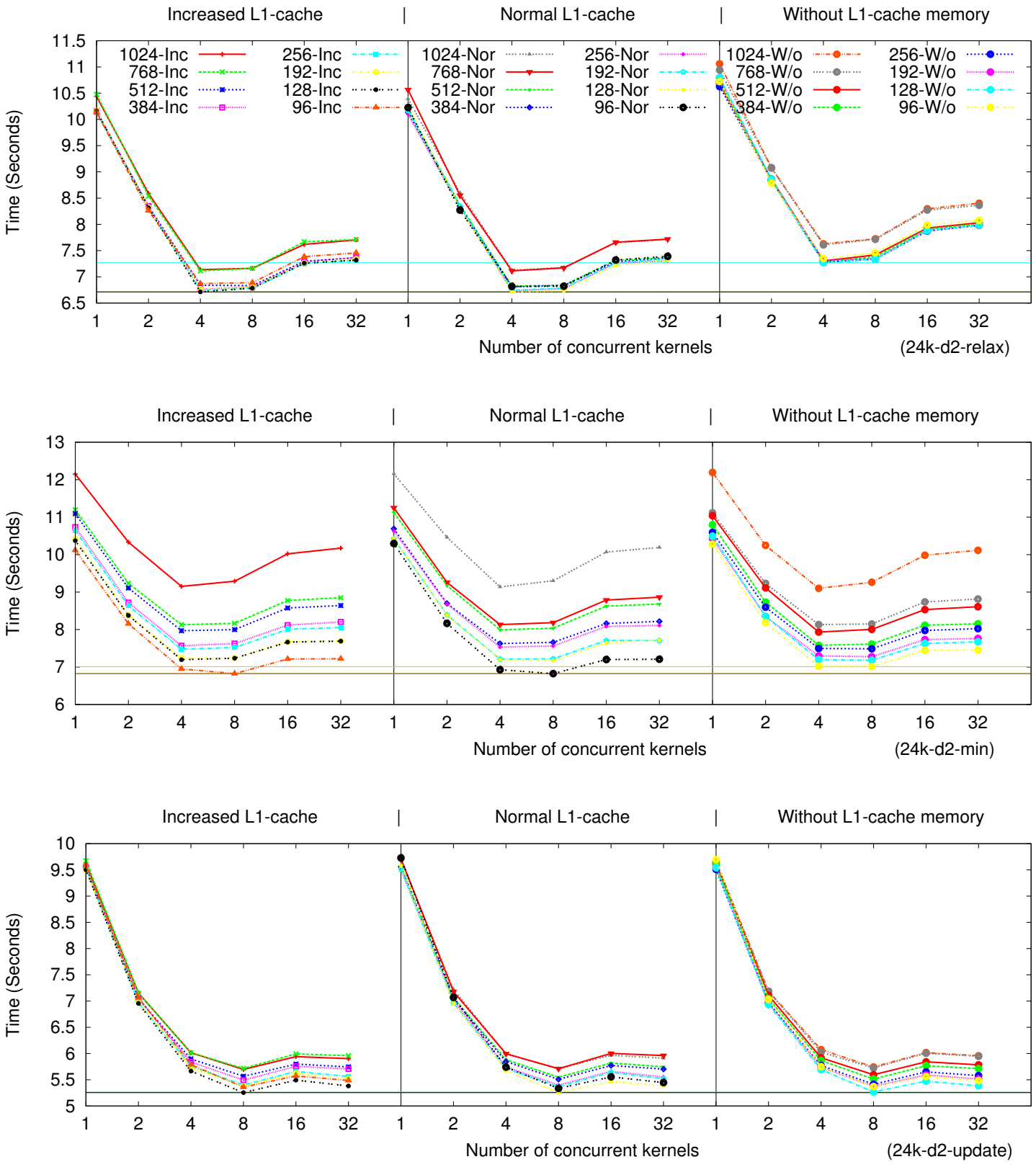


Figure A.10: Exhaustive search for optimal values in the graph **24k-d2** scenario for the **Kepler GK104** architecture. Different threadblock sizes, managements of the L1-cache, and number of concurrent kernels have been evaluated for the *relax kernel (top)*, *minimum kernel (middle)*, and *update kernel (bottom)*.

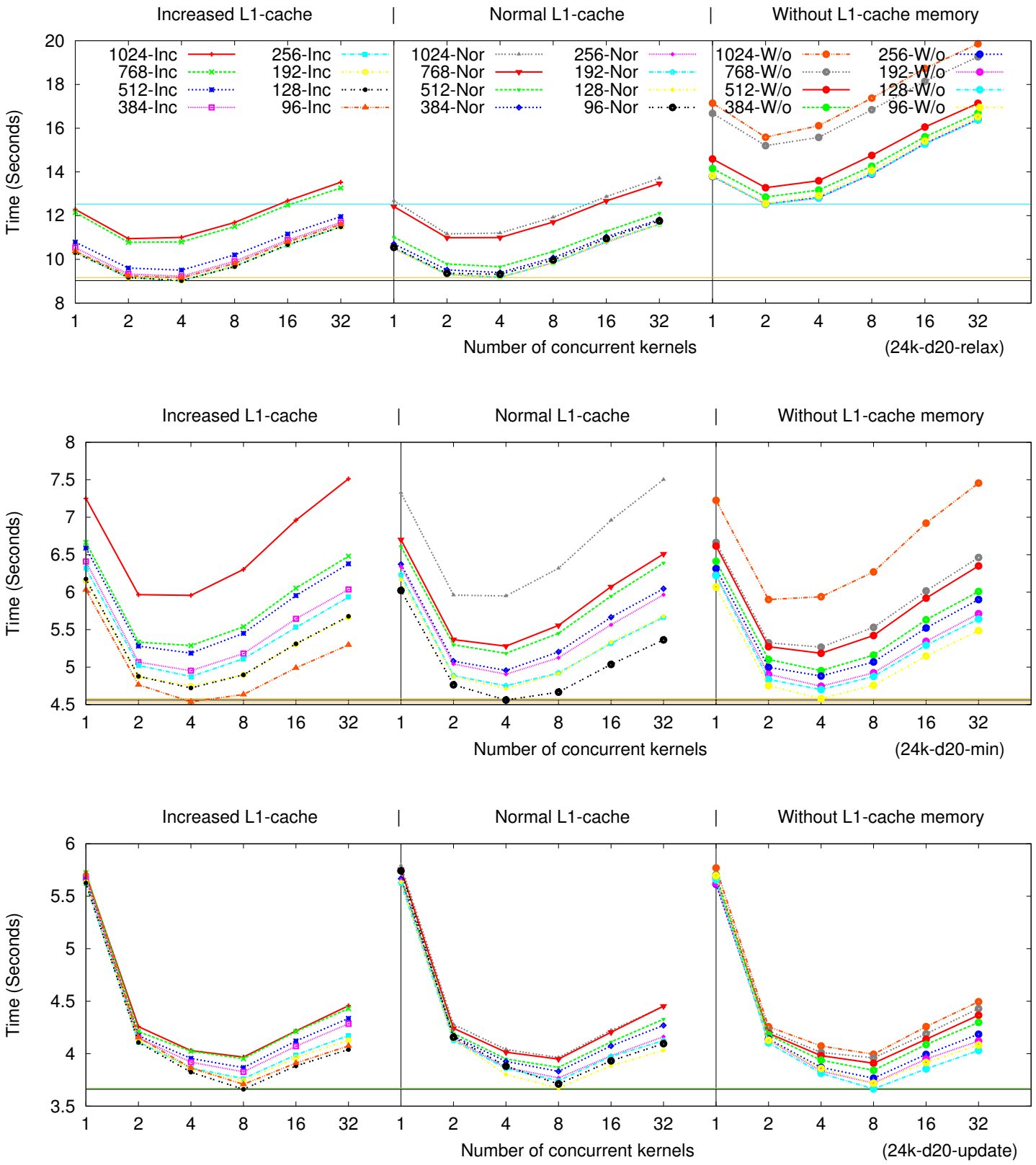


Figure A.11: Exhaustive search for optimal values in the graph **24k-d20** scenario for the **Kepler GK104** architecture. Different threadblock sizes, managements of the L1-cache, and number of concurrent kernels have been evaluated for the *relax kernel (top)*, *minimum kernel (middle)*, and *update kernel (bottom)*.

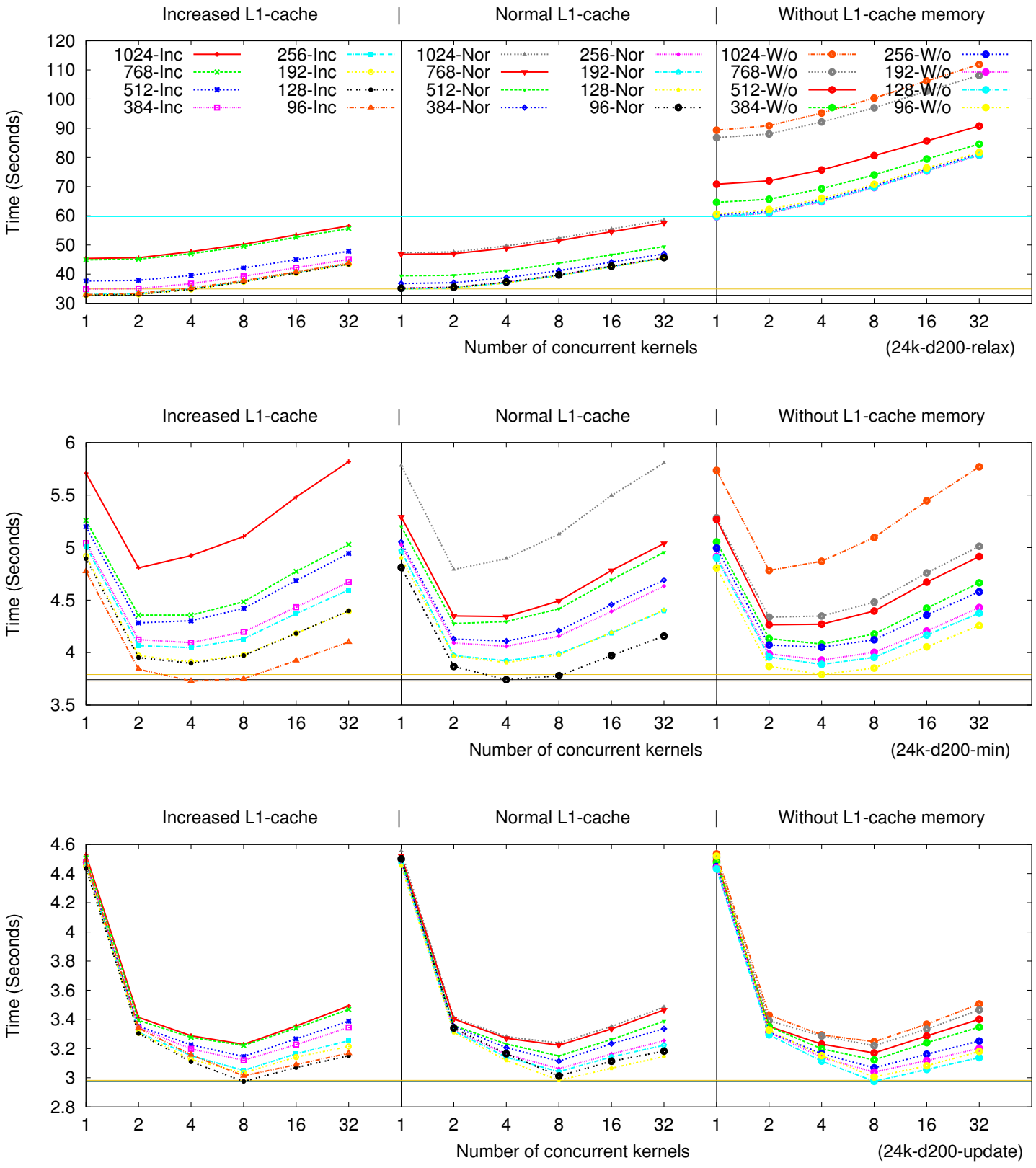


Figure A.12: Exhaustive search for optimal values in the graph **24k-d200** scenario for the **Kepler GK104** architecture. Different threadblock sizes, managements of the L1-cache, and number of concurrent kernels have been evaluated for the *relax kernel (top)*, *minimum kernel (middle)*, and *update kernel (bottom)*.

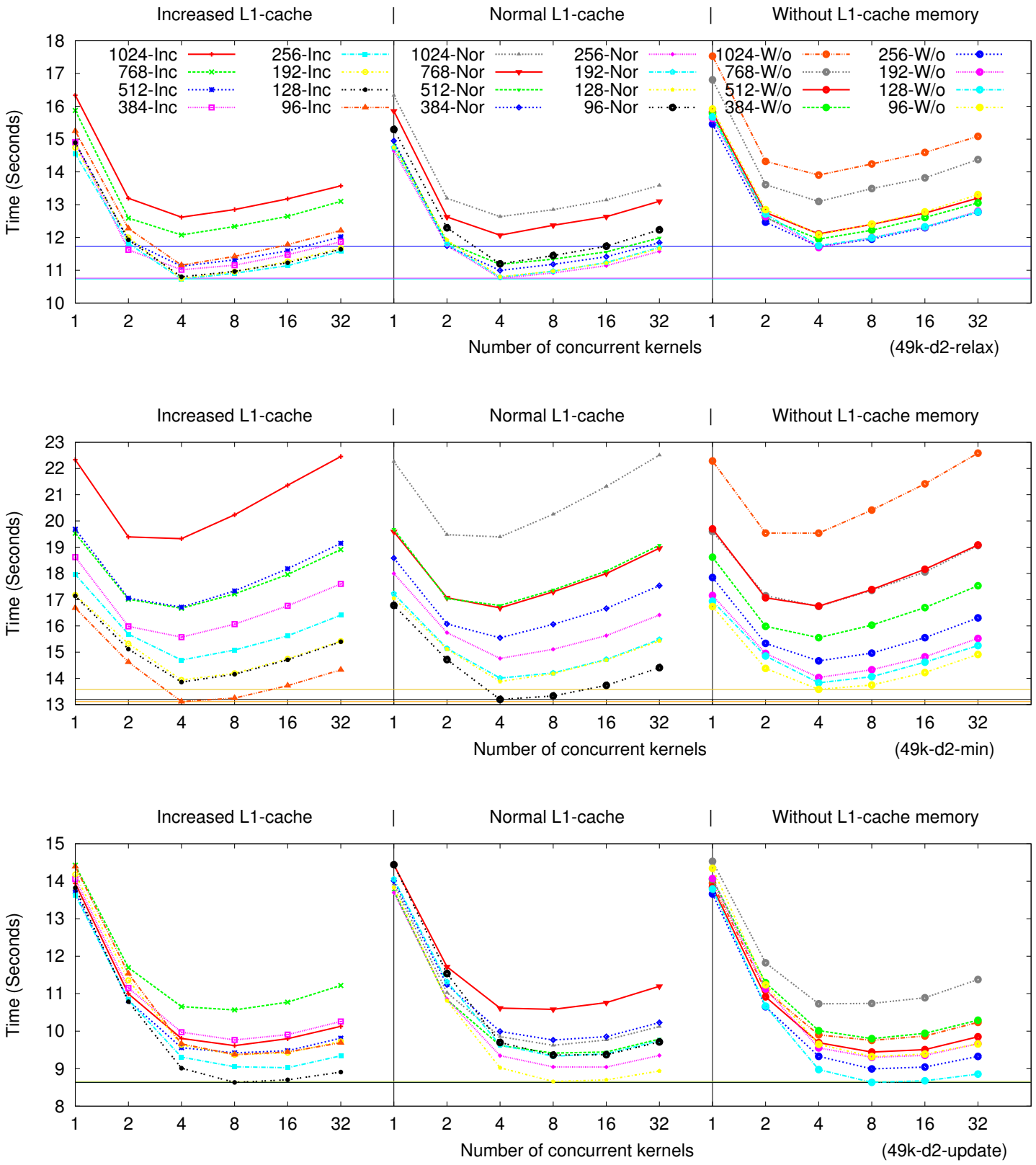


Figure A.13: Exhaustive search for optimal values in the graph **49k-d2** scenario for the **Kepler GK104** architecture. Different threadblock sizes, managements of the L1-cache, and number of concurrent kernels have been evaluated for the *relax kernel (top)*, *minimum kernel (middle)*, and *update kernel (bottom)*.

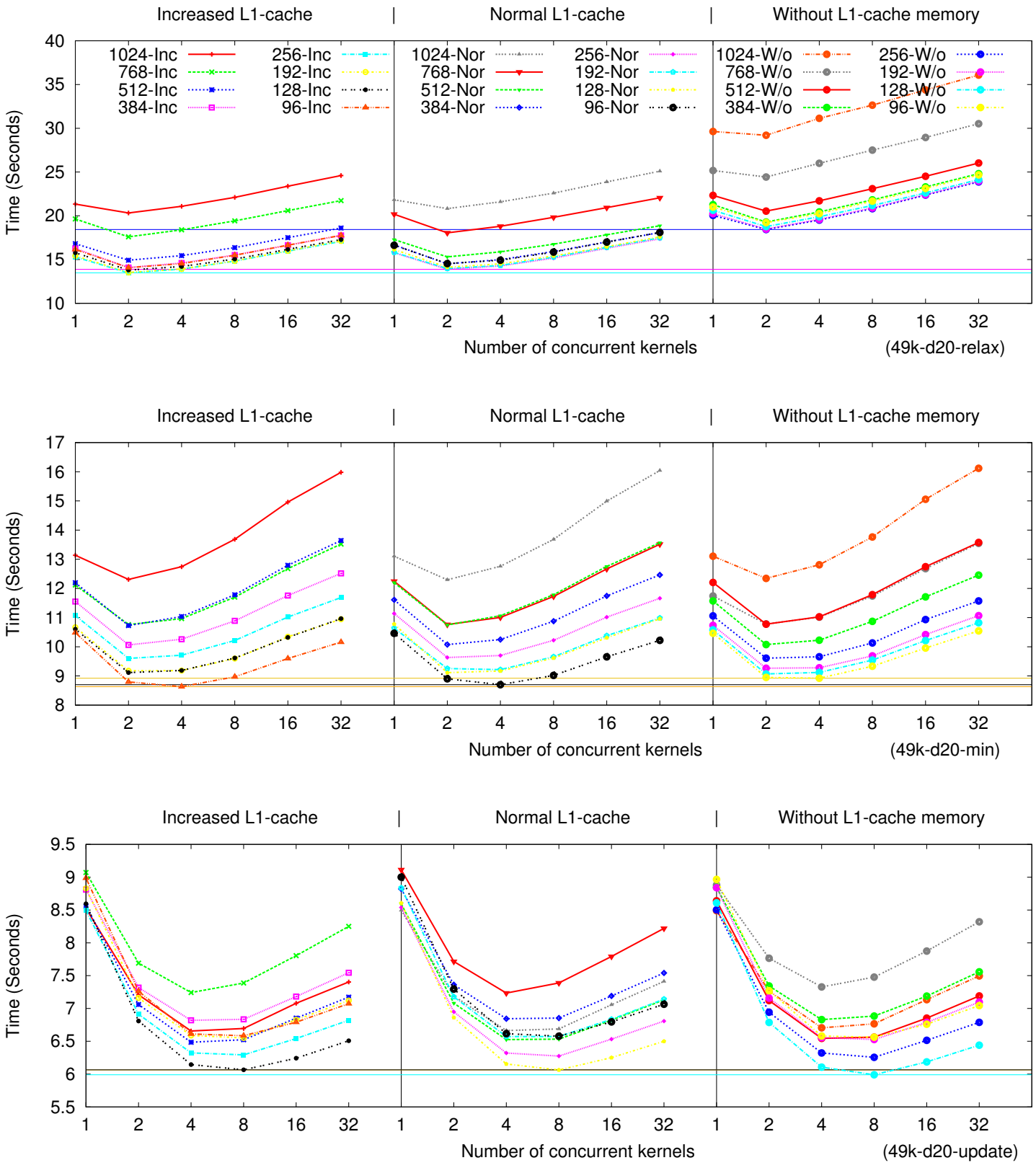


Figure A.14: Exhaustive search for optimal values in the graph **49k-d20** scenario for the **Kepler GK104** architecture. Different threadblock sizes, managements of the L1-cache, and number of concurrent kernels have been evaluated for the *relax kernel (top)*, *minimum kernel (middle)*, and *update kernel (bottom)*.

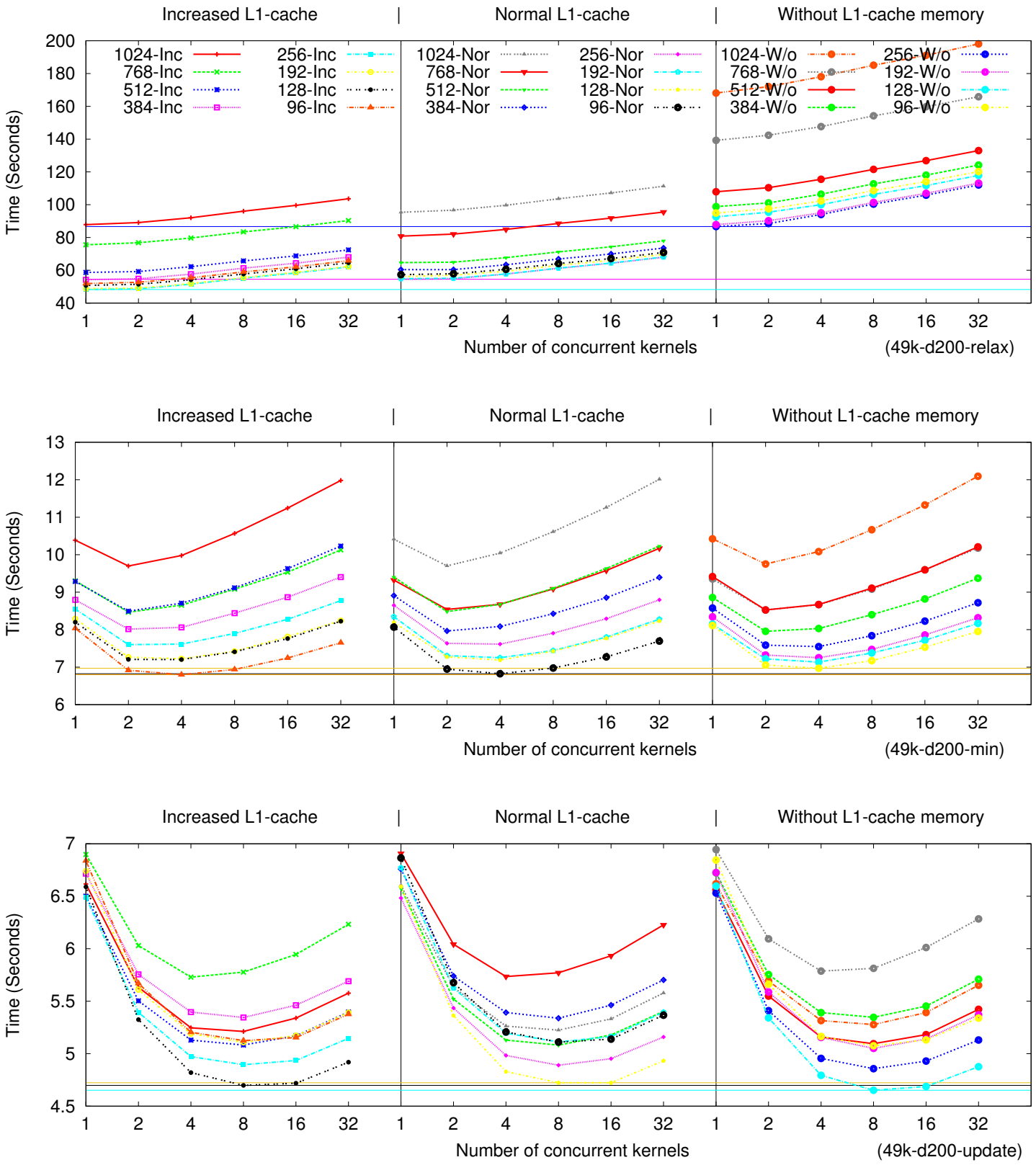


Figure A.15: Exhaustive search for optimal values in the graph **49k-d200** scenario for the **Kepler GK104** architecture. Different threadblock sizes, managements of the L1-cache, and number of concurrent kernels have been evaluated for the *relax kernel (top)*, *minimum kernel (middle)*, and *update kernel (bottom)*.

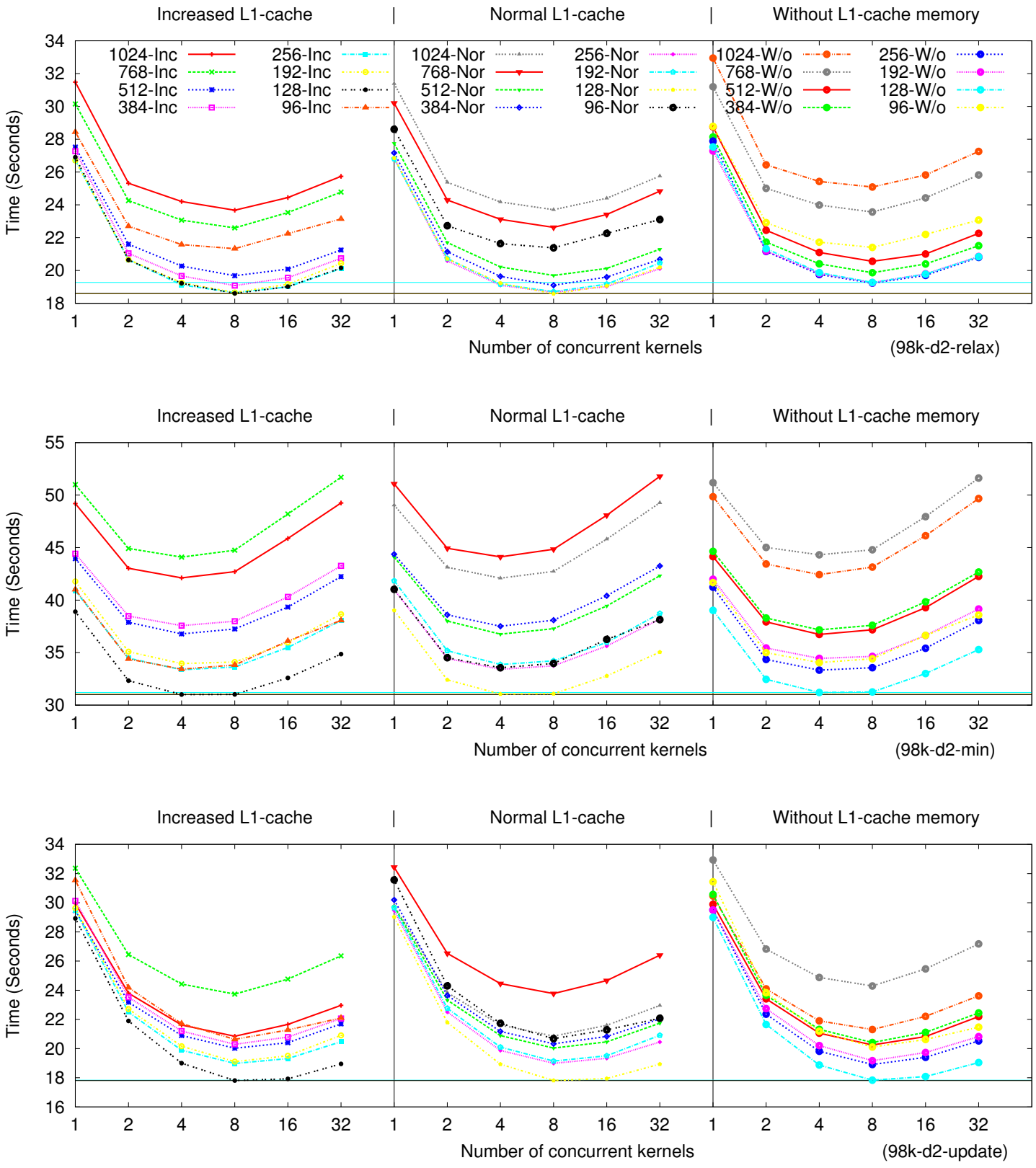


Figure A.16: Exhaustive search for optimal values in the graph **98k-d2** scenario for the **Kepler GK104** architecture. Different threadblock sizes, managements of the L1-cache, and number of concurrent kernels have been evaluated for the *relax kernel (top)*, *minimum kernel (middle)*, and *update kernel (bottom)*.

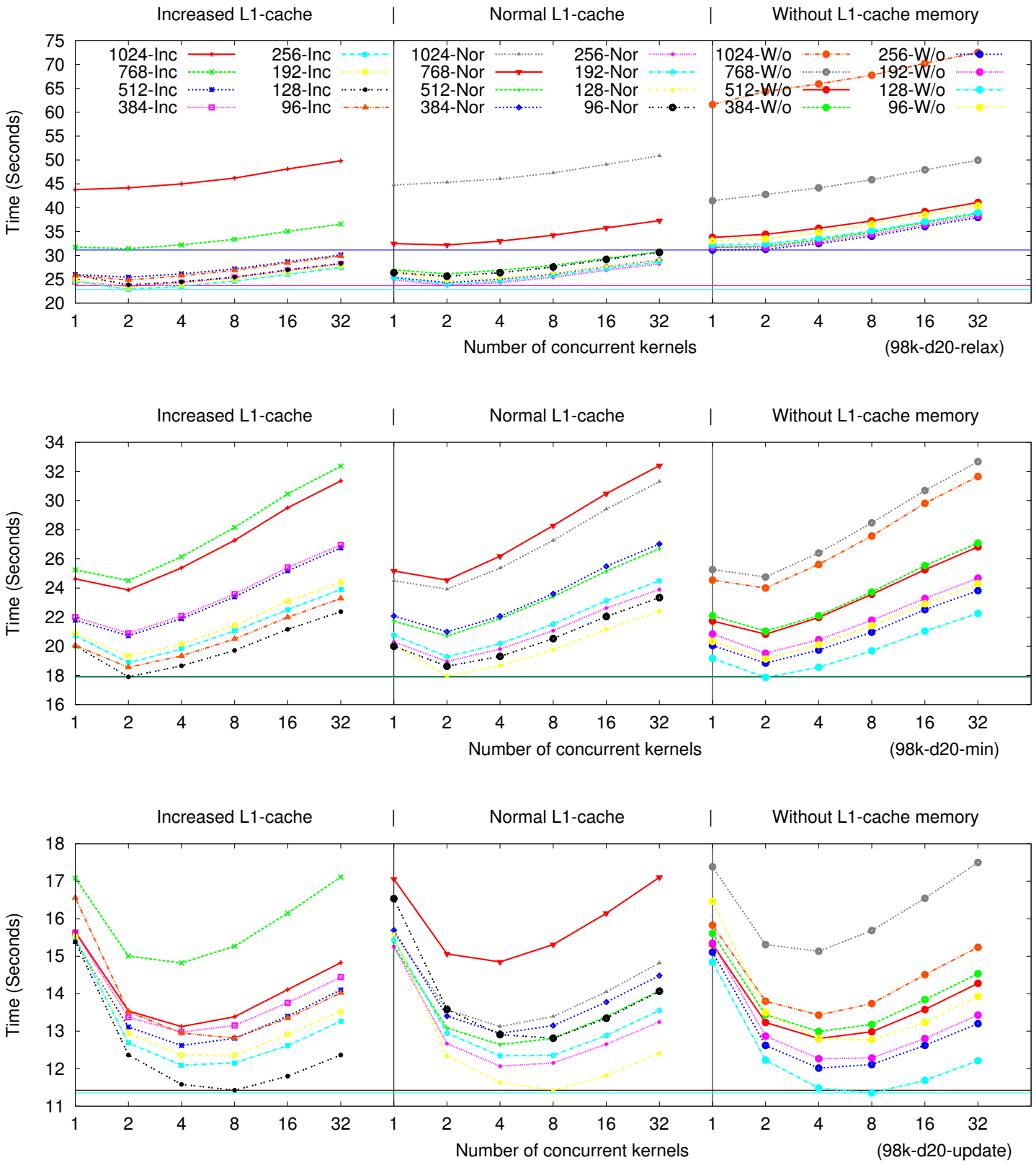


Figure A.17: Exhaustive search for optimal values in the graph **98k-d20** scenario for the **Kepler GK104** architecture. Different threadblock sizes, managements of the L1-cache, and number of concurrent kernels have been evaluated for the *relax kernel (top)*, *minimum kernel (middle)*, and *update kernel (bottom)*.

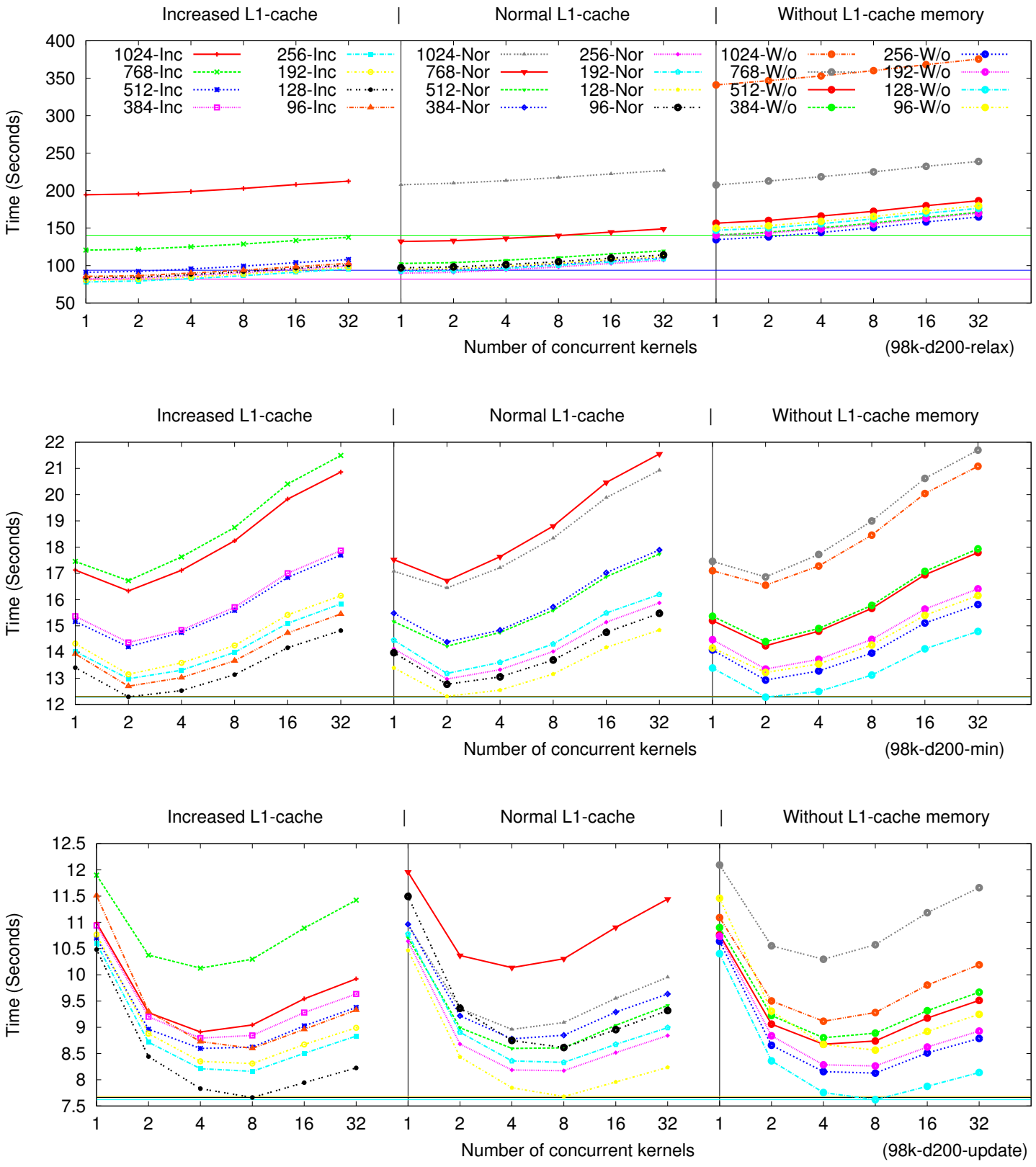


Figure A.18: Exhaustive search for optimal values in the graph **98k-d200** scenario for the **Kepler GK104** architecture. Different threadblock sizes, managements of the L1-cache, and number of concurrent kernels have been evaluated for the *relax kernel (top)*, *minimum kernel (middle)*, and *update kernel (bottom)*.

Bibliography

[1] P. Sanders, D. Schultes, and C. Vetter, "Mobile route planning," in *Proceedings of the 16th Annual European Conference on Algorithms*, ser. ESA'08. Berlin, Germany: Springer Berlin Heidelberg, 2008, pp. 732–743.

[2] J. Barceló, E. Codina, J. Casas, J. L. Ferrer, and D. García, "Microscopic traffic simulation: A tool for the design, analysis and evaluation of intelligent transport systems," *Journal of Intelligent & Robotic Systems*, vol. 41, no. 2-3, pp. 173–203, 2005.

[3] H. Bast, E. Carlsson, A. Eigenwillig, R. Geisberger, C. Harrelson, V. Raychev, and F. Viger, "Fast routing in very large public transportation networks using transfer patterns," in *Proceedings of the 18th Annual European Conference on Algorithms: Part I*, ser. ESA'10. Berlin, Germany: Springer Berlin Heidelberg, 2010, pp. 290–301.

[4] C. Chen and M. Lee, "Global path planning in mobile robot using omnidirectional camera," in *Proceedings of the International Conference on Consumer Electronics, Communications and Networks*, ser. CECNet'11. Washington, D.C., USA: IEEE, Apr 2011, pp. 4986–4989.

[5] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao, "Query processing in spatial network databases," in *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, ser. VLDB '03. Berlin, Germany: VLDB Endowment, 2003, pp. 802–813.

[6] S. Shekhar, A. Fetterer, and B. Goyal, "Materialization Trade-Offs in Hierarchical Shortest Path Algorithms," in *Proceedings of the 5th International Symposium on Advances in Spatial Databases*, ser. SSD '97. London, UK: Springer-Verlag, 1997, pp. 94–111.

[7] G. Rétvári, J. J. Bíró, and T. Cinkler, "On shortest path representation," *IEEE/ACM Trans. Netw.*, vol. 15, pp. 1293–1306, December 2007.

[8] C. Böhm, E. Kny, B. Emde, Z. Abedjan, and F. Naumann, "SPRINT: ranking search results by paths," in *Proceedings of the 14th International Conference on Extending Database Technology*, ser. EDBT/ICDT '11. New York, NY, USA: ACM, 2011, pp. 546–549.

[9] C. Barrett, R. Jacob, and M. Marathe, "Formal-Language-Constrained Path Problems," *SIAM Journal of Computing*, vol. 30, no. 3, pp. 809–837, May 2000.

[10] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck, "Hierarchical Hub Labelings for Shortest Paths," in *Proceedings of the 20th Annual European Conference on Algorithms*, ser. ESA'12. Berlin, Germany: Springer Berlin Heidelberg, 2012, pp. 24–35.

[11] D. Delling, A. Goldberg, and R. Werneck, "Hub Label Compression," in *Experimental Algorithms*, ser. Lecture Notes in Computer Science, V. Bonifaci, C. Demetrescu, and A. Marchetti-Spaccamela, Eds. Berlin, Germany: Springer Berlin Heidelberg, 2013, vol. 7933, p. 18–29.

[12] A. Vajda, "Multi-core and Many-core Processor Architectures," in *Programming Many-Core Chips*. New York, NY, USA: Springer US, 2011, p. 9–43.

[13] NVIDIA, "NVIDIA CUDA C Programming Guide 6.5," 2014, Last visit: June 29th, 2015. [Online]. Available: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[14] C. A. Navarro, N. Hitschfeld-Kahler, and L. Mateu, "A survey on parallel computing and its applications in data-parallel problems using gpu architectures," *Communications in Computational Physics*, vol. 15, no. 2, pp. 285–329, 2014.

[15] Y. Torres, A. Gonzalez-Escribano, and D. Llanos, "Using Fermi Architecture Knowledge to Speed up CUDA and OpenCL Programs," in *Proceedings of the 10th IEEE International Symposium on Parallel and Distributed Processing with Applications*, ser. ISPA '12. Washington, D.C., USA: IEEE, July 2012, pp. 617–624.

[16] Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos, "uBench: Exposing the impact of CUDA block geometry in terms of performance," *Journal of Supercomputing*, vol. 65, no. 3, pp. 1150–1163, 2013.

[17] I. Ekmecic, I. Tartalja, and V. Milutinovic, "A survey of heterogeneous computing: concepts and systems," *Proceedings of the IEEE*, vol. 84, no. 8, pp. 1127–1144, Aug 1996.

[18] A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. O. Storaasli, "State-of-the-art in Heterogeneous Computing," *Sci. Program.*, vol. 18, no. 1, pp. 1–33, Jan 2010.

[19] W. R. Adrion, "Research Methodology in Software Engineering. Summary of the Dagstuhl Workshop on Future Directions in Software Engineering," *SIGSOFT Software Engineering Notes*, vol. 18, no. 1, p. 36–37, 1993.

[20] H. Ortega-Arranz, D. R. Llanos, and A. Gonzalez-Escribano, *The Shortest-Path Problem: Analysis and Comparison of Methods*, 1st ed., ser. Synthesis Lectures on Theoretical Computer Science. San Rafael, CA, USA: Morgan and Claypool Publishers, 2014.

[21] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010.

[22] P. Harish, V. Vineet, and P. J. Narayanan, "Large Graph Algorithms for Massively Multithreaded Architectures," Centre for Visual Information Technology, International Institute of IT, Hyderabad, India, Tech. Rep. IIIT/TR/2009/74, Feb 2009.

[23] P. Martín, R. Torres, and A. Gavilanes, "CUDA Solutions for the SSSP Problem," in *Computational Science – ICCS 2009*, ser. LNCS, G. Allen, J. Nabrzyski, E. Seidel, G. van Albada, J. Dongarra, and P. Sloot, Eds. Berlin, Germany: Springer Berlin Heidelberg, 2009, vol. 5544, pp. 904–913.

[24] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders, "A parallelization of Dijkstra's shortest path algorithm," in *Mathematical Foundations of Compt. Science 1998*, ser. LNCS, L. Brim, J. Gruska, and J. Zlatuška, Eds. Berlin, Germany: Springer Berlin Heidelberg, 1998, vol. 1450, pp. 722–731.

[25] J. G. Siek, L.-Q. Lee, and A. Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., 2002.

[26] H. Ortega-Arranz, Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos, "Comprehensive Evaluation of a New GPU-based Approach to the Shortest Path Problem," *International Journal of Parallel Programming*, p. 1–21, 2015.

[27] H. Ortega-Arranz, Y. Torres, D. R. Llanos, and A. Gonzalez-Escribano, "A New GPU-based Approach to the Shortest Path Problem," in *Proceedings of the 11th International Conference on High Performance Computing and Simulation*, ser. HPCS '2013. Helsinki, Finland: IEEE, 2013, pp. 505–511.

[28] H. Ortega-Arranz, Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos, "Optimizing an APSP implementation for NVIDIA GPUs using kernel characterization criteria," *The Journal of Supercomputing*, vol. 70, no. 2, p. 786–798, 2014.

[29] H. Ortega-Arranz, Y. Torres, D. R. Llanos, and A. Gonzalez-Escribano, "A Tuned, Concurrent-Kernel Approach to Speed Up the APSP Problem," in *Proceedings of the 13th International Conference on Computational and Mathematical Methods in Science and Engineering*, ser. CMMSE '13, I. Hamilton and J. Vigo-Aguilar, Eds., vol. 4, Almería, Spain, 2013, pp. 1114–1125.

[30] ——, *The All-Pair Shortest-Path Problem in Shared-Memory Heterogeneous Systems*, ser. Series on Parallel and Distributed Computing. John Wiley & Sons, Inc., 2014, ch. 15, p. 283–299.

[31] H. Ortega-Arranz, Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos, "TuCCompi: A Multi-layer Model for Distributed Heterogeneous Computing with Tuning Capabilities," *International Journal of Parallel Programming*, p. 1–22, 2015.

[32] H. Ortega-Arranz, Y. Torres, D. R. Llanos, and A. Gonzalez-Escribano, "TuCCompi: A Multi-Layer Programing Model for Heterogeneous Systems with Auto-Tuning Capabilities," in *Proceedings of Workshop on High-level Programming for Heterogeneous and Hierarchical Parallel Systems*, ser. HLPGPU '13, H. 2014, Ed., Vienna, Austria, 2014, pp. 18–25.

[33] J. Fresno, "Supporting general data structures and execution models in runtime environments," Ph.D. dissertation, Universidad de Valladolid, 2015.

[34] IEEE, "IEEE Xplore XML Gateway API," Last visit: June 30th, 2015. [Online]. Available: http://ieeexplore.ieee.org/gateway/

[35] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream computing on graphics hardware," *ACM Trans. Graph.*, vol. 23, no. 3, pp. 777–786, 2004.

[36] Khronos, "Open Computing Language (OpenCL)," 2010, Last visit: June 27th, 2015. [Online]. Available: http://www.khronos.org/opencl/

[37] J. A. Bondy and U. S. R. Murty, *Graph theory with applications*, 6th ed. London, England: Macmillan, 1976.

[38] D. B. West *et al.*, *Introduction to graph theory*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2001.

[39] R. Raman, "Recent results on the single-source shortest paths problem," *SIGACT News*, vol. 28, no. 2, pp. 81–87, Jun 1997.

[40] M. Thorup, "Undirected single-source shortest paths with positive integer weights in linear time," *Journal of the ACM*, vol. 46, no. 3, pp. 362–394, May 1999.

[41] U. Zwick, "Exact and Approximate Distances in Graphs — A Survey," in *Algorithms — ESA 2001*, ser. LNCS, F. M. auf der Heide, Ed. Berlin, Germany: Springer Berlin Heidelberg, 2001, vol. 2161, pp. 33–48.

[42] Meyer, U. and Sanders, P., "Δ-Stepping: a parallelizable shortest path algorithm," *Journal of Algorithms*, vol. 49, no. 1, pp. 114–152, 2003.

[43] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. Cambridge, MA, USA: The MIT Press, 2009.

[44] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[45] A. V. Goldberg, "A simple shortest path algorithm with linear average time," in *Algorithms — ESA 2001*, ser. LNCS, F. M. auf der Heide, Ed. Berlin, Germany: Springer Berlin Heidelberg, 2001, vol. 2161, pp. 230–241.

[46] Seth Pettie and Vijaya Ramachandran, "Computing shortest paths with comparisons and additions," in *Proceedings of the 13th annual ACM-SIAM symposium on Discrete algorithms*, ser. SODA '02. Philadelphia, PA, USA: SIAM, 2002, pp. 267–276.

[47] R. E. Bellman, "On a routing problem," *Quarterly of Applied Mathematics*, vol. 16, no. 1, pp. 87–90, 1958.

[48] L. R. Ford and D. R. Fulkerson, *Flows in Networks*, 1st ed. Princeton, NJ, USA: Princeton University Press, 1963.

[49] M. Thorup, "Floats, Integers, and Single Source Shortest Paths," *Journal of Algorithms*, vol. 35, no. 2, pp. 189–201, 2000.

[50] ——, "Integer priority queues with decrease key in constant time and the single source shortest paths problem," *Journal of Computer and System Sciences*, vol. 69, no. 3, pp. 330–353, 2004.

[51] A. V. Goldberg, "Scaling Algorithms for the Shortest Paths Problem," *SIAM Journal of Computing*, vol. 24, no. 3, pp. 494–504, Jun 1995.

[52] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *Journal of the ACM*, vol. 34, no. 3, p. 596–615, July 1987.

[53] R. K. Ahuja, K. Mehlhorn, J. Orlin, and R. E. Tarjan, "Faster Algorithms for the Shortest Path Problem," *Journal of the ACM*, vol. 37, no. 2, pp. 213–223, Apr 1990.

[54] R. K. Ahuja and T. L. Magnanti and J. B. Orlin, *Network flows: Theory, Algorithms, and Applications*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall, 1993.

[55] J. W. J. Williams, "Algorithm 232: Heapsort," *Communications of the ACM*, vol. 7, no. 6, p. 347–348, 1964.

[56] J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan, "Relaxed Heaps: An Alternative to Fibonacci Heaps with Applications to Parallel Computation," *Communications of the ACM*, vol. 31, no. 11, pp. 1343–1354, Nov 1988.

[57] M. Thorup, "Undirected single-source shortest paths with positive integer weights in linear time," *Journal of the ACM*, vol. 46, no. 3, p. 362–394, 1999.

[58] Meyer, Ulrich, "Single-source Shortest-paths on Arbitrary Directed Graphs in Linear Average-case Time," in *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '01. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics (SIAM), 2001, pp. 797—806.

[59] M. Papaefthymiou and J. Rodrigue, "Implementing Parallel Shortest-Paths Algorithms," in *Parallel Algorithms: Third DIMACS Implemenation Challenge*, ser. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, S. N. Bhatt, Ed. Providence, RI, USA: AMS/DIMACS, 1997, vol. 30, pp. 59–68.

[60] U. Meyer and P. Sanders, "Delta-Stepping: A Parallel Single Source Shortest Path Algorithm," in *Proceedings of the 6th Annual European Symposium on Algorithms*, ser. ESA '98. London, UK: Springer-Verlag, 1998, pp. 393–404.

[61] P. Harish and P. Narayanan, "Accelerating Large Graph Algorithms on the GPU Using CUDA," in *High Performance Computing – HiPC 2007*, ser. LNCS, S. Aluru, M. Parashar, R. Badrinath, and V. Prasanna, Eds. Berlin, Germany: Springer Berlin Heidelberg, 2007, vol. 4873, p. 197–208.

[62] S. Kumar, A. Misra, and R. Tomar, "A modified parallel approach to Single Source Shortest Path Problem for massively dense graphs using CUDA," in *2nd International Conference on Computer and Communication Technology*, ser. ICCCT '11. Washington, D.C., USA: IEEE, Sept 2011, pp. 635–639.

[63] Y. Tang, Y. Zhang, and H. Chen, "A Parallel Shortest Path Algorithm Based on Graph-Partitioning and Iterative Correcting," in *10th IEEE International Conference on High Performance Computing and Communications*, ser. HPCC '08. Washington, D.C., USA: IEEE, Sept 2008, pp. 155–161.

[64] U. Zwick, "All pairs shortest paths in weighted directed graphs-exact and almost exact algorithms," in *Proceedings of 39th Annual Symposium on Foundations of Computer Science*, ser. FoCS '98. Washington, D.C., USA: IEEE, Nov 1998, pp. 310–319.

[65] ——, "All pairs shortest paths using bridging sets and rectangular matrix multiplication," *Journal of the ACM*, vol. 49, no. 3, pp. 289–317, May 2002.

[66] R. Seidel, "On the all-pairs-shortest-path problem in unweighted undirected graphs," *Journal of Computer and System Sciences*, vol. 51, no. 3, pp. 400–403, 1995.

[67] Z. Galil and O. Margalit, "All Pairs Shortest Distances for Graphs with Small Integer Length Edges," *Information and Computation*, vol. 134, no. 2, pp. 103–139, 1997.

[68] ——, "All Pairs Shortest Paths for Graphs with Small Integer Length Edges," *Journal of Computer and System Sciences*, vol. 54, no. 2, pp. 243–254, 1997.

[69] T. Chan, "All-pairs shortest paths for unweighted undirected graphs in O(mn) time," in *Proceedings of the 17th annual ACM-SIAM symposium on Discrete algorithm*, ser. SODA '06. New York, NY, USA: ACM, 2006, pp. 514–523.

[70] D. Karger, D. Koller, and S. Phillips, "Finding the hidden path: Time bounds for all-pairs shortest paths," in *Proceedings of 32nd Annual Symposium on Foundations of Computer Science*, ser. FoCS '91. Washington, D.C., USA: IEEE, 1991, pp. 560–568.

[71] C. McGeoch, "All-pairs shortest paths and the essential subgraph," *Algorithmica*, vol. 13, no. 5, pp. 426–441, 1995.

[72] S. Pettie, "A Faster All-Pairs Shortest Path Algorithm for Real-Weighted Sparse Graphs," in *Automata, Languages and Programming*, ser. LNCS, P. Widmayer, S. Eidenbenz, F. Triguero, R. Morales, R. Conejo, and M. Hennessy, Eds. Berlin, Germany: Springer Berlin Heidelberg, 2002, vol. 2380, pp. 85–97.

[73] ——, "A new approach to all-pairs shortest paths on real-weighted graphs," *Theoretical Computer Science*, vol. 312, no. 1, pp. 47–74, 2004.

[74] D. B. Johnson, "Efficient Algorithms for Shortest Paths in Sparse Networks," *Journal of the ACM*, vol. 24, no. 1, pp. 1–13, Jan 1977.

[75] R. W. Floyd, "Algorithm 97: Shortest path," *Communications of the ACM*, vol. 5, no. 6, pp. 345–, Jun 1962.

[76] S. Warshall, "A Theorem on Boolean Matrices," *Journal of the ACM*, vol. 9, no. 1, pp. 11–12, Jan 1962.

[77] Y. Han and T. Takaoka, "An O($n^3 \log\log n / \log^2 n$) Time Algorithm for All Pairs Shortest Paths," in *Algorithm Theory – SWAT 2012*, ser. LNCS, F. V. Fomin and P. Kaski, Eds. Berlin, Germany: Springer Berlin Heidelberg, 2012, vol. 7357, pp. 131–141.

[78] T. Hagerup, "Improved Shortest Paths on the Word RAM," in *Automata, Languages and Programming*, ser. LNCS, U. Montanari, J. D. Rolim, and E. Welzl, Eds. Berlin, Germany: Springer Berlin Heidelberg, 2000, vol. 1853, pp. 61–72.

[79] A. Shoshan and U. Zwick, "All pairs shortest paths in undirected graphs with integer weights," in *Proceedings of 40th Annual Symposium on Foundations of Computer Science*, ser. FoCS '99. Washington, D.C., USA: IEEE, 1999, pp. 605–614.

[80] H. Yanagisawa, "A multi-source label-correcting algorithm for the all-pairs shortest paths problem," in *IEEE 24th International Symposium on Parallel Distributed Processing*, ser. IPDPS '10. Washington, D.C., USA: IEEE, Apr 2010, pp. 1–10.

[81] M. Hilger, E. Köhler, R. H. Möhring, and H. Schilling, "Fast Point-to-Point Shortest Path Computations with Arc-Flags," in *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, ser. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, C. Demetrescu, A. V. Goldberg, and D. S. Johnson, Eds. Providence, RI, USA: American Mathematical Society, 2009, vol. 74, pp. 41–72.

[82] C. Demetrescu and G. F. Italiano, "Engineering Shortest Path Algorithms," in *Experimental and Efficient Algorithms*, ser. LNCS, C. C. Ribeiro and S. L. Martins, Eds. Berlin, Germany: Springer Berlin Heidelberg, 2004, vol. 3059, pp. 191–198.

[83] W. Peng, X. Hu, F. Zhao, and J. Su, "A Fast Algorithm to Find All-Pairs Shortest Paths in Complex Networks," *Procedia Computer Science*, vol. 9, no. 0, pp. 557–566, 2012.

[84] M. L. Fredman, "New bounds on the complexity of the shortest path problem," *SIAM Journal of Computing*, vol. 5, no. 1, pp. 49–60, 1976.

[85] T. Takaoka, "A new upper bound on the complexity of the all pairs shortest path problem," *Information Processing Letters*, vol. 43, no. 4, pp. 195–199, 1992.

[86] W. Dobosiewicz, "A more efficient algorithm for the min-plus multiplication," *International Journal of Computer Mathematics*, vol. 32, no. 1-2, pp. 49–60, 1990.

[87] Y. Han, "Improved algorithm for all pairs shortest paths," *Information Processing Letters*, vol. 91, no. 5, pp. 245–250, 2004.

[88] T. Takaoka, "A Faster Algorithm for the All-Pairs Shortest Path Problem and Its Application," in *Computing and Combinatorics*, ser. LNCS, K.-Y. Chwa and J. I. Munro, Eds. Berlin, Germany: Springer Berlin Heidelberg, 2004, vol. 3106, pp. 278–289.

[89] ——, "An $o(n^3 \log\log n/\log n)$ time algorithm for the all-pairs shortest path problem," *Information Processing Letters*, vol. 96, no. 5, pp. 155–161, 2005.

[90] U. Zwick, "A Slightly Improved Sub-cubic Algorithm for the All Pairs Shortest Paths Problem with Real Edge Lengths," in *Algorithms and Computation*, ser. LNCS, R. Fleischer and G. Trippen, Eds. Berlin, Germany: Springer Berlin Heidelberg, 2004, vol. 3341, pp. 921–932.

[91] ——, "A Slightly Improved Sub-Cubic Algorithm for the All Pairs Shortest Paths Problem with Real Edge Lengths," *Algorithmica*, vol. 46, no. 2, pp. 181–192, 2006.

[92] T. Chan, "All-Pairs Shortest Paths with Real Weights in $O(n^3/\log n)$ Time," in *Algorithms and Data Structures*, ser. LNCS, F. Dehne, A. López-Ortiz, and J.-R. Sack, Eds. Berlin, Germany: Springer Berlin Heidelberg, 2005, vol. 3608, pp. 318–324.

[93] ——, "All-Pairs Shortest Paths with Real Weights in $O(n^3/\log n)$ time," *Algorithmica*, vol. 50, no. 2, pp. 236–243, 2008.

[94] Y. Han, "An $O(n^3(\log\log n/\log n)^{5/4})$ Time Algorithm for All Pairs Shortest Paths," in *Algorithms – ESA 2006*, ser. LNCS, Y. Azar and T. Erlebach, Eds. Berlin, Germany: Springer Berlin Heidelberg, 2006, vol. 4168, pp. 411–417.

[95] ——, "An $O(n^3(\log\log n/\log n)^{5/4})$ Time Algorithm for All Pairs Shortest Path," *Algorithmica*, vol. 51, no. 4, pp. 428–434, 2008.

[96] T. Chan, "More algorithms for all-pairs shortest paths in weighted graphs," in *Proceedings of the 39th Annual ACM Symposium on Theory of Computing*, ser. STOC '07. New York, NY, USA: ACM, 2007, pp. 590–598.

[97] G. Venkataraman, S. Sahni, and S. Mukhopadhyaya, "A blocked all-pairs shortest-paths algorithm," *Journal of Experimental Algorithmics*, vol. 8, no. 2.2, pp. 1–19, Dec 2003.

[98] G. J. Katz and J. T. Kider, Jr, "All-pairs shortest-paths for large graphs on the gpu," in *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, ser. GH '08. Aire-la-Ville, Switzerland: Eurographics Association, 2008, pp. 47–55.

[99] J.-S. Park, M. Penner, and V. Prasanna, "Optimizing graph algorithms for improved cache performance," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 9, pp. 769–782, Sep 2004.

[100] M. Penner and V. K. Prasanna, "Cache-friendly implementations of transitive closure," *Journal of Experimental Algorithmics*, vol. 11, no. 1.3, pp. 1–25, Feb 2007.

[101] C. Duin, "A Branch-Checking Algorithm for All-Pairs Shortest Paths," *Algorithmica*, vol. 41, no. 2, pp. 131–145, 2005.

[102] S.-C. Han, F. Franchetti, and M. Püschel, "Program generation for the all-pairs shortest path problem," in *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '06. New York, NY, USA: ACM, 2006, pp. 222–232.

[103] V. V. Williams, "Multiplying matrices faster than coppersmith-winograd," in *Proceedings of the 44th symposium on Theory of Computing*, ser. STOC '12. New York, NY, USA: ACM, 2012, pp. 887–898.

[104] B. Diament and A. Ferencz, "Comparison of Parallel APSP Algorithms," 1999, Last visit: July 6th, 2015. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.4556

[105] P. Micikevicius, "General Parallel Computation on Commodity Graphics Hardware: Case Study with the All-Pairs Shortest Paths Problem," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications - Volume 3*, ser. PDPTA '04. Las Vegas, NV, USA: CSREA Press, 2004, pp. 1359–1365.

[106] A. Buluç, J. R. Gilbert, and C. Budak, "Solving Path Problems on the GPU," *Parallel Computing*, vol. 36, no. 5-6, pp. 241–253, Jun 2010.

[107] Wu, Junkai, "Accelerating all-pairs shortest path search on GPUs," Master's thesis, Department of Science and Engineering, Hong Kong University of Science and Technology, Hong Kong, Aug 2013.

[108] H. Djidjev, S. Thulasidasan, G. Chapuis, R. Andonov, and D. Lavenier, "Efficient Multi-GPU Computation of All-Pairs Shortest Paths," in *IEEE 28th International Symposium on Parallel Distributed Processing*, ser. IPDPS '14. Washington, D.C., USA: IEEE, May 2014, pp. 360–369.

[109] B. K. Sen, "Study of Parallel Graph Algorithms for Minimum Spanning Tree and All-Pairs Shortest-Paths Using a Large Scale Cluster," Master's thesis, School of Engineering and Computer Science, Independent University of Bangladesh, Bangladesh, Jun 2006.

[110] T. Okuyama, F. Ino, and K. Hagihara, "A Task Parallel Algorithm for Computing the Costs of All-Pairs Shortest Paths on the CUDA-Compatible GPU," in *Proceedings of the 6th IEEE International Symposium on Parallel and Distributed Processing with Applications*, ser. ISPA '08. Washington, D.C., USA: IEEE, Dec 2008, pp. 284–291.

[111] ——, "A Task Parallel Algorithm for Finding All-pairs Shortest Paths Using the GPU," *International Journal of High Performance Computing and Networking*, vol. 7, no. 2, pp. 87–98, Apr 2012.

[112] G. Hajela and M. Pandey, "Parallel Implementations for Solving Shortest Path Problem using Bellman-Ford," *International Journal of Computer Applications*, vol. 95, no. 15, pp. 1–6, June 2014.

[113] ——, "A Fine Tuned Hybrid Implementation for Solving Shortest Path Problems using Bellman Ford," *International Journal of Computer Applications*, vol. 99, no. 2, pp. 29–33, Aug 2014.

[114] V. Volkov and J. Demmel, "LU, QR and Cholesky Factorizations using Vector Capabilities of GPUs," EECS Department, University of California, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2008-49, 2008.

[115] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, 2nd ed. New York, NY, USA: Pearson Education, 2003.

[116] R. Bauer, D. Delling, and D. Wagner, "Experimental study of speed up techniques for timetable information systems," *Networks*, vol. 57, no. 1, p. 38–52, 2011.

[117] G. Dantzig, *Linear Programming And Extensions*, 1st ed. Princeton, NJ, USA: Princeton University Press, 1963.

[118] T. A. J. Nicholson, "Finding the Shortest Route Between Two Points in a Network," *The Computer Journal*, vol. 9, no. 3, pp. 275–280, 1966.

[119] D. Dreyfus, "An Appraisal of Some Shortest Path Algorithms," Rand Corporation, Santa Monica, CA, USA, Tech. Rep. RM-5433, 1967.

[120] A. V. Goldberg and C. Harrelson, "Computing the shortest path: A search meets graph theory," Microsoft Research, Vancouver, Canada, Tech. Rep. MSR-TR-2004-24, 2004.

[121] P. E. Hart, N. J. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[122] R. J. Gutman, "Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks," in *Proceedings 6th Workshop on Algorithm Engineering and Experiments*, ser. ALENEX'04. Philadelphia, PA, USA: SIAM, 2004, pp. 100–111.

[123] I. S. Pohl, "Bi-directional and heuristic search in path problems," Ph.D. dissertation, Stanford University, 1969.

[124] A. V. Goldberg and C. Harrelson, "Computing the shortest path: A search meets graph theory," in *Proceedings of the 16th annual ACM-SIAM Symposium On Discrete Algorithms*, ser. SODA'05. Philadelphia, PA, USA: SIAM, 2005, pp. 156–165.

[125] C. Sommer, "Shortest-path Queries in Static Networks," *ACM Computing Surveys*, vol. 46, no. 4, pp. 45:1–45:31, Mar 2014.

[126] H. Bast, D. Delling, A. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. Werneck, "Route Planning in Transportation Networks," Microsoft Research, Vancouver, Canada, Tech. Rep. MSR-TR-2014-4, January 2014.

[127] P. Sanders and D. Schultes, "Highway hierarchies hasten exact shortest path queries," in *Proceedings of the 13th Annual European Conference on Algorithms*, ser. ESA'05. Berlin, Germany: Springer Berlin Heidelberg, 2005, pp. 568–579.

[128] ——, "Engineering highway hierarchies," *Journal of Experimental Algorithmics*, vol. 17, no. 1, pp. 1.6:1.1–1.6:1.40, Sep 2012.

[129] ——, "Engineering highway hierarchies," in *Proceedings of the 14th Annual European Conference on Algorithms*, ser. ESA'06. Berlin, Germany: Springer Berlin Heidelberg, 2006, pp. 804–816.

[130] D. Schultes and P. Sanders, "Dynamic Highway-node Routing," in *Proceedings of the 6th International Conference on Experimental Algorithms*, ser. WEA'07. Berlin, Germany: Springer Berlin Heidelberg, 2007, pp. 66–79.

[131] H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes, "In transit to constant time shortest-path queries in road networks," in *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments*, ser. ALENEX'07. Philadelphia, PA, USA: SIAM, 2007, pp. 46–59.

[132] R. Geisberger, P. Sanders, D. Schultes, and D. Delling, "Contraction hierarchies: faster and simpler hierarchical routing in road networks," in *Proceedings of the 7th International Conference on Experimental Algorithms*, ser. WEA'08. Berlin, Germany: Springer Berlin Heidelberg, 2008, pp. 319–333.

[133] R. Geisberger, P. Sanders, D. Schultes, and C. Vetter, "Exact Routing in Large Road Networks Using Contraction Hierarchies," *Transportation Science*, vol. 46, no. 3, pp. 388–404, Aug 2012.

[134] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck, "A Hub-based Labeling Algorithm for Shortest Paths in Road Networks," in *Proceedings of the 10th International Conference on Experimental Algorithms*, ser. SEA'11. Berlin, Germany: Springer Berlin Heidelberg, 2011, pp. 230–241.

[135] A. V. Goldberg, H. Kaplan, and R. F. Werneck, "Better landmarks within reach," in *Proceedings of the 6th International Conference on Experimental Algorithms*, ser. WEA'07. Berlin, Germany: Springer Berlin Heidelberg, 2007, pp. 38–51.

[136] D. Wagner, T. Willhalm, and C. Zaroliagis, "Geometric containers for efficient shortest-path computation," *Journal of Experimental Algorithmics*, vol. 10, no. 1.3, pp. 1–30, Dec 2005.

[137] U. Lauther, "An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background," *Geoinformation und Mobilität – von der Forschung zur praktischen Anwendung*, vol. 22, no. 1, pp. 219–230, 2004.

[138] J. Maue, P. Sanders, and D. Matijevic, "Goal Directed Shortest Path Queries Using Precomputed Cluster Distances," in *Proceedings of the 5th International Conference on Experimental Algorithms*, ser. WEA'06. Berlin, Germany: Springer Berlin Heidelberg, 2006, pp. 316–327.

[139] ——, "Goal-directed shortest-path queries using precomputed cluster distances," *Journal of Experimental Algorithmics*, vol. 14, no. 2, pp. 2:3.2–2:3.27, Jan 2010.

[140] R. Bauer, D. Delling, P. Sanders, D. Schieferdecker, D. Schultes, and D. Wagner, "Combining hierarchical and goal-directed speed-up techniques for Dijkstra's algorithm," *Journal of Experimental Algorithmics*, vol. 15, no. 2.3, pp. 2.3:2.1–2.3:2.31, Mar 2010.

[141] D. Delling, M. Holzer, K. Müller, F. Schulz, and D. Wagner, "High-performance multi-level graphs," C. Demetrescu, A. V. Goldberg, and D. S. Johnson, Eds. Providence, RI, USA: American Mathematical Society, 2006.

[142] D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck, "Customizable route planning," in *Proceedings of the 10th International Conference on Experimental Algorithms*, ser. SEA'11. Berlin, Germany: Springer Berlin Heidelberg, 2011, pp. 376–387.

[143] D. Delling, P. Sanders, D. Schultes, and D. Wagner, "Highway hierarchies star," C. Demetrescu, A. V. Goldberg, and D. S. Johnson, Eds. Providence, RI, USA: American Mathematical Society, 2006.

[144] R. Bauer and D. Delling, "SHARC: Fast and robust unidirectional routing," in *Proceedings of the 10th Workshop on Algorithm Engineering and Experiments*, ser. ALENEX'08. Philadelphia, PA, USA: SIAM, 2008, pp. 13–26.

[145] ——, "Sharc: Fast and robust unidirectional routing," *Journal of Experimental Algorithmics*, vol. 14, no. 4, pp. 4:2.4–4:2.29, Jan 2010.

[146] R. Bauer, D. Delling, P. Sanders, D. Schieferdecker, D. Schultes, and D. Wagner, "Combining hierarchical and goal-directed speed-up techniques for Dijkstra's algorithm," in *Proceedings of the 7th International Conference on Experimental Algorithms*, ser. WEA'08. Berlin, Germany: Springer Berlin Heidelberg, 2008, pp. 303–318.

[147] M. Harris, *Optimizing Parallel Reduction in CUDA*, developer.download.nvidia.com/assets/cuda/files/reduction.pdf, nVidia, 2008.

[148] S. Nobari, X. Lu, P. Karras, and S. Bressan, "Fast random graph generation," in *Proceedings of the 14th International Conference on Extending Database Technology*, ser. EDBT/ICDT '11. New York, NY, USA: ACM, 2011, pp. 331–342.

[149] "DIMACS implementation challenge," 2012, Last visit: June 29th, 2015. [Online]. Available: http://www.cc.gatech.edu/dimacs10/downloads.shtml

[150] D. Bader, H. Meyerhenke, P. Sanders, C. Schulz, A. Kappes, and D. Wagner, "Benchmarking for Graph Clustering and Partitioning," in *Encyclopedia of Social Network Analysis and Mining*, R. Alhajj and J. Rokne, Eds. New York, NY, USA: Springer New York, 2014, p. 73–82.

[151] David F Gleich, "Graph of Flickr Photo-Sharing Social Network Crawled in May 2006," Feb 2012, Last visit: June 29th, 2015. [Online]. Available: https://purr.purdue.edu/publications/1002

[152] E. Wynters, "Parallel Processing on NVIDIA Graphics Processing Units Using CUDA," *Journal of Computing Sciences in Colleges*, vol. 26, no. 3, pp. 58–66, Jan 2011.

[153] V. Challis, A. Roberts, and J. Grotowski, "High resolution topology optimization using graphics processing units (GPUs)," *Structural and Multidisciplinary Optimization*, vol. 49, no. 2, p. 315–325, 2014.

[154] Y. Torres, A. Gonzalez-Escribano, and D. Llanos, "Understanding the impact of CUDA tuning techniques for Fermi," in *Proceedings of the 9th International Conference on High Performance Computing and Simulation*, ser. HPCS '2011. Washington, D.C., USA: IEEE, July 2011, pp. 631–639.

[155] X. Cui, Y. Chen, C. Zhang, and H. Mei, "Auto-tuning Dense Matrix Multiplication for GPGPU with Cache," in *IEEE 16th International Conference on Parallel and Distributed Systems*, ser. ICPADS '10. Washington, D.C., USA: IEEE, Dec 2010, pp. 237–242.

[156] A. Shahzad, M. O'Halloran, M. Glavin, and E. Jones, "A novel optimized parallelization strategy to accelerate microwave tomography for breast cancer screening," in *36th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, ser. EMBC '14. Washington, D.C., USA: IEEE, Aug 2014, pp. 2456–2459.

[157] T. Aila and S. Laine, "Understanding the Efficiency of Ray Traversal on GPUs," in *Proceedings of the Conference on High Performance Graphics*, ser. HPG '09. New York, NY, USA: ACM, 2009, pp. 145–149.

[158] C. Wu, S. Agarwal, B. Curless, and S. Seitz, "Multicore bundle adjustment," in *IEEE Conference on Computer Vision and Pattern Recognition*, ser. CVPR '11. Washington, D.C., USA: IEEE, June 2011, pp. 3057–3064.

[159] N. Maruyama and T. Aoki, "Optimizing stencil computations for NVIDIA Kepler GPUs," in *Proceedings of the 1st International Workshop on High-Performance Stencil Computations*, A. Größlinger and H. Köstler, Eds., Vienna, Austria, Jan. 2014, pp. 89–95. [Online]. Available: http://www.exastencils.org/histencils/2014/

[160] G. Quintana-Ortí, F. D. Igual, E. S. Quintana-Ortí, and R. A. van de Geijn, "Solving Dense Linear Systems on Platforms with Multiple Hardware Accelerators," in *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '09. New York, NY, USA: ACM, 2009, pp. 121–130.

[161] J. Stratton, S. Stone, and W.-m. Hwu, "MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs," in *Languages and Compilers for Parallel Computing*, ser. LNCS, J. Amaral, Ed. Berlin, Germany: Springer Berlin Heidelberg, 2008, vol. 5335, p. 16–30.

[162] Y. Torres, A. González-Escribano, and D. R. Llanos, "uBench: Performance Impact of CUDA Block Geometry," Universidad de Valladolid, Valladolid, Spain, Tech. Rep. IT-DI-2012-0001, 2012.

[163] S. Williams, A. Waterman, and D. Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, Apr 2009.

[164] NVIDIA, "Tuning CUDA Applications for Kepler," 2015, Last visit: June 29th, 2015. [Online]. Available: http://docs.nvidia.com/cuda/kepler-tuning-guide/#l1-cache

[165] ——, "Tuning CUDA Applications for Maxwell," 2015, Last visit: June 29th, 2015. [Online]. Available: http://docs.nvidia.com/cuda/maxwell-tuning-guide/#l1-cache

[166] W. Shen, Z. Luo, D. Wei, W. Xu, and X. Zhu, "Load-prediction scheduling algorithm for computer simulation of electrocardiogram in hybrid environments," *Journal of Systems and Software*, vol. 102, no. 0, pp. 182—191, 2015.

[167] A. Heinecke, W. Eckhardt, M. Horsch, and H.-J. Bungartz, "Parallelization of MD Algorithms and Load Balancing," in *Supercomputing for Molecular Dynamics Simulations*, ser. SpringerBriefs in Computer Science. Gewerbestrasse, Switzerland: Springer International Publishing, 2015, pp. 31—-44.

[168] V. V. Kindratenko, R. J. Brunner, and A. D. Myers, "Dynamic load-balancing on multi-FPGA systems: a case study," *ArXiv e-prints*, Nov. 2007.

[169] S. Singh, "Computing without processors," *Communications of the ACM*, vol. 54, pp. 46–54, August 2011.

[170] D. Houzet, S. Huet, and A. Rahman, "SysCellC: a data-flow programming model on multi-GPU," *Procedia Computer Science*, vol. 1, no. 1, p. 1035–1044, 2010.

[171] S. Tzeng, A. Patney, and J. D. Owens, "Task Management for Irregular-Parallel Workloads on the GPU," in *Proceedings of the Conference on High Performance Graphics*, ser. HPG '10. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2010, pp. 29–37.

[172] C. de la Lama, P. Toharia, J. Bosque, and O. Robles, "Static Multi-device Load Balancing for OpenCL," in *Proceedings of the 10th IEEE International Symposium on Parallel and Distributed Processing with Applications*, ser. ISPA '12. Washington, D.C., USA: IEEE, July 2012, pp. 675–682.

[173] A. Binotto, C. Pereira, and D. Fellner, "Towards dynamic reconfigurable load-balancing for hybrid desktop platforms," in *IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, ser. IPDPSW '10. Washington, D.C., USA: IEEE, april 2010, pp. 1–4.

[174] A. Leung, O. Lhoták, and G. Lashari, "Automatic Parallelization for Graphics Processing Units," in *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, ser. PPPJ '09. New York, NY, USA: ACM, 2009, pp. 91–100.

[175] N. R. Satish, "Compile Time Task and Resource Allocation of Concurrent Applications to Multiprocessor Systems," Ph.D. dissertation, EECS Department, University of California, Berkeley, Jan 2009.

[176] E. Burrows and M. Haveraaen, "A Hardware Independent Parallel Programming Model," *Journal of Logic and Algebraic Programming*, vol. 78, pp. 519–538, 2009.

[177] E. Hermann, B. Raffin, F. Faure, T. Gautier, and J. Allard, "Multi-GPU and multi-CPU parallelization for interactive physics simulations," in *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part II*, ser. Euro-Par '10. London, UK: Springer-Verlag, 2010, pp. 235–246.

[178] D. Cederman and P. Tsigas, "On Sorting and Load Balancing on GPUs," *SIGARCH Comput. Archit. News*, vol. 36, no. 5, pp. 11–18, Jun 2009.

[179] P. Yao, H. An, M. Xu, G. Liu, X. Li, Y. Wang, and W. Han, "CuHMMer: A load-balanced CPU-GPU cooperative bioinformatics application," in *Proceedings of the 8th International Conference on High Performance Computing and Simulation*, ser. HPCS '2010. Washington, D.C., USA: IEEE, July 2010, pp. 24–30.

[180] I. Foster, *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., 1995.

[181] U. Hoelzle and L. A. Barroso, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*, 1st ed. San Rafael, CA, USA: Morgan and Claypool Publishers, 2009.

[182] W. Cirne, D. Paranhos, L. Costa, E. Santos-Neto, F. Brasileiro, J. Sauve, F. A. B. Silva, C. Barros, and C. Silveira, "Running Bag-of-Tasks applications on computational grids: The MyGrid approach," in *Proceedings of the 2003 International Conference on Parallel Processing*, ser. ICPP 2003. Washington, D.C., USA: IEEE, 2003, pp. 407–416.

[183] R. Mangharam and A. A. Saba, "Anytime Algorithms for GPU Architectures," in *Proceedings of the 2011 IEEE 32Nd Real-Time Systems Symposium*, ser. RTSS '11. Washington, D.C., USA: IEEE Computer Society, 2011, pp. 47–56.

[184] M. B. Taylor, "Bitcoin and the Age of Bespoke Silicon," in *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ser. CASES '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 16:1–16:10.

[185] M. Ogata, S. Muraki, X. Liu, and K.-L. Ma, "The Design and Evaluation of a Pipelined Image Compositing Device for Massively Parallel Volume Rendering," in *Proceedings of the 2003 Eurographics/IEEE TVCG Workshop on Volume Graphics*, ser. VG '03. New York, NY, USA: ACM, 2003, pp. 61–68.

[186] Z. Chen, X. Chen, Z. Shao, Z. Yao, and L. T. Biegler, "Parallel calculation methods for molecular weight distribution of batch free radical polymerization," *Computers & Chemical Engineering*, vol. 48, no. 0, pp. 175–186, 2013.

[187] S. Kazmi, M. Kane, and M. Krauthammer, "Benchmarking technology infrastructures for embarrassingly and non-embarrassingly parallel problems in biomedical domain," in *Biomedical Sciences and Engineering Conference*, ser. BSEC '13. Washington, D.C., USA: IEEE, 2013, pp. 1–4.

[188] A. Khlopotine, V. Jandhyala, and D. Kirkpatrick, "A Variant of Parallel Plane Sweep Algorithm for Multicore Systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 6, pp. 966–970, 2013.

[189] R. Reyes and F. de Sande, "Optimization strategies in different CUDA architectures using llCoMP," *Microprocess. Microsyst.*, vol. 36, no. 2, pp. 78–87, Mar 2012.

[190] T. Liang, H. Li, and J. Chiu, "Enabling Mixed OpenMP/MPI Programming on Hybrid CPU/GPU Computing Architecture," in *IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, ser. IPDPSW '12. Washington, D.C., USA: IEEE, 2012, pp. 2369–2377.

[191] K. Hamidouche, J. Falcou, and D. Etiemble, "A Framework for an Automatic Hybrid MPI+OpenMP Code Generation," in *Proceedings of the 19th High Performance Computing Symposia*, ser. HPC '11. San Diego, CA, USA: Society for Computer Simulation International, 2011, pp. 48–55.

[192] M. Steuwer and S. Gorlatch, "SkelCL: Enhancing OpenCL for High-Level Programming of Multi-GPU Systems," in *Parallel Computing Technologies*, ser. LNCS, V. Malyshkin, Ed. Berlin, Germany: Springer Berlin Heidelberg, 2013, vol. 7979, pp. 258–272.

[193] C. Yang, C. Huang, and C. Lin, "Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters," *Computer Physics Communications*, vol. 182, no. 1, p. 266–269, 2011.

[194] M. Howison, E. Bethel, and H. Childs, "Hybrid Parallelism for Volume Rendering on Large-, Multi-, and Many-Core Systems," *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, no. 1, pp. 17–29, 2012.

[195] A.-E. Hugo, A. Guermouche, P.-A. Wacrenier, and R. Namyst, "Composing Multiple StarPU Applications over Heterogeneous Machines: A Supervised Approach," in *IEEE 27th International Parallel and Distributed Processing Symposium Workshops PhD Forum*, ser. IPDPSW '13. Washington, D.C., USA: IEEE, 2013, pp. 1050–1059.

[196] U. Dastgeer, J. Enmyren, and C. W. Kessler, "Auto-tuning SkePU: A Multi-backend Skeleton Programming Framework for multi-GPU Systems," in *Proceedings of the 4th International Workshop on Multicore Software Engineering*, ser. IWMSE '11. New York, NY, USA: ACM, 2011, pp. 25–32.

[197] R. Reyes, I. López-Rodríguez, J. Fumero, and F. de Sande, "accULL: an OpenACC implementation with CUDA and OpenCL support," in *Proceedings of the 18th international Euro-Par conference on Parallel Processing*, ser. Euro-Par '12. Berlin, Germany: Springer Berlin Heidelberg, 2012, pp. 871–882.

[198] N. Farooqui, A. Kerr, G. F. Diamos, S. Yalamanchili, and K. Schwan, "A framework for dynamically instrumenting GPU compute applications within GPU Ocelot," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-4. New York, NY, USA: ACM, 2011, pp. 9:1–9:9.

[199] S. Pai, M. J. Thazhuthaveetil, and R. Govindarajan, "Improving GPGPU Concurrency with Elastic Kernels," *SIGPLAN Not.*, vol. 48, no. 4, pp. 407–418, Mar 2013.

[200] C. Demetrescu, A. V. Goldberg, and D. S. Johnson, Eds., *9th DIMACS Implementation Challenge - Shortest Paths*. Providence, RI, USA: American Mathematical Society, 2006.