Universidad de Valladolid

Escuela de Ingeniería Informática

Trabajo Fin de Grado

Grado en Ingeniería Informática

Herramientas para la Evaluación de Compiladores para OpenACC

Autor:
D. Daniel Barba Gutiérrez

Tutores:
Dr. Diego R. Llanos Ferraris
Dr. Arturo González-Escribano

Agradecimientos

A mi familia, sin cuyo apoyo, paciencia y comprensión este proyecto nunca habría visto la luz.

A mis amigos del GCC (Gran Comando Cafetería), que día a día me han dado la motivación necesaria para nunca tirar la toalla, y que valen mucho más de lo que imaginan.

A mis compañeros y amigos del Grupo Trasgo, que desde el primer día me acogieron como a uno de los suyos y que no han dudado en perder un poco de su tiempo en echarme una mano cuando ha hecho falta.

A Diego y a Arturo, por darme la oportunidad de trabajar con gente excepcional y orientarme durante estos meses.

A José Manuel Marqués, por sus consejos y su ayuda constantes durante estos años.

A la Escuela de Ingeniería Informática en su conjunto, porque después de cuatro años la actitud de todos ha hecho que se convierta en una segunda casa.

Y finalmente, y con un recuerdo muy especial, a Agustín de Dios, que nos dejó antes de tiempo pero a quien siempre recordaremos gracias a su carácter amable y su gran labor docente.

Este proyecto ha sido parcialmente financiado por el MICINN y el programa ERDF de la Unión Europea: proyecto HomProg-HetSys (TIN2014-58876-P), la red CAPAP-H5 (TIN2014-53522-REDT) y el COST Program Action IC1305: Network for Sustainable Ultrascale Computing (NESUS).

Resumen

Este Trabajo de Fin de Grado presenta *TORMENT OpenACC2016*, una herramienta de benchmarking para OpenACC, un nuevo modelo de programación paralela para aceleradores de tipo GPU y Xeon Phi. Existen una serie de compiladores que implementan este estándar, varios de los cuales son de algún modo gratuitos o de código abierto. En paralelo a estos compiladores han aparecido diversas herramientas de benchmarking.

Durante el desarrollo de este proyecto se ha realizado una tarea de investigación del estándar OpenACC y de los compiladores disponibles, utilizando las herramientas existentes. Esto ha servido para obtener una visión de conjunto, así como poder enumerar los pros y los contras de las distintas herramientas con el objetivo de diseñar *TORMENT OpenACC2016* de modo que se diera respuesta a las necesidades encontradas.

Con estas necesidades en mente, se ha diseñado la herramienta con un total de seis benchmarks que utilizan las funcionalidades disponibles y operativas en los compiladores soportados. Estos benchmarks cubren un conjunto de características fundamentales de la computación en GPUs.

Finalmente, se ha desarrollado una métrica denominada *TORMENT_ACC2016* que permite dar un valor numérico para la evaluación del rendimiento obtenido por los pares máquina-compilador estudiados.

Abstract

This Bachelor's thesis presents *TORMENT OpenACC2016*, a benchmarking tool for OpenACC, a new parallel programming model designed for accelerators like GPUs and Xeon Phi. There are a number of compilers implementing the standard, some of which are somehow free or open source. Along with these compilers, several benchmarking tools have also appeared.

During the development of this project it has been carried out a research work on the OpenACC standard and the available compilers, using the existing tools. This has been useful to obtain an overview of the different tools, discovering their pros and cons with the aim of designing *TORMENT OpenACC2016* in such a way that the lacking aspects could be given response.

With these needs in mind, the tool has been designed with six benchmarks using the supported compilers' available functionalities. These benchmarks cover a set of essential characteristics of GPU computation.

Moreover, a metric has been developed, named *TORMENT_ACC2016*, which allows to give a single numeric value as a result of the performance evaluation obtained from the host-compiler pairs in study.

Tabla de Contenidos

Lista de Figuras

Lista de Tablas

Capítulo 1

Introducción

El desarrollo del estándar OpenACC, un nuevo modelo de programación paralela basado en el uso de directivas de compilación sobre código secuencial, ha propiciado la aparición de diferentes compiladores tanto en el ámbito académico como en la industria. Estos compiladores permiten la generación automática de código ejecutable en aceleradores de tipo *GPU* (Graphical Processing Unit) como en las *Xeon PHI* de Intel.

Estas herramientas llevan a cabo en segundos tareas de paralelización de código que llevarían horas o incluso días para hacerse manualmente. No obstante, las técnicas automáticas no siempre son capaces de dar una respuesta óptima a determinados problemas. A pesar de ello, estas herramientas permiten la paralelización de código con una mínima inversión de tiempo, dinero y aprendizaje.

Para poder evaluar el comportamiento del código generado por los distintos compiladores es necesario disponer de una herramienta de benchmarking adecuada tanto al estado de desarrollo de los mismos y del estándar como a las características particulares de la ejecución de código en *GPUs*.

1.1. Contexto e Historia

Las medidas de rendimiento de hardware y sistemas tienen ya mucha historia. A lo largo de los años han ido apareciendo herramientas, a menudo denominadas *Benchmark Suites* que se componen de una serie de programas (los denominados *benchmarks*) y cuyas medidas de tiempo resultan interesantes para la evaluación de diferentes conceptos. A continuación se describen brevemente algunas de estas herramientas [14]:

Livermore Fortran Kernels También conocido como *Livermore Loops* [17] y publicado en 1986, es un programa de benchmarking desarrollado para ser representativo de las partes de Fortran más utilizadas en las aplicaciones más extendidas. La métrica de rendimiento utilizada son los MFLOPS. Este benchmark ha sido traducido a otros lenguajes a parte de Fortran. Cada parte incorpora un kernel matemático diferente.

Results Table

Benchmark	Base						Peak					
	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio
400.perlbench	410	23.8	408	24.0	**408**	**23.9**	353	27.7	**354**	**27.6**	354	27.6
401.bzip2	552	17.5	**552**	**17.5**	552	17.5	**527**	**18.3**	527	18.3	527	18.3
403.gcc	320	25.2	321	25.1	**320**	**25.2**	311	25.9	311	25.9	**311**	**25.9**
429.mcf	**208**	**43.7**	217	42.0	208	43.8	**187**	**48.8**	187	48.8	187	48.7
445.gobmk	**478**	**21.9**	473	22.2	486	21.6	**450**	**23.3**	450	23.3	450	23.3
456.hmmer	213	43.7	**214**	**43.7**	214	43.7	210	44.4	**210**	**44.4**	210	44.4
458.sjeng	501	24.1	502	24.1	**502**	**24.1**	492	24.6	493	24.6	**492**	**24.6**
462.libquantum	**26.1**	**795**	30.7	675	23.1	898	**26.1**	**795**	30.7	675	23.1	898
464.h264ref	**639**	**34.6**	640	34.6	639	34.6	569	38.9	569	38.9	**569**	**38.9**
471.omnetpp	**297**	**21.0**	297	21.0	298	21.0	242	25.8	242	25.9	**242**	**25.8**
473.astar	**334**	**21.0**	334	21.0	336	20.9	393	17.9	392	17.9	**393**	**17.9**
483.xalancbmk	199	34.7	196	35.3	**197**	**35.1**	**208**	**33.1**	208	33.1	209	33.1

Results appear in the order in which they were run. Bold underlined text indicates a median measurement.

Figura 1.1: Fragmento de la hoja de resultados de SPEC CINT2006 en la que se muestra la tabla de resultados

NAS Kernel Benchmark Desarrollado por la NASA [2], su objetivo era analizar el rendimiento de la vectorización en los diferentes sistemas al ejecutar programas de cálculo matemático. Usa como métrica de rendimiento los MFLOPS. Posteriormente fue desarrollada una versión denominada *NAS Parallel benchmarks* [1] y hecha específicamente para computadores altamente paralelos, a diferencia del original que se enfocaba más al uso de vectorización.

LINPACK Un programa de benchmarking [9] que muestra el rendimiento de un sistema al ejecutar una serie de cálculos matemáticos del dominio del álgebra lineal. Los resultados finales se muestran como tiempo de ejecución y MFLOPS. Hace uso de los denominados *BLAS* (Subprogramas Básicos de Álgebra Lineal). Al igual que en el caso de NAS Kernel Benchmark, una versión paralela fue implementada, denominada High Performance Linpack.

PERFECT club Este benchmark [5] se compone de problemas de cómputo de varios dominios, con el objetivo de tener un amplio espectro de problemas susceptibles de ser resueltos mediante ordenadores. PERFECT es un acrónimo de *Performance evaluation of cost-effective transformations*, refiriéndose a las transformaciones realizadas por un compilador para generar código ejecutable en paralelo de un programa secuencial. Sus resultados están dados como tiempo de ejecución y tiempo de CPU.

SPEC CPU Uno de los programas de benchmarking más extendidos y conocidos. Los benchmarks que contiene se van actualizando con las nuevas versiones de SPEC CPU. El resultado de cada benchmark está dado como tiempo de ejecución y se realiza la división entre un tiempo de ejecución de referencia y el tiempo obtenido, lo que da como resultado un ratio denominado *SPECratio*. El resultado final se obtiene como la media geométrica de todos los ratios. Un ejemplo de tabla de resultados de SPEC CPU puede verse en la figura 1.1. El resumen completo se puede encontrar en el Anexo I.

1.2. Motivación

La aparición de un nuevo modelo de programación paralela destinado a aceleradores, tanto *GPUs* como para *Xeon PHI*, supone un nuevo campo para el desarrollo de toda una serie de compiladores y traductores de código cuya finalidad principal es la de paralelizar automáticamente código secuencial destinado a ser ejecutado en los aceleradores señalados. La investigación que se ha realizado ha demostrado que en relación con el estándar OpenACC no existe aún una herramienta de benchmarking que nos de resultados que sean lo suficentemente expresivos. Más adelante se detallará esta afirmación y se procederá a describir las herramientas existentes durante la redacción de este proyecto.

El rendimiento del código que se genera tiene que competir no sólo entre los diferentes compiladores, sino con el código equivalente en *CUDA* u *OpenCL*, así como el código *OpenMP* ejecutado en una *Xeon PHI*. No obstante, OpenACC no aspira a superar el rendimiento de estos códigos hechos a mano, sino que su objetivo fundamental es facilitar la creación de código paralelo destinado a este tipo de aceleradores sin necesidad de dedicar mucho tiempo al aprendizaje de, por ejemplo, *CUDA*. No obstante, uno de los objetivos de las implementaciones del estándar OpenACC es reducir el *overhead* (la penalización en el rendimiento) producido por el conjunto de llamadas a las bibliotecas necesarias para que esta paralelización automática pueda realizarse.

TORMENT OpenACC2016 surge como respuesta a la necesidad de disponer de un conjunto de herramientas para la evaluación de compiladores para OpenACC [21]. En la actualidad, aunque existe un soporte mínimo para el uso de *Xeon PHI* en algún compilador, el escaso soporte en el conjunto de compiladores hace que, por el momento, esta herramienta se centre únicamente en el uso de *GPUs*. Además, dado que *CUDA* se ha convertido en el referente en cuanto a computación paralela en *GPUs*, se ha decidido que la comparación se haga con respecto al rendimiento de código *CUDA*. Es cierto que en el rendimiento del código *CUDA* entra en juego el factor humano, ya que la experiencia y las capacidades de quien lo programe afectarán directamente al resultado. Esto no es algo negativo, ya que también es interesante ver hasta que punto las herramientas automáticas son comparables en rendimiento con códigos hechos a mano.

1.3. Objetivos

Este proyecto tiene como objetivo dar respuesta a las necesidades antes mencionadas. Se intentará obtener una herramienta o *benchmark suite* que utilice uno de los compiladores existentes para compilar la colección de programas de prueba o *benchmarks*, que contendrán diferentes tipos de código paralelizable para, midiendo el tiempo que tarda su ejecución, poder obtener resultados acerca del rendimiento del código que el compilador genera.

Estos resultados serán comparados con el rendimiento de código *CUDA*, de modo que se obtenga una comparativa entre el compilador utilizado y un programa hecho a mano.

Además de los resultados, otro objetivo fundamental del desarrollo de esta herramienta es que, además de ejecutar los diferentes programas de prueba, la propia herramienta se encargue de analizar, procesar y presentar los datos obtenidos, igual que hace *SPEC CPU2006* [26] o similares. Un ejemplo de estos resultados proporcionados por SPEC se puede ver en el Anexo I.

1.4. Estructura de la Memoria

El resto de la memoria se estructura de la siguiente manera:

El capítulo dos presenta el Plan de Desarrollo de Proyecto Software. Incluye el Plan de Proceso, la gestión del mismo, la planificación temporal, el Plan de Riesgos y otros planes similares. Contiene también una estimación del presupuesto para este proyecto.

El capítulo tres trata sobre el estado del arte, expresión proveniente del inglés *state-of-the-art* y que hace referencia a la situación actual del contexto del trabajo desarrollado. En concreto, se introducirá el estándar OpenACC con un breve resumen de historia y sus objetivos fundamentales. Se continuará hablando del estado o de la madurez en la que se encuentra el estándar. Después se comentará en qué situación se está actualmente en relación con las diferentes implementaciones del estándar y las herramientas que han ido surgiendo. Finalmente se señalarán los problemas que se han detectado en el tiempo que ha durado la fase de investigación previa al comienzo del desarrollo de *TORMENT OpenACC2016*.

El capítulo cuatro contiene el análisis de requisitos extraído del trabajo previo de investigación que se ha realizado, gran parte de lo cual figura en el capítulo tercero.

El capítulo cinco trata sobre el diseño de *TORMENT OpenACC2016*. Presenta el diseño de los scripts y del programa para dar respuesta a los requisitos establecidos. Además, enumera y describe los programas de prueba o *benchmarks* que se han elegido para formar parte de *TORMENT OpenACC2016*, además de la motivación que ha llevado a su elección. También se explica el diseño de la métrica *TORMENT_ACC2016* que acompaña al desarrollo de *TORMENT OpenACC2016*.

El capítulo seis detalla el proceso de implementación, con información y detalles del resultado final obtenido.

En el capítulo siete se aplica la herramienta a la investigación sobre el rendimiento del código generado por los compiladores de OpenACC, explicando la máquina de referencia usada y las máquinas de estudio junto con los resúmenes de resultados obtenidos con *TORMENT OpenACC2016*.

Finalmente, el capítulo ocho expone las conclusiones obtenidas y detalla ideas de trabajo futuro, concluyendo este Trabajo de Fin de Grado.

Capítulo 2

Plan de Proyecto

2.1. Resumen del Proyecto

A continuación se detalla el Plan de Desarrollo de Proyecto Software de *TORMENT OpenACC2016*, objeto del presente Trabajo de Fin de Grado.

2.1.1. Propósito, Alcance y Objetivos

El objetivo de este proyecto es obtener una aplicación, denominada *TORMENT OpenACC2016* (*Trasgo BenchmARkinG and Evaluation Tool for OpenACC*). Esta aplicación permitirá obtener una evaluación del rendimiento del código generado por diferentes compiladores de OpenACC ejecutado en distintos aceleradores de tipo GPU.

La herramienta debe dar respuesta a las necesidades de comparación de códigos generados por distintos compiladores para poder analizar sus fortalezas y debilidades. La herramienta generará informes según los datos obtenidos.

Los requisitos que complementan esta información serán desarrollados en profundidad en el artefacto "Modelo de Análisis".

2.1.2. Presupuesto

Con la planificación que se detalla posteriormente, el presente proyecto se desarrolla aproximadamente en un plazo de 10 meses en régimen de media jornada. Teniendo en cuenta que el salario anual para un Ingeniero Informático ronda los 22.000€, más unos 7.300€ de costes sociales, se estima el presupuesto de este proyecto en 12.210€ más IVA. No existe ningún coste derivado de amortización de equipos o licencias.

2.1.3. Suposiciones y Restricciones

Deberá disponerse de una versión funcional de la aplicación el día 15 de Mayo. Esto se debe a la fecha límite para la presentación de artículos para las Jornadas de Paralelismo.

2.1.4. Definiciones y Acrónimos

A continuación se listan acrónimos o definiciones que aparecen en el documento.

Cluster Conjunto de ordenadores comunicados entre sí.

GPU Graphics Processing Unit.

TORMENT TrasgO benchmaRking and peRformance EvaluatioN Tool.

OpenACC Acrónimo de *Open Accelerators*, se trata de un estándar de programación de computación paralela en sistemas heterogéneos.

Pragma En programación, un *pragma* es una directiva de compilación. Indica al compilador cómo procesar algo.

RUP Acrónimo de *Rational Unified Process*. Un proceso de desarrollo de software ampliamente utilizado.

2.1.5. Artefactos de Proyecto

Al utilizar para el desarrollo del proyecto la metodología RUP (*Rational Unified Process*) [13], aunque adaptada a la elaboración de un Trabajo de Fin de Grado como resultado de una necesidad de investigación, deben elaborarse los siguientes artefactos para las fechas establecidas en cada uno:

Plan de Desarrollo de Proyecto Software 31 de Marzo de 2016

Análisis 1 de Abril de 2016

Diseño 8 de Abril de 2016

Implementación 15 de Mayo de 2016

2.1.6. Evolución del Plan

El Plan de Desarrollo de Proyecto describe la planificación del proyecto en detalle. Es un documento que evoluciona conforme lo hace el propio proyecto. El presente documento será revisado semanalmente para atender las posibles desviaciones y actualizarlo en consecuencia.

Resumen de Planificación

Durante el desarrollo de la fase de elaboración, concretamente durante el desarrollo de los diferentes diagramas UML, se hizo patente el hecho de que en este caso concreto, los diagramas habituales no eran suficientemente expresivos o forzaban a la introducción de conceptos que no se ajustaban a la realidad.

Debido a esto, y tras la consulta con varios expertos en la materia, se tomo la decisión de excluir los diagramas habituales y sustituirlos por diagramas de actividad.

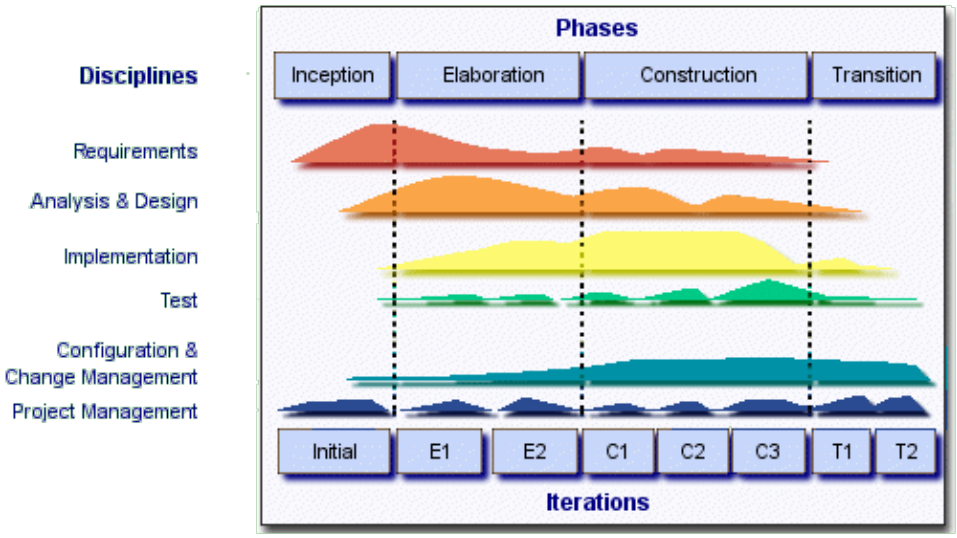


Figura 2.1: Fases de RUP

2.2. Plan de Proceso

Para el desarrollo de este proyecto se ha optado por utilizar la metodología de RUP (Rational Unified Process), adaptado a las necesidades propias de un Trabajo de Fin de Grado resultado de una necesidad de investigación. Las características fundamentales de RUP son:

Dirigido por casos de uso: Estableciendo la funcionalidad como guía para el análisis y diseño.

Centrado en la arquitectura: Se parte de una visión global para ir refinando hasta definir los componentes básicos.

Iterativo e incremental: El sistema se divide en incrementos para ir refinando de forma iterativa.

Enfocado en riesgos: En el orden se tienen en consideración los riesgos más importantes en primer lugar.

Uso de UML: para la descripción del sistema, aprovechando que UML es independiente del lenguaje de programación a utilizar.

2.2.1. Ciclo de Vida del Proyecto

El ciclo de vida en RUP se compone de cuatro fases tal y como de muestra en la figura 2.1: Inicio, Elaboración, Construcción y Transición o Cierre. Al adaptarlo a las necesidades de un Trabajo de Fin de Grado originado como una necesidad de investigación se ha decidido añadir una fase cero, denominada Trabajo Previo.

Trabajo Previo: Incluye todo el trabajo necesario relativo al aprendizaje de los elementos del contexto, la investigación del estado actual y el análisis de las necesidades encontradas.

Inicio Establecimiento de la visión general del sistema, incluyendo alcance y límites, estimación de coste y calendarización y análisis de riesgos.

Elaboración Elaboración de los modelos de análisis y diseño, estableciendo una arquitectura base de acuerdo a los casos de uso, despejando aquellos con mayor incertidumbre.

Construcción Obtención de versiones funcionales de forma incremental e iterativa, completando análisis, diseño, implementación y pruebas de todos los casos de uso requeridos.

Transición o cierre Se valida el sistema y se elabora la documentación necesaria para la entrega.

Estas fases se dividen en iteraciones. Se ha decidido que la fase de Construcción se realice en dos iteraciones. La finalidad de las iteraciones consiste en poder descomponer los resultados que se van obteniendo, de forma que se pueda ir refinando todo el trabajo realizado.

2.3. Gestión del Proceso

A continuación se detallan los diferentes planes involucrado en la Gestión del Proceso.

2.3.1. Plan de Puesta en Marcha

El establecimiento del esfuerzo y la duración del proyecto se realizará por estimación, teniendo como referencia la experiencia propia y otros proyectos de características similares.

El desarrollo del proyecto es individual, por lo que no es necesario la gestión de comunicación y la disponibilidad es total desde el comienzo.

Para la correcta finalización del proyecto son necesarias las siguientes habilidades:

- Utilización de sistemas GNU/Linux.
- Conocimiento de lenguajes C, CUDA y HTML.
- Conocimiento del estándar OpenACC y los diferentes pragmas disponibles para la paralelización de código.
- Conocimientos de programación paralela en GPUs.
- Conocimiento del funcionamiento típico de los compiladores, especialmente lo relativo a *flags* de optimización.
- Conocimiento de herramientas de debugging como GDB.

- Capacidad para escribir *scripts Bash*.
- Utilización de GNUplot para realización de gráficas.

No será necesaria ninguna adquisición para la realización de este proyecto ya que se dispone de los recursos de la **Escuela de Ingeniería Informática** y, en particular, los del **Grupo Trasgo**, entre los que se encuentran el *cluster* del grupo de investigación con máquinas en las que están instaladas varias GPUs de Nvidia.

2.3.2. Plan de Trabajo

En esta sección se detallan las diferentes actividades en las que se divide el proyecto, así como la calendarización de las mismas.

Actividades

ID: 02	**Aprendizaje OpenACC**
Predecesoras:	-
Duración:	20 días
Durante el desarrollo de esta actividad, se procederá a la lectura de las partes más relevantes del estándar, en particular lo relativo a los *pragmas* existentes y su funcionamiento. Se utilizará como herramienta el *Nvidia OpenACC toolkit* que incluye un compilador y se acompaña de fragmentos de ejemplo.	

ID: 03	**Recopilación de Compiladores**
Predecesoras:	-
Duración:	10 días
Se realizará una búsqueda de los compiladores de OpenACC existentes y disponibles para uso académico. Incluye la lectura de la documentación y su instalación en el *cluster* del grupo de investigación.	

ID: 04	**Recopilación de Benchmarks**
Predecesoras:	-
Duración:	5 días
Se investigará acerca de los diferentes intentos de evaluación del estándar OpenACC o de sus implementaciones, intentando hacer una recopilación de programas de prueba que puedan usarse.	

ID: 05	**Compilación de Benchmarks**
Predecesoras:	03, 04
Duración:	15 días
Una vez recopilados los benchmarks se procederá a su compilación con los diferentes compiladores disponibles, incluyendo la detección de errores para los que se decidirá entre solucionar los problemas o excluir el benchmark de las pruebas.	

ID: 06	Experimentación
Predecesoras:	05
Duración:	5 días
Se prepararán scripts de ejecución para lanzar los benchmarks y obtener resultados, que serán almacenados para su análisis.	

ID: 07	Detección de carencias y defectos
Predecesoras:	05
Duración:	15 días
Una vez compilados los benchmarks y en paralelo a la experimentación deberán ir observándose carencias o defectos en los mismos con el objetivo de descubrir las necesidades en la evaluación de OpenACC y sus implementaciones que tratará de suplir el proyecto.	

ID: 08	Selección de Benchmarks
Predecesoras:	07
Duración:	10 días
A la vista de los resultados obtenidos durante la experimentación, las necesidades detectadas y la madurez del estándar y de sus implementaciones, se procederá a la selección o al diseño de un conjunto de benchmarks que permitan evaluar el rendimiento del código generado por los diferentes compiladores.	

ID: 10	Planificación de Riesgos
Predecesoras:	-
Duración:	4 días
Se realizará la planificación de riesgos del proyecto, describiendo los riesgos detectados, planes de reducción y de contingencia.	

ID: 11	Definición y Secuenciación de Actividades
Predecesoras:	-
Duración:	5 días
Definición de las actividades necesarias para la consecución de los objetivos del proyecto, incluida la secuenciación de las mismas para disponer de la información necesaria de cara a la planificación temporal del proyecto.	

ID: 12	Calendarización
Predecesoras:	11
Duración:	2 días
Con las actividades definidas y secuenciadas se procederá a la estimación temporal de cada una de ellas, lo que permitirá la elaboración de los diagramas de Gantt que muestren la planificación temporal del proyecto.	

ID: 13	Control de Versiones
Predecesoras:	-
Duración:	1 día
Se establecerá el sistema de Control de Versiones del proyecto con un repositorio online en el que se almacenarán las diferentes versiones del proyecto, tanto documentación como código.	

ID: 14	Redacción del Plan de Desarrollo de Proyecto
Predecesoras:	12
Duración:	8 días
Con la información sobre actividades, calendarización y riesgos, se está en disposición de redactar el Plan de Desarrollo de Proyecto, con los diferentes planes necesarios.	

ID: 16	Requisitos
Predecesoras:	-
Duración:	4 días
Se definirán los requisitos del proyecto de acuerdo a la visión obtenida en la fase de Trabajo Previo.	

ID: 17	Casos de Uso
Predecesoras:	16
Duración:	5 días
Con los requisitos definidos, se descubrirán los diferentes casos de uso y se elaborará el diagrama de casos de uso y las definiciones de los mismos.	

ID: 18	Modelo de Dominio
Predecesoras:	17
Duración:	5 días
Se modelará el dominio del problema mediante UML.	

ID: 19	Diagramas de Secuencia
Predecesoras:	18
Duración:	10 días
Mediante la información obtenida de los casos de uso y del modelo de dominio se procederá a la realización de los diagramas de secuencia.	

ID: 20	Diseño de Scripts
Predecesoras:	19
Duración:	4 días
Se elaborará un diseño de los scripts bash necesarios para el funcionamiento del sistema.	

ID: 21	Arquitectura del Sistema
Predecesoras:	19
Duración:	20 días
Se elaborará toda la documentación necesaria para la definición de la arquitectura del sistema, incluyendo lo habitual en la etapa de diseño.	

ID: 24	Implementación del programa principal
Predecesoras:	-
Duración:	10 días
Se elaborará el código del programa principal, encargado de hacer las veces de punto de entrada y distribuidor de la ejecución de los benchmarks.	

ID: 25	Implementación de los benchmarks
Predecesoras:	-
Duración:	12 días
Codificación de los diferentes benchmarks elegidos tanto en OpenACC como en CUDA.	

ID: 26	Elaboración de los Scripts
Predecesoras:	-
Duración:	8 días
En base al diseño efectuado en la fase anterior, se escribirán los diferentes script necesarios que serán el nexo de unión entre el usuario y el programa.	

ID: 28	Refinamiento del programa principal
Predecesoras:	24, 25, 26
Duración:	6 días
Se efectuarán las modificaciones que se decidan al finalizar la primera iteración.	

ID: 29	Refinamiento de los benchmarks
Predecesoras:	24, 25, 26
Duración:	4 días
Se efectuarán las modificaciones que se decidan al finalizar la primera iteración.	

ID: 30	Almacenamiento de resultados en ficheros
Predecesoras:	28, 29
Duración:	2 días
Se decidirá la mejor manera de almacenar los resultados obtenidos para su posterior tratamiento.	

ID: 31	Scripts gnuplot
Predecesoras:	30
Duración:	4 días
Se procederá a escribir los scripts gnuplot necesarios para obtener las gráficas que muestren los resultados obtenidos	

ID: 32	Resultados en ficheros HTML
Predecesoras:	30
Duración:	4 días
Se elaborará el código necesario para la obtención de ficheros HTML de resumen de los resultados obtenidos.	

ID: 34	Plan de Pruebas
Predecesoras:	-
Duración:	5 días
Implementación de las pruebas necesarias para asegurar la corrección de resultados en los benchmarks	

ID: 35	Verificación
Predecesoras:	34
Duración:	2 días
Comprobación de que la herramienta se ejecuta correctamente en varias máquinas.	

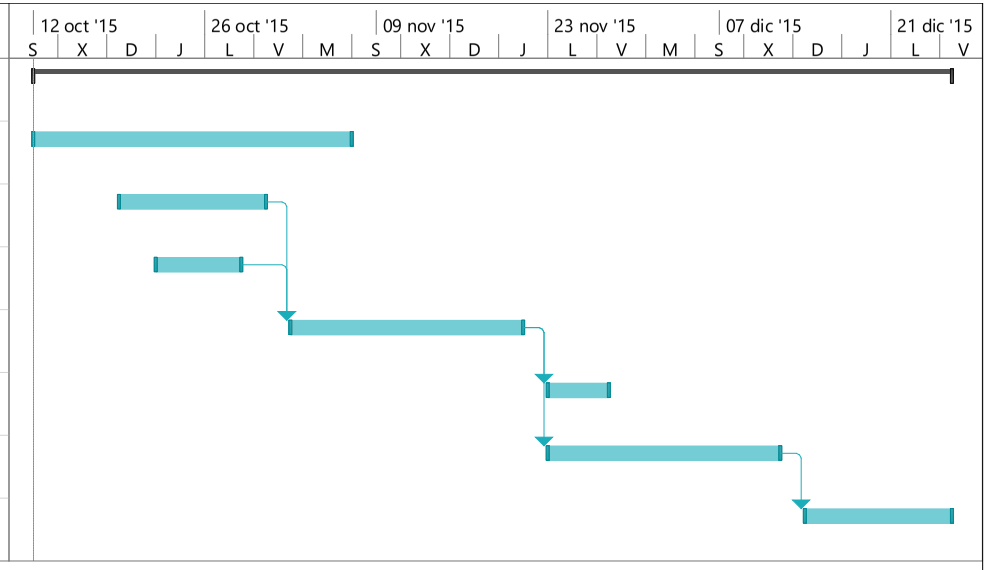
ID: 36	Manual de Usuario
Predecesoras:	-
Duración:	2 días
Elaboración del Manual de Usuario del sistema.	

ID: 37	Manual de Instalación
Predecesoras:	-
Duración:	2 días
Elaboración del Manual de Instalación del sistema.	

Calendarización

En las siguientes 5 páginas se incluyen los diagramas de Gantt de las diferentes fases del proyecto: Fase Previa, Inicio, Elaboración, Construcción y Transición. Estos diagramas de Gantt detallan la calendarización de las actividades que conforman el proyecto y muestran, además, las relaciones de dependencias entre ellas.

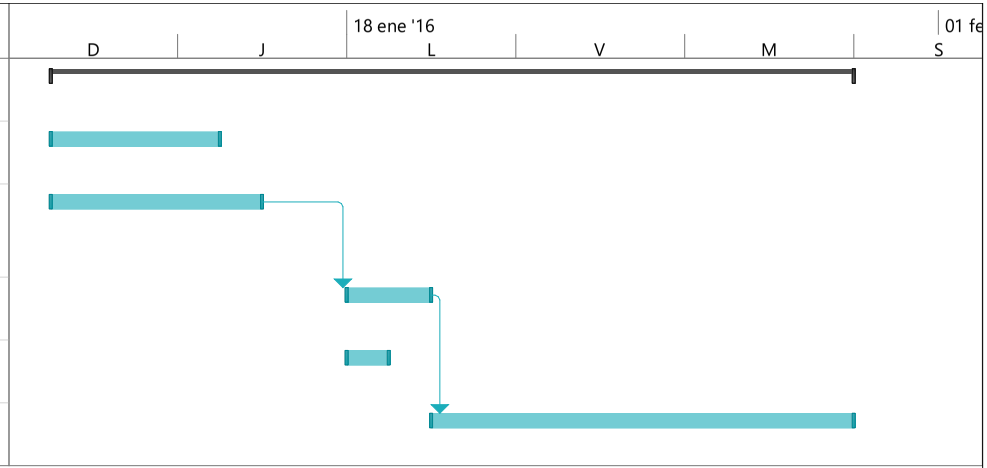
Id	Nombre de tarea	Duración	Comienzo	Fin
1	**Trabajo Previo**	**55 días**	**lun 12/10/15**	**vie 25/12/15**
2	Aprendizaje OpenACC	20 días	lun 12/10/15	vie 06/11/15
3	Recopilación de compiladores	10 días	lun 19/10/15	vie 30/10/15
4	Recopilación de benchmarks	5 días	jue 22/10/15	mié 28/10/15
5	Compilación de benchmarks	15 días	lun 02/11/15	vie 20/11/15
6	Experimentación	5 días	lun 23/11/15	vie 27/11/15
7	Detección de Carencias y Defectos	15 días	lun 23/11/15	vie 11/12/15
8	Selección de Benchmarks	10 días	lun 14/12/15	vie 25/12/15



Proyecto: TORMENT OpenACC

Tarea	Resumen inactivo	Tareas externas	
División	Tarea manual	Hito externo	
Hito	solo duración	Fecha límite	
Resumen	Informe de resumen manual	Progreso	
Resumen del proyecto	Resumen manual	Progreso manual	
Tarea inactiva	solo el comienzo		
Hito inactivo	solo fin		

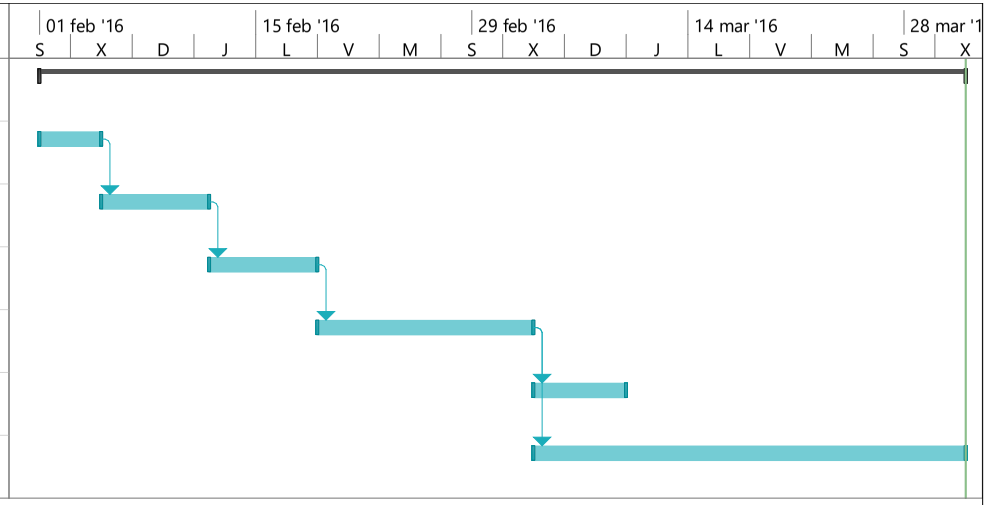
Id	Nombre de tarea	Duración	Comienzo	Fin
9	**Inicio**	**15 días**	**lun 11/01/16**	**vie 29/01/16**
10	Planificación de Riesgos	4 días	lun 11/01/16	jue 14/01/16
11	Definición y Secuenciación de Actividades	5 días	lun 11/01/16	vie 15/01/16
12	Calendarización	2 días	lun 18/01/16	mar 19/01/16
13	Control de Versiones	1 día	lun 18/01/16	lun 18/01/16
14	Redacción Plan de Desarrollo de Proyecto	8 días	mié 20/01/16	vie 29/01/16



Proyecto: TORMENT OpenACC

Tarea	Resumen inactivo	Tareas externas
División	Tarea manual	Hito externo
Hito	solo duración	Fecha límite
Resumen	Informe de resumen manual	Progreso
Resumen del proyecto	Resumen manual	Progreso manual
Tarea inactiva	solo el comienzo	
Hito inactivo	solo fin	

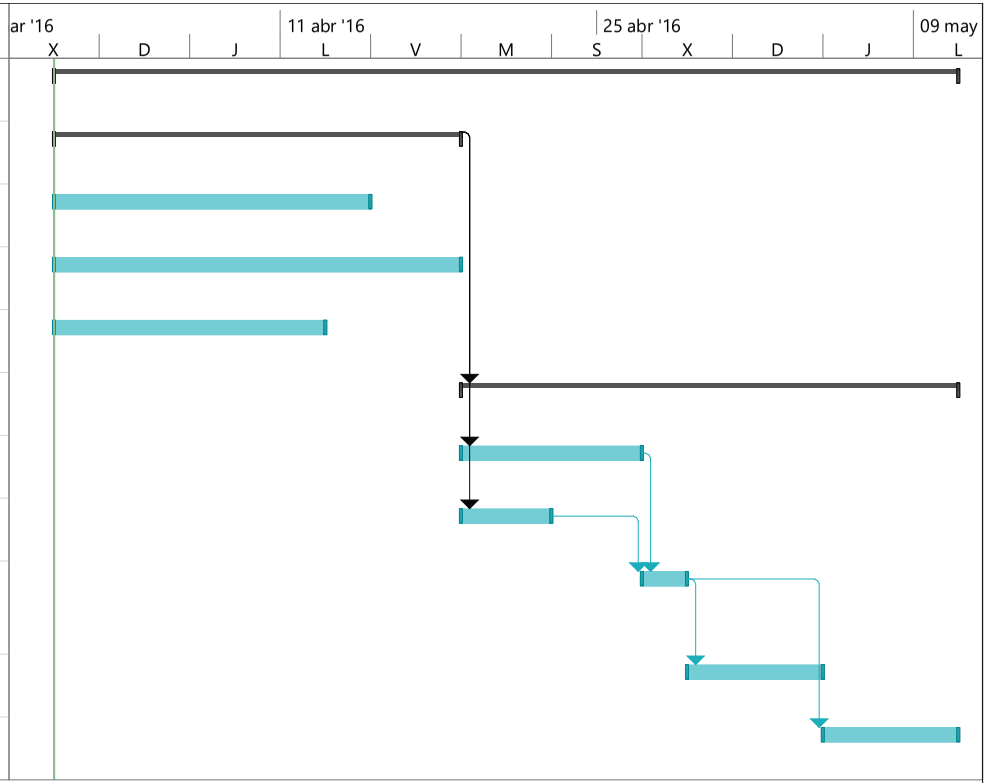
Id	Nombre de tarea	Duración	Comienzo	Fin
15	**Elaboración**	**44 días**	**lun 01/02/16**	**jue 31/03/16**
16	Requisitos	4 días	lun 01/02/16	jue 04/02/16
17	Casos de Uso	5 días	vie 05/02/16	jue 11/02/16
18	Modelo de Dominio	5 días	vie 12/02/16	jue 18/02/16
19	Diagramas de Secuencia	10 días	vie 19/02/16	jue 03/03/16
20	Diseño de Scripts	4 días	vie 04/03/16	mié 09/03/16
21	Arquitectura del Sistema	20 días	vie 04/03/16	jue 31/03/16



Proyecto: TORMENT OpenACC

Tarea	Resumen inactivo	Tareas externas
División	Tarea manual	Hito externo
Hito	solo duración	Fecha límite
Resumen	Informe de resumen manual	Progreso
Resumen del proyecto	Resumen manual	Progreso manual
Tarea inactiva	solo el comienzo	
Hito inactivo	solo fin	

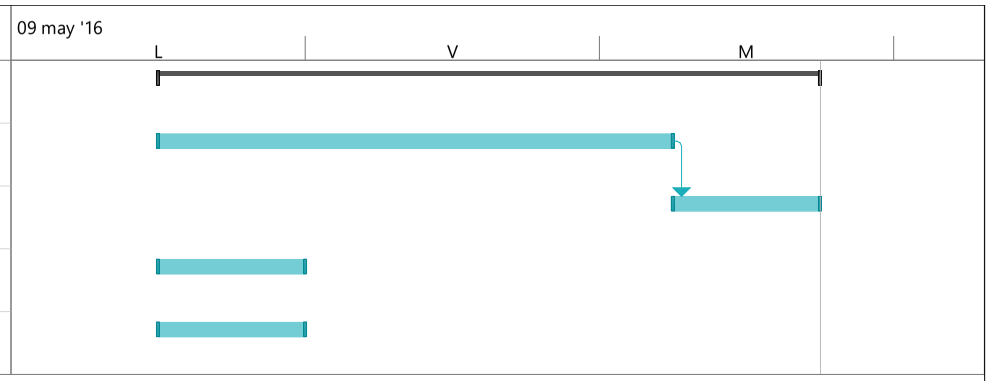
Id	Nombre de tarea	Duración	Comienzo	Fin
22	**Construcción**	**28 días**	**vie 01/04/16**	**mar 10/05/16**
23	**Iteracion 1**	**12 días**	**vie 01/04/16**	**lun 18/04/16**
24	Implementación del programa principal	10 días	vie 01/04/16	jue 14/04/16
25	Implementación de los benchmarks	12 días	vie 01/04/16	lun 18/04/16
26	Elaboración de los Scripts	8 días	vie 01/04/16	mar 12/04/16
27	**Iteracion 2**	**16 días**	**mar 19/04/16**	**mar 10/05/16**
28	Refinamiento del programa principal	6 días	mar 19/04/16	mar 26/04/16
29	Refinamiento de los benchmarks	4 días	mar 19/04/16	vie 22/04/16
30	Almacenamiento de resultados en ficheros	2 días	mié 27/04/16	jue 28/04/16
31	Scripts gnuplot	4 días	vie 29/04/16	mié 04/05/16
32	Resultados en ficheros HTML	4 días	jue 05/05/16	mar 10/05/16



Proyecto: TORMENT OpenACC

Tarea	Resumen inactivo	Tareas externas
División	Tarea manual	Hito externo
Hito	solo duración	Fecha límite
Resumen	Informe de resumen manual	Progreso
Resumen del proyecto	Resumen manual	Progreso manual
Tarea inactiva	solo el comienzo	
Hito inactivo	solo fin	

Id	Nombre de tarea	Duración	Comienzo	Fin
33	**Transición**	**7 días**	**mié 11/05/16**	**jue 19/05/16**
34	Plan de pruebas	5 días	mié 11/05/16	mar 17/05/16
35	Verificación	2 días	mié 18/05/16	jue 19/05/16
36	Manual de Usuario	2 días	mié 11/05/16	jue 12/05/16
37	Maual de Instalación	2 días	mié 11/05/16	jue 12/05/16



Proyecto: TORMENT OpenACC

Tarea	Resumen inactivo	Tareas externas
División	Tarea manual	Hito externo
Hito	solo duración	Fecha límite
Resumen	Informe de resumen manual	Progreso
Resumen del proyecto	Resumen manual	Progreso manual
Tarea inactiva	solo el comienzo	
Hito inactivo	solo fin	

2.3.3. Plan de Control

En este plan se detallan las acciones a realizar respecto al control en el desarrollo del proyecto.

Gestión de Requisitos

En caso de proponerse modificaciones a los requisitos una vez definidos y con el proyecto en curso, se efectuarán las siguientes tareas:

1. Analizar y priorizar los cambios
2. Analizar las consecuencias de la aplicación de los cambios
3. Incorporar los cambios al plan
4. Evaluar posibles nuevos riesgos, debido a los cambios introducidos.
5. Replanificar la calendarización si es necesario.

Control de Calendario

El control del cumplimiento del calendario planificado se basa en el establecimiento de hitos internos en los que realizar una labor de monitorización con el objetivo de detectar desviaciones respecto a lo planificado y poder efectuar acciones correctivas.

Estos hitos están únicamente pensados como herramienta para el desarrollo del proyecto.

2.3.4. Plan de Gestión de Riesgos

A continuación se exponen tanto los riesgos detectados, con una descripción de los mismos, su impacto y probabilidad, como la matriz de exposición mostrada en la tabla 2.1. Con ambos datos se obtiene la exposición al riesgo.

Junto a cada riesgo se detallan también los planes de reducción y contingencia.

R01 - Cluster del grupo averiado/inaccesible

El desarrollo del proyecto se realiza utilizando los recursos de la Escuela de Ingeniería Informática y, especialmente, los del Grupo Trasgo. En el caso de que el cluster resultará inaccesible se dejaría de tener acceso a máquinas en las que ya están instalados y configurados los diferentes compiladores, y en las que se dispone de aceleradores GPU necesarios.

Impacto Catastrófico

Probabilidad 1 %

Exposición Baja

Tabla 2.1: Matriz Impacto/Probabilidad

Impacto\ Probabilidad	Muy Alto	Alto	Medio	Bajo	Muy Bajo
Catastrófico	Alto	Alto	Moderado	Moderado	Bajo
Crítico	Alto	Alto	Moderado	Bajo	Ninguno
Marginal	Moderado	Moderado	Bajo	Ninguno	Ninguno
Despreciable	Moderado	Bajo	Bajo	Ninguno	Ninguno

Plan de Protección Ajustarse al Plan de Gestión de Configuraciones de modo que el control de versiones mantenga seguras las actualizaciones y el trabajo que se vaya realizando.

Plan de Contingencia Dado que todo el trabajo estará guardado en el repositorio, se podrá continuar el trabajo en otras máquinas.

R02 - Pérdida de datos

Pérdida de documentos o código con la consecuente pérdida del tiempo invertido.

Impacto Catastrófico

Probabilidad 5 %

Exposición Moderado

Plan de Protección Utilizar constantemente el control de versiones para mantener la evolución de los documentos o código en lugar seguro.

Plan de Contingencia En caso de pérdida de datos, habrá que evaluar el impacto sobre el proyecto y, en consecuencia, hacer una replanificación tan extensa como sea necesario.

R03 - Enfermedad

No se puede continuar con el trabajo según lo planificado debido a enfermedad.

Impacto Marginal

Probabilidad 5 %

Exposición Ninguno

Plan de Protección N/A

Plan de Contingencia Evaluar el tiempo perdido y replanificar según sea necesario.

R04 - Cambios en Requisitos

Durante la evolución del proyecto se ha encontrado un nuevo requisito que debe incorporarse al mismo.

Impacto Crítico

Probabilidad 5 %

Exposición Bajo

Plan de Protección Efectuar un análisis de requisitos exhaustivo.

Plan de Contingencia Cumplir con lo estipulado en el plan de control.

R05 - Fallo de Calendarización

La planificación temporal demuestra ser insuficiente para cumplir con los plazos previstos.

Impacto Crítico

Probabilidad 20 %

Exposición Alto

Plan de Protección Tratar de ser realista en la planificación y cumplir con el trabajo previsto en cada momento.

Plan de Contingencia Cumplir con lo estipulado en el plan de control.

R06 - Diseño Erroneo

El diseño que ha sido elaborado demuestra no ser el correcto.

Impacto Crítico

Probabilidad 10 %

Exposición Moderado

Plan de Protección Ajustarse a lo obtenido en la fase de análisis y tener especial cuidado con cualquier duda que se presente en fase de diseño.

Plan de Contingencia Corregir el defecto, incluyendo una replanificación si es necesario.

R07 - Tiempo de aprendizaje de las tecnologías

El aprendizaje del funcionamiento de OpenACC y de las peculiaridades de las diferentes implementaciones es mayor del pensado al realizar la planificación.

Impacto Crítico

Probabilidad 5 %

Exposición Bajo

Plan de Protección N/A

Plan de Contingencia Replanificar en función del retraso producido.

R08 - Problemas con el software de terceros

La utilización de compiladores o implementaciones de benchmarks realizados por personas ajenas al proyecto presenta problemas no previstos en el desarrollo del mismo.

Impacto Crítico

Probabilidad 10 %

Exposición Moderado

Plan de Protección N/A

Plan de Contingencia Tratar de solucionar el problema y, si no fuera posible, tratar de sortearlo. Como último recurso, considerar la exclusión de la funcionalidad que presente el problema.

R09 - Fallos en implementación

El software desarrollado presenta bugs.

Impacto Crítico

Probabilidad 20 %

Exposición Alto

Plan de Protección Utilizar buenas prácticas durante la codificación, manteniendo un código fácil de leer y ordenado, menos propenso a errores.

Plan de Contingencia Corregir los bugs y replanificar en caso necesario.

2.4. Planes de Procesos de Soporte

En esta sección se detallan los planes de soporte para el proyecto.

2.4.1. Plan de Gestión de Configuraciones

Se dispondrá de un sistema de control de versiones online para toda la documentación y el código del proyecto. Se utilizará tecnología GIT mediante un repositorio privado alojado en BitBucket.

Revisiones de Progreso del Proyecto

Semanalmente se analizará el cumplimiento de la planificación temporal, actualizando el Plan de Desarrollo de Proyecto según sea necesario.

Capítulo 3

Estado del Arte

En este capítulo se detalla el Estado del Arte entorno a OpenACC y el análisis de rendimiento de código OpenACC.

En primer lugar se comentará brevemente qué es OpenACC, su origen, similitudes con otros modelos de programación y algún ejemplo. Posteriormente se hablará de la madurez del estándar OpenACC, las empresas que forman parte del comité y las implementaciones del mismo. Más adelante se explicará la situación actual en cuánto al estado de los compiladores y de las herramientas de benchmarking. Para terminar, se expondrán los problemas detectados en las herramientas existentes.

3.1. OpenACC

OpenACC es un estándar abierto que define una colección de directivas de compilación, también conocidas como *pragmas*, que permiten la ejecución de fragmentos de código en aceleradoras de tipo *GPU* o *Xeon Phi*. El objetivo de OpenACC es reducir tanto el tiempo de aprendizaje como el de paralelización de código secuencial antiguo de forma que permanezca portable [19]. Esto se debe a que en muchos ámbitos científicos, muchas veces la paralelización de código antiguo es difícil. A veces no se tienen los conocimientos necesarios (*CUDA*) o, si se tienen, no se tiene tiempo para hacerlo.

La especificación de OpenACC está, en el momento de la escritura de este documento, en su versión 2.5. En su origen, OpenACC fue fundado por Nvidia, CRAY, CAPS y PGI. Actualmente CAPS ha desaparecido, aparentemente comprada por Exxact Coporation, y PGI (*The Portland Group*) ha sido adquirido por la propia Nvidia.

OpenACC posee muchos elementos comunes con OpenMP, dado que ambos son modelos de programación que usan directivas de compilación. Pueden verse unos ejemplos de fragmentos de código para la inicialización de un array de enteros en las figuras 3.1 y 3.2.

Como se puede observar, la forma de usar estos modelos de programación es muy similar de cara al programador. Estos pragmas permiten que los compiladores modifiquen el código, generando código paralelo adecuado al dispositivo de destino. En el caso de OpenMP, el código que se genera es un código multi-hilo, usando el

```
1 int array[ELEMENTOS];
2 #pragma omp parallel for private(i) firstprivate (array)
3 for(int i = 0; i < ELEMENTOS; i++){
4     array[i] = i;
5 }
```

Figura 3.1: Un ejemplo de fragmento de código usando OpenMP

```
1 int array[ELEMENTOS];
2 #pragma acc parallel loop private(i) firstprivate (array)
3 for(int i = 0; i < ELEMENTOS; i++){
4     array[i] = i;
5 }
```

Figura 3.2: Un ejemplo de fragmento de código usando OpenACC

modelo fork-join. En cambio, OpenACC genera kernels que serán lanzados en un acelerador, ya sea GPU o Xeon PHI. Cuando estos kernels terminen su ejecución, el control se devuelve a la CPU.

El estándar OpenACC define una serie de directivas con diferentes cláusulas para cada una de ellas. En general, la sintaxis en C es de la forma **#pragma acc** *directiva [cláusulas]*. Las directivas más comúnmente usadas son las siguientes:

- #pragma acc parallel: Crea una región en la que se lanzarán una serie de *gangs* en paralelo, cada uno de los cuales tendrá un número de *workers* que a su vez tienen un *vector* u operaciones SIMD.
- #pragma acc loop: Se aplica a un bucle o una serie de bucles anidados para distribuir las iteraciones de acuerdo al nivel de paralelismo que se utilice. Puede fusionarse con #pragma acc parallel en un #pragma acc parallel loop.
- #pragma acc kernels: Se aplica entorno a un bucle o bucles para que sean ejecutados en el acelerador, dejando la especificación del paralelismo al compilador.
- #pragma acc data: Define una región mediante llaves. Al entrar en la región se copian al dispositivo los datos que se indiquen. Al salir de la región se copian al host los datos que se especifiquen. Dentro de esta región, el dispositivo puede acceder a estos datos en su memoria.

OpenACC define tres niveles de paralelismo en lo que denomina *gangs, workers y vector length*. El nivel más grueso lo forman los *gangs*. Una región paralela se caracteriza por el lanzamiento de un número de *gangs* en paralelo. En términos de CUDA, los gangs se corresponden con los bloques de hilos en un grid de un kernel. El siguiente nivel de paralelismo se asocia con los *workers*, que en un kernel CUDA coincidiría con la dimensión vertical de un bloque bidimensional. Finalmente, el nivel más fino de paralelismo es para el denominado *vector length*. En un kernel CUDA, el *vector length* se correspondería con la dimensión horizontal de un bloque bidimensional.

En cualquier caso, según el estándar, la definición concreta de los diferentes niveles de paralelismo corresponde a cada implementación, por lo que puede variar el comportamiento dependiendo del compilador utilizado.

3.2. Madurez del Estándar

En el momento de la escritura de este documento, OpenACC se encuentra en su versión 2.5 [20]. Los miembros del estándar han crecido e incluyen entidades tanto académicas como industriales, como por ejemplo el *Oak Ridge National Laboratory*, la Universidad de Houston, AMD o el *Edinburgh Parallel Computing Centre* (*EPCC*). El comité de responsables está formado por gente de Nvidia, *Oak Ridge Laboratory*, CRAY y AMD.

Hay una amplia variedad de compiladores que soportan OpenACC. El compilador de PGI es uno de ellos (en realidad, perteneciente a Nvidia desde hace algún tiempo). Este compilador se distribuye como parte del denominado *Nvidia OpenACC toolkit*, con una licencia gratuita de 90 días y la posibilidad de adquirir una licencia anual renovable para investigadores, o bien una licencia comercial que permite el uso de los programas compilados en otras máquinas. CAPS, uno de los fundadores de OpenACC que se dedicaba a proveer software y servicios para la comunidad HPC (*High Performance Computing*), había desarrollado también su propio compilador. Lamentablemente, desde que la compañía fue vendida, este compilador ya no puede ser adquirido, de acuerdo a las informaciones recibidas por uno de los empleados de Exxact Corporation, la empresa que compró CAPS. CRAY Inc tiene otro de los compiladores de OpenACC. CRAY es una empresa que se dedica a comercializar supercomputadores y, según parece, la única manera de conseguir su compilador es mediante la adquisición de una de sus máquinas. Para terminar con las empresas que desarrollan compiladores tenemos a Pathscale Inc, dedicados a la comercialización de software multicore y compiladores. Su compilador ENZO tiene soporte para OpenACC pero, por desgracia, al ser contactados para ver las posibilidades de adquirir una licencia se negaron rotundamente a que su software fuera evaluado junto con otros compiladores.

En el terreno académico hay un par de intentos de desarrollar un compilador de OpenACC. Uno de ellos se realiza en la Universidad de Houston y se llama *OpenUH* [18]. En España, la Universidad de la Laguna está trabajando en su propio compilador, *accULL*. En este caso, el compilador se basa en dos componentes, el propio compilador y una biblioteca que se encarga del manejo de la memoria y de la ejecución de los *kernels* [11] (los fragmentos de código que se ejecutan en la aceleradora).

3.3. Situación Actual

Se ha investigado acerca del estado de OpenACC y las diferentes implementaciones en diferentes aspectos como completitud en el soporte del estándar, robustez, rendimiento relativo del código generado y sensibilidad en cuanto a la geometría de bloque, concepto que será explicado unos párrafos a continuación.

La completitud en el soporte del estándar es muy subjetiva y poco fiable, puesto

que se utiliza la documentación aportada por cada compilador para su evaluación. No obstante, a pesar de no poder dar resultados objetivos al respecto, la sensación general es que ningún compilador soporta completamente el estándar y aún queda mucho trabajo por delante.

En relación a la robustez de la implementación, la idea ha consistido en probar el funcionamiento y el rendimiento de los diferentes *pragmas* soportados. Para ello se utilizó una herramienta del *Edinburgh Parallel Computing Centre* que se describe en profundidad en las siguientes páginas.

El rendimiento relativo se ha obtenido midiendo los tiempos de ejecución de benchmarks de prueba. Lo ideal sería la utilización de problemas reales del día a día, pero es una meta muy difícil.

La geometría del bloque se refiere a un concepto del funcionamiento de las *GPUs*. Una *GPU* posee miles de cores que se encargan de realizar los cálculos. Estos cores se agrupan en varios multiprocesadores (15 en algunos modelos) denominados Streaming Multiprocessors o SM. La ejecución de los programas se realiza en bloques consecutivos. No es extraño que se lancen miles de bloques en un *kernel*. Estos bloques tienen hilos de ejecución que se organizan de acuerdo a una geometría dada y que puede ser de una, dos o tres dimensiones. Esta geometría de bloque es fundamental por varios aspectos. En primer lugar, hay que maximizar la ocupación. Esto significa que al ejecutarse los bloques en los SMs ningún SM debe quedar infrautilizado. En segundo lugar, los accesos a memoria deben ser coalescentes, es decir, la posición relativa de cada hilo y la posición de memoria a la que acceden debe estar relacionada. Para terminar, debe maximizarse el uso de las caches. Debido a todo esto, la elección de una geometría u otra puede tener consecuencias enormes en los rendimientos y, por ello, se considera que es un concepto fundamental a tener en cuenta.

En OpenACC, la geometría del bloque se expresa en términos de *gang*, *worker* y *vector*. Esto se debe a que no solo se pretende usar *GPUs* sino también *Xeon Phis*, dispositivos en los que las instrucciones vectoriales son sumamente importantes. Desgraciadamente, según lo que se afirma en [27], la especificación es ambigua y no se dice mucho más aparte de que *gang* se relaciona con el paralelismo de grano grueso y *worker* y *vector* con paralelismo de grano fino. Cada implementación decide exactamente como funciona. En general, podemos asumir que el denominado *gang number* equivale al numero de bloques que se lanzan, el *worker number* equivale a la dimensión Y del bloque y el *vector length* equivale a la dimensión X del bloque.

3.3.1. Compiladores

En esta sección vamos a comentar algunos detalles de los compiladores que se encuentran disponibles de forma gratuita o mediante licencia académica, decisión pragmática pero necesaria. Se comentarán las fortalezas y debilidades que se han detectado, así como su estado de desarrollo y disponibilidad.

PGI Compiler

El compilador de PGI [23] está siendo desarrollado por *The Portland Group*, propiedad de Nvidia. Este compilador es ampliamente conocido y está siendo utilizado

en numerosos *webinarios*, talleres y conferencias. En el momento de escribir este documento, el compilador de PGI está disponible para su descarga gratuita como parte del denominado *OpenACC Toolkit* de Nvidia. Con la descarga se incluye una licencia de 90 días, limitado su uso a un único ordenador, pero con la posibilidad de obtener una licencia de investigador de un año.

El proceso de instalación en un sistema GNU/Linux es sencillo debido al instalador que se incluye en la descarga. Este compilador usa su propia versión de los drivers de *CUDA*. Se necesitan escribir a mano algunas rutas en las variables de entorno, pero es un proceso que viene correctamente descrito en la documentación, que es extensa y muy completa.

El compilador de PGI es una versión muy pulida en cuanto a instalación y facilidad de uso. Obtiene el primer puesto en todas las categorías utilizadas para la evaluación, aunque a pesar de ello, sigue estando en proceso de adquirir la madurez necesaria. Su principal fortaleza consiste en un muy reducido *overhead* lo que le hace ganar posiciones en cuanto a rendimiento con tamaños de problema pequeños o muy pequeños. Una de sus debilidades es que necesita mucho tiempo para iniciar el dispositivo (entorno a 3 segundos) y esto no ocurre hasta la ejecución del primer *kernel*.

Las pruebas que se han realizado parecen indicar que el compilador de PGI no permite modificar los parámetros de geometría de bloque, pero por otra parte su comportamiento es el mas cercano al que cabría esperar según los conocimientos del Grupo Trasgo.

accULL

El compilador accULL [25], desarrollado por la Universidad de La Laguna es una iniciativa de código abierto. Consiste en una estructura de dos capas que contiene YaCF [**?**] (*Yet another Compiler Framework*) y Frangollo [**?**], una biblioteca. YaCF es un traductor fuente a fuente mientras que Frangollo funciona como una interfaz que provee las operaciones más comunes en una aceleradora.

El proceso de instalación es simple y consiste en el lanzamiento de varios scripts, pero se necesitan algunas dependencias no incluidas. Estas dependencias son Ply 3.3 o superior (*Python Lex-Yacc*) [4] y Mako Templates [3]. Ambos están disponibles de forma gratuita. La documentación de accULL es corta, pero suficiente. Se incluyen scripts que exportan rutas a las variables de entorno.

En general, el compilador accULL es el más modesto de los tres, pero se defiende muy bien en muchos aspectos. Tiene un fuerte overhead, lo que dispara los tiempos en programas con pocos datos. Su mayor debilidad es su dificultad para compilar algunas características de C como los punteros a función.

La modificación de los valores de número de *gangs*, número de *workers* o longitud de *vector* no parecen tener ningún tipo de consecuencia en el rendimiento, lo que lleva a pensar que no está implementado.

OpenUH

El compilador OpenUH [18] ha sido desarrollado por la Universidad de Houston y se trata de otra iniciativa de código abierto. Utiliza Open64, un antiguo compilador-

optimizador de código abierto que ya no se desarrolla.

Existen binarios precompilados tanto para arquitecturas de 32 como de 64 bits.

Ocupa la segunda posición de los tres compiladores que se han evaluado, por encima de accULL. En rendimiento está relativamente cerca de accULL y tiene un overhead más pequeño. Al igual que accULL, recorta distancias con PGI con tamaño de datos grande.

En cuanto a la geometría de bloque, OpenUH es el único que permite que los valores introducidos tengan repercusión en el rendimiento. Jugar con estos valores permite que OpenUH supere a PGI en rendimiento en algunos casos y por un amplio margen, lo que muestra la importancia de la geometría del bloque.

3.3.2. Benchmarks

En esta sección se detallarán los benchmarks para uso con OpenACC existentes en el momento de escribir este documento, incluyendo su descripción, sus carencias, su disponibilidad, etc.

OpenACC Validation Testsuite

La *OpenACC Validation Testsuite* [28] desarrollada por la Universidad de Houston es una herramienta de pruebas muy interesante que se hizo con el objetivo de ayudar en el desarrollo de implementaciones del estándar OpenACC. Ejecuta las pruebas a modo de árbol. En el primer nivel están cada una de las directivas de OpenACC. En el segundo nivel, se encuentran cada cláusula admitida por la directiva situada en el nivel anterior, y así sucesivamente. De esta manera, se prueban todas las combinaciones posibles de directivas y cláusulas de forma totalmente sistemática.

Esta herramienta solo dice si una directiva está implementada o no, y si dicha implementación es o no correcta. No ofrece datos de rendimiento. No obstante, la *OpenACC Validation Testsuite* sólo está disponible para miembros del estándar OpenACC.

EPCC OpenACC Benchmark Suite

Desarrollado por el *Edinburgh Parallel Computing Centre*, el *EPCC OpenACC Benchmark Suite* [10] está dividido en tres categorías principales: Nivel 0, Nivel 1 y Aplicaciones. Por defecto realiza 10 mediciones dando como resultado la media aritmética. Mide el tiempo en microsegundos usando la función de OpenMP `omp_get_wtime()`.

En el denominado Nivel 0 se encuentran los siguientes benchmarks:

ContigH2D & ContigD2H son benchmarks que miden el tiempo necesario para copiar una cantidad de datos contiguos hacia o desde la aceleradora.

SlicedD2H & SlicedH2D son benchmarks que miden el tiempo necesario para copiar una cantidad de datos no contiguos hacia o desde la aceleradora.

Kernels_If este benchmark mide el overhead de la directiva `#pragma acc kernels if(0)`. Esta directiva ejecuta el código comentado en el host en lugar de en la aceleradora.

Para medir el overhead, ejecuta un fragmento sin *pragmas* y después ejecuta el mismo fragmento comentado como se ha dicho. La diferencia entre los dos tiempos de ejecución es el overhead de la directiva.

Parallel_If este benchmark mide el overhead de la directiva #pragma acc parallel if(0). Al igual que en el caso anterior, este *pragma* ejecuta el bloque de código en el host.

Parallel_private mide la diferencia entre ejecutar una directiva con una cláusula de datos private, o ejecutarla sin dicha cláusula. El resultado es el overhead de la cláusula private en una directiva.

Parallel_firstprivate mide la diferencia entre ejecutar una directiva con una cláusula de datos firstprivate, o ejecutarla sin dicha cláusula. El resultado es el overhead.

Kernels_combined mide la diferencia de tiempo entre la ejecución de un bloque de código con dos directivas, #pragma acc kernels y #pragma acc loop, y el mismo bloque de código con una única directiva #pragma acc kernels loop.

Parallel_combined mide la diferencia de tiempo entre la ejecución de un bloque de código con dos directivas, #pragma acc parallel y #pragma acc loop, y el mismo bloque de código con una única directiva #pragma acc parallel loop.

Update_host mide el tiempo requerido por el pragma #pragma acc update host(a[0:n]) para actualizar el parámetro indicado en el host.

Kernels_invocation mide el overhead de la invocación de un kernel con la directiva #pragma acc kernels. Se consigue mediante la medición de tres bloques de código con dicho pragma. Los dos primeros ejecutan una única instrucción, mientras que el tercero ejecuta dos. La diferencia del tiempo requerido por el tercer bloque con respecto a los otros dos es el tiempo necesario para la invocación.

Parallel_invocation repite el procedimiento del benchmark anterior para medir el overhead de la invocación de un bloque con la directiva #pragma acc parallel.

Parallel_reduction mide el overhead de una reducción en un pragma #pragma acc parallel. Esto lo consigue midiendo dos bloques de código, uno con la reducción y el otro sin ella.

Kernels_reduction mide el overhead de una reducción en un pragma #pragma acc kernels. Repite el proceso del benchmark anterior.

El Nivel 1 se compone de una serie de kernels tipo BLAS (Subprogramas Básicos de Álgebra Lineal). Se basan en los benchmarks de Polybench y Polybench/GPU [24] y miden el tiempo de ejecución como medida de rendimiento. Los benchmarks se ejecutan primero en la CPU y luego se comparan con el resultado obtenido en la GPU.

Finalmente el Nivel de Aplicaciones contiene tres aplicaciones:

27stencil es un programa que utiliza un stencil de tres dimensiones y 27 puntos. Cada punto se actualiza con el valor de los puntos adyacentes en la estructura tridimensional.

LeCore consiste en una simulación de elasticidad lineal.

Himeno resuelve una ecuación de Poisson de tres dimensiones utilizando el método de Jacobi en un stencil de 9 puntos.

La utilización de esta suite de benchmarks implica la compilación manual de la misma con cada uno de los compiladores a evaluar. Al ejecutar la suite los benchmarks se ejecutan uno detrás de otro y se va imprimiendo en pantalla un resumen de los tiempos obtenidos en cada uno.

En la investigación previa realizada se ha utilizado esta suite y los resultados de la compilación con los tres compiladores que se han estudiado puede verse en la tabla 3.1.

La herramienta permite modificar los tamaños de entrada a utilizar en los benchmarks. En la investigación llevada a cabo se utilizaron tamaños de 1kB, 1MB y 10MB. En el caso de los benchmarks de movimiento de datos se usó también un tamaño de 1GB. Los resultados del Nivel 0 han sido recopilados en la tabla 3.2. En cuanto a los datos de movimiento de datos se pueden ver en las tablas 3.3, 3.4, 3.5 y 3.6. Los resultados del Nivel 1 y Aplicaciones se han recopilado en las tablas 3.7, 3.8 y 3.9, en las que además se ha normalizado con respecto a los resultados obtenidos por el código generado con el compilador de PGI y se ha calculado la media geométrica de los ratios.

Rodinia para OpenACC

Rodinia para OpenACC [22] es un intento por parte de la empresa *PathScale Inc.* para traducir los benchmarks originales de la suite de Rodinia [6,7] para computación heterogénea. La suite de Rodinia se compone de benchmarks para CPUs *multicore* y GPUs.

A fecha de realización de los experimentos, la última versión disponible es la del 25 de Abril, disponible en *GitHub*. Esta versión requiere la compilación manual de cada uno de los benchmarks, así como su ejecución. Incluye ficheros de datos para su uso con algunos de los benchmarks. Los resultados de compilación obtenidos en la experimentación pueden verse en la Tabla 3.10.

Las características de los benchmarks que funcionan con al menos dos de los compiladores disponibles son las siguientes:

Gaussian Elimination Consiste en la computación de todas las variables de un sistema lineal fila a fila, usando el método de eliminación gaussiana.

Needleman-Wunsch Se trata de un programa de alineamiento de cadenas de ADN que organiza pares potenciales en una matriz bidimensional, utilizando un algoritmo de programación dinámica. El resultado final se obtiene aplicando *backtracking*.

LU Decomposition Es un algoritmo para la resolución de sistemas de ecuaciones mediante la descomposición de la matriz en un producto de matrices triangulares.

Tabla 3.1: Resultados de Compilación de la EPCC Benchmark Suite

EPCC	PGI	OpenUH	accULL
Nivel 0			
ContigH2D	OK	OK	OK
ContigD2H	OK	OK	OK
SlicedD2H	OK	OK	OK
SlicedH2D	OK	OK	OK
Kernels_If	OK	FALLO	OK
Parallel_If	OK	OK	OK
Parallel_private	OK	FALLO	OK
Parallel_1stpriv.	OK	FALLO	OK
Kernels_comb.	OK	OK	OK
Parallel_comb.	OK	OK	OK
Update_Host	OK	OK	OK
Kernels_Invoc.	OK	OK	OK
Parallel_Invoc.	OK	OK	OK
Parallel_Reduct.	OK	OK	OK
Kernels_Reduct.	OK	OK	OK
Nivel 1			
2MM	OK	OK	OK
3MM	OK	OK	OK
ATAX	OK	OK	OK
BICG	OK	OK	OK
MVT	OK	OK	OK
SYRK	OK	OK	OK
COV	OK	OK	OK
COR	OK	OK	OK
SYR2K	OK	OK	OK
GESUMMV	OK	OK	OK
GEMM	OK	OK	OK
2DCONV	OK	OK	OK
3DCONV	OK	OK	OK
Aplicaciones			
27S	OK	OK	OK
LE2D	OK	FALLO	OK
HIMENO	OK	FALLO	OK

Tabla 3.2: Resultados de la ejecución del Nivel 0 con EPCC Benchmark Suite

EPCC Nivel 0	**PGI**	**OpenUH**	**accULL**
Kernels_if	-37.50	Fallo	4.54
Parallel_if	-30.76	-0.48	1237.02
Parallel_private	-21.94	Fallo	51.09
Parallel_1stpriv	Fallo	Fallo	-213.83
Kernels_comb.	-1.67	-108.43	-127.17
Parallel_comb.	-0.05	-2.74	33.38
Update_host	478.63	373.22	548.77
Kernels_Invoc.	Fallo	12.76	2398.20
Parallel_Invoc.	31.81	13.47	1377.88
Parallel_reduct.	-14.85	-164.41	-2168.12
Kernels_reduct.	-8.49	-172.31	-2009.11

Tabla 3.3: Resultados de movimiento de datos EPCC Benchmark Suite: 1kB

Mov. Datos	PGI		OpenUH		accULL	
10reps, 1kB	μsec	norm.	μsec	norm.	μsec	norm.
ContigH2D	30.827	1.0	322.699	10.47	338.218	10.97
ContigD2H	14.686	1.0	323.319	22.01	343.919	23.42
SlicedH2D	12.087	1.0	310.897	25.72	315.914	26.13
SlicedD2H	14.948	1.0	324.010	21.67	327.714	21.92
			M.Geométrica	18.93	M.Geométrica	19.58

Tabla 3.4: Resultados de movimiento de datos EPCC Benchmark Suite: 1MB

Mov. Datos	PGI		OpenUH		accULL	
10reps, 1MB	μsec	norm.	μsec	norm.	μsec	norm.
ContigH2D	484.347	1.0	950.789	1.96	727.839	1.50
ContigD2H	461.936	1.0	632.691	1.37	792.761	1.72
SlicedH2D	17.094	1.0	267.982	15.68	274.462	16.06
SlicedD2H	36.335	1.0	254.702	7.01	285.685	7.86
			M.Geométrica	4.14	M.Geométrica	4.24

Tabla 3.5: Resultados de movimiento de datos EPCC Benchmark Suite: 10MB

Mov. Datos	PGI		OpenUH		accULL	
10reps, 10MB	μsec	norm.	μsec	norm.	μsec	norm.
ContigH2D	4141.402	1.0	6887.984	1.66	3354.666	0.81
ContigD2H	5876.088	1.0	2043.747	0.35	4396.052	0.74
SlicedH2D	27.322	1.0	404.214	14.79	427.203	15.64
SlicedD2H	48.017	1.0	269.818	5.62	280.203	5.84
			M.Geométrica	2.64	M.Geométrica	2.72

Tabla 3.6: Resultados de movimiento de datos EPCC Benchmark Suite: 1GB

Mov. Datos	PGI		OpenUH		accULL	
10reps, 1GB	μsec	norm.	μsec	norm.	μsec	norm.
ContigH2D	32310.009	1.0	788945.913	24.42	296340.991	9.17
ContigD2H	55179.119	1.0	553282.976	10.03	347280.359	6.29
SlicedH2D	400.066	1.0	535.011	1.34	533.943	1.34
SlicedD2H	158.071	1.0	2818.100	17.83	4294.407	27.17
			M.Geométrica	8.75	M.Geométrica	6.76

Tabla 3.7: Resultados de EPCC Benchmark Suite: 1kB

Tiempo ejec.	PGI		OpenUH		accULL	
10reps, 1kB	μsec	norm.	μsec	norm.	μsec	norm.
2MM	99.087	1.0	522.304	5.27	2799.229	28.25
3MM	80.204	1.0	380.683	4.75	3799.048	47.37
ATAX	58.103	1.0	327.110	5.63	2564.702	44.14
BICG	72.408	1.0	350.380	4.84	2628.499	36.30
MVT	80.037	1.0	354.743	4.43	2665.299	33.30
SYRK	68.426	1.0	289.512	4.23	2394.803	35.00
COV	87.261	1.0	314.617	3.61	3795.372	43.49
COR	104.976	1.0	337.362	3.21	5208.668	49.62
SYR2K	73.290	1.0	317.574	4.33	2469.765	33.70
GESUMMV	65.613	1.0	312.996	4.77	1500.021	22.86
GEMM	49.710	1.0	323.725	6.51	1237.473	24.89
2DCONV	46.444	1.0	286.174	6.16	1207.528	26.00
3DCONV	45.514	1.0	285.792	6.28	1202.494	26.42
27S	335.884	1.0	432.801	1.29	3273.728	9.75
LE2D	6842374	1.0	FALLO	FALLO	FALLO	FALLO
HIMENO	547939	1.0	FALLO	FALLO	FALLO	FALLO
			M.Geom.	4.39	M.Geom.	24.38

Tabla 3.8: Resultados de EPCC Benchmark Suite: 1MB

Tiempo ejec.	PGI		OpenUH		accULL	
10reps, 1MB	μsec	norm.	μsec	norm.	μsec	norm.
2MM	2305.698	1.0	3467.703	1.50	4002.412	1.74
3MM	705.409	1.0	1453.137	2.06	5265.778	7.46
ATAX	484.204	1.0	1222.420	2.52	4212.914	8.70
BICG	502.849	1.0	1256.871	2.50	4229.466	8.41
MVT	538.135	1.0	FALLO	FALLO	4355.322	8.09
SYRK	1374.769	1.0	2543.616	1.85	4000.674	2.91
COV	3681.660	1.0	4251.957	1.15	23969.443	6.51
COR	3863.096	1.0	4318.953	1.12	25732.814	6.66
SYR2K	1968.789	1.0	2532.029	1.29	4586.741	2.37
GESUMMV	406.623	1.0	1195.669	2.94	2709.591	6.66
GEMM	1041.651	1.0	23642.850	22.70	3595.218	3.45
2DCONV	1637.363	1.0	1912.236	1.17	2991.542	1.83
3DCONV	9388.137	1.0	9670.520	1.03	10058.497	1.07
27S	2179.599	1.0	2224.064	1.02	8342.865	3.83
LE2D	6861089	1.0	FALLO	FALLO	FALLO	FALLO
HIMENO	540513	1.0	FALLO	FALLO	FALLO	FALLO
			M.Geom.	1.92	M.Geom.	4.11

Tabla 3.9: Resultados de EPCC Benchmark Suite: 10MB

Tiempo ejec.	PGI		OpenUH		accULL	
10reps, 10MB	msec	norm.	msec	norm.	msec	norm.
2MM	64.407	1.0	64.336	0.99	20.476	0.32
3MM	11.610	1.0	21.113	1.82	30.009	2.58
ATAX	4.345	1.0	7.415	1.71	7.659	1.76
BICG	4.385	1.0	7.397	1.69	7.689	1.75
MVT	4.406	1.0	FALLO	FALLO	7.945	1.80
SYRK	26.537	1.0	57.242	2.16	42.834	1.61
COV	117.757	1.0	134.047	1.14	230.241	1.96
COR	120.612	1.0	122.814	1.02	223.836	1.86
SYR2K	23.450	1.0	27.939	1.16	30.062	1.28
GESUMMV	3.567	1.0	7.191	2.02	6.297	1.77
GEMM	17.788	1.0	239.713	13.48	48.567	2.73
2DCONV	7.848	1.0	7.725	0.98	8.687	1.11
3DCONV	40.551	1.0	40.235	0.99	43.729	1.08
27S	9.243	1.0	9.254	1.00	42.389	4.59
LE2D	6863	1.0	FALLO	FALLO	FALLO	FALLO
HIMENO	528	1.0	FALLO	FALLO	FALLO	FALLO
			GeoMean	1.59	GeoMean	1.63

Tabla 3.10: Compilación de benchmarks de Rodinia

RODINIA	PGI	OpenUH	accULL
backprop	OK	FALLO	FALLO
bfs	OK	OK	FALLO
b+tree	OK	FALLO	FALLO
cfd	OK	OK	FALLO
gaussian	OK	OK	OK
heartwall	FALLO	FALLO	FALLO
hotspot	OK	OK	FALLO
kmeans	FALLO	FALLO	FALLO
lavaMD	OK	FALLO	FALLO
leukocyte	FALLO	FALLO	FALLO
lud	OK	OK	FALLO
myocyte	FALLO	FALLO	FALLO
nn	OK	OK	FALLO
nw	OK	OK	OK
particlefilter	OK	FALLO	FALLO
pathfinder	OK	OK	FALLO
srad1	OK	FALLO	FALLO
srad2	OK	OK	FALLO
streamcluster	FALLO	FALLO	FALLO

CFD Solver Consiste en un algoritmo para la solución de una ecuación de Euler tridimensional.

HotSpot Es un *stencil* de dos dimensiones para la estimación de la temperatura de un procesador usando ecuaciones diferenciales.

Pathfinder Utiliza programación dinámica para la obtención del camino más corto desde un extremo a otro de una matriz de pesos.

SRAD2 Se trata de un programa para la reducción de ruido en una imagen, utilizando ecuaciones diferenciales parciales.

Los resultados obtenidos con la suite de Rodinia hecha por *Pathscale Inc.* pueden verse en las Tablas 3.11 y 3.12.

Tabla 3.11: Tiempos de ejecución de Rodinia, incluyendo transferencias de memoria. Tiempo total en milisegundos.

Tiempo Ejecución 3 reps	PGI	OpenUH	accULL
gaussian	2440.206	52.491	15422.944
nw	2640.497	652.180	322.101
lud	3803.756	1723.576	FALLO
cfd	2677.387	0.846	FALLO
hotspot	2386.325	53.219	FALLO
pathfinder	5137.865	34.738	FALLO
srad2	2488.895	692.063	FALLO

Tabla 3.12: Tiempos de ejecución de Rodinia, sin incluir transferencias de memoria. Tiempo total en milisegundos.

Tiempo Ejecución 3 reps	PGI	OpenUH	accULL
Gaussian	57.345	36.092	15415.992

3.4. Problemas Detectados con las Herramientas Actuales

En la sección anterior se ha presentado la situación actual en el ámbito del estándar OpenACC en relación a los compiladores existentes y las herramientas de benchmarking disponibles a fecha de realización del presente Trabajo de Fin de Grado. Así mismo, se ha explicado brevemente la investigación realizada con dichas herramientas relativa al análisis comparativo del rendimiento del código generado con los distintos compiladores. En esta sección se comentarán las conclusiones obtenidas de dicha investigación que han llevado finalmente al diseño e implementación de *TORMENT OpenACC2016*.

3.4.1. Robustez y completitud de los compiladores de OpenACC

Del estudio previo realizado, se concluye que los compiladores de OpenACC varían en cuanto a robustez y completitud, pero en ningún caso se alcanza un grado de madurez suficientemente alto.

El compilador de PGI es el más robusto y completo. A fecha de redacción de este Trabajo de Fin de Grado, y utilizando la versión 15.7 de PGI, algunas de las características menos utilizadas no están aún implementadas. Durante el uso de este compilador con las herramientas *EPCC Benchmark Suite* y *Rodinia* se han encontrado algunos problemas de compilación menores.

El compilador de la Universidad de Houston, OpenUH, mantiene una segunda posición en cuanto a robustez y completitud, con la mayoría de las funcionalidades habituales disponibles. No obstante, varias implementaciones no se ajustan estrictamente al estándar OpenACC.

El compilador accULL, de la Universidad de la Laguna, ocupa la tercera posición debido a la falta de algunas funcionalidades muy útiles, como las reducciones

para máximos y mínimos, que no parecen funcionar correctamente, o el soporte de características de C como los punteros a funciones.

En general, estos compiladores pueden usarse para la paralelización de un considerable número de fragmentos de código utilizando OpenACC, pero aún necesitan un considerable desarrollo para alcanzar un nivel de madurez suficientemente alto.

3.4.2. Problemas en el análisis de rendimiento del código generado

Una parte muy importante del tiempo dedicado al estudio previo del estado actual del estándar OpenACC y sus implementaciones se ha dedicado al análisis de rendimiento del código generado por los compiladores. Para ello se han utilizado las herramientas existentes: *EPCC Benchmark Suite* y *Rodinia*.

Overhead en la ejecución de código en la GPU

Este aspecto del rendimiento ha sido posible gracias a los benchmarks del Nivel 0 del *EPCC Benchmark Suite*. Se ha podido evaluar el tiempo que se gasta en la invocación de kernels generados del procesamiento de los #pragmas correspondientes. No obstante, esta información no es útil ya que solo observa una pequeña parte de la ejecución de código en la GPU. No tiene en cuenta transferencias de memoria ni los cálculos y computación necesarios para la resolución de un problema. Esta información sobre el *overhead* de los diferentes #pragmas es útil para el desarrollo del compilador como tal, pero no para el público en general.

Estos motivos llevan a decidir que *TORMENT OpenACC2016* no debe centrarse en el uso de microkernels o en el análisis del rendimiento de operaciones de tan bajo nivel, como puede ser el lanzamiento de un kernel CUDA generado del procesamiento de un #pragma.

Transferencias de memoria

Parte del Nivel 0 del *EPCC Benchmark Suite* consiste en una serie de benchmarks para la medición del tiempo de transferencia de memoria, tanto contigua como dispersa, desde el Host a la GPU y viceversa. Este resultado tiene mucha relevancia, ya que, habitualmente, en los cálculos en una GPU el cuello de botella suele ser la memoria y es fundamental conseguir que las transferencias sean lo más eficientes posible.

Al igual que en el apartado anterior, los resultados de tiempo de transferencia de memoria son muy relevantes, pero por si solos no dicen demasiado y tienen el problema de no poder ser comparados con ningún resultado secuencial de referencia. Esto lleva a la idea de que *TORMENT OpenACC2016*, si bien debe implementar benchmarks que hagan uso de la memoria de la GPU y transferencias entre la misma y la memoria del Host, no debe medir estos resultados de forma independiente sino como parte de un problema concreto.

Rendimiento de ejecución

El grueso de los benchmarks tanto del *EPCC Benchmark Suite* como de *Rodinia* consisten en la ejecución de diferentes problemas de varios dominios como la ingeniería, las matemáticas o la biología. Muchos de los benchmarks que implementan han sido ampliamente utilizados a los largo de los años y son muy conocidos en el ámbito del análisis de rendimiento.

En ambos casos, el principal problema que se presenta es que no están preparados para el estado de desarrollo de un conjunto de compiladores de OpenACC y, como consecuencia, en ningún caso es posible lanzar las suites con todos los compiladores con resultados correctos. Durante la investigación llevada a cabo fue necesario en varias ocasiones modificar el código fuente de varios #pragmas para poder compilar correctamente, volviendo a recompilar usando los tres compiladores después de cada cambio para evitar usar código distinto. Esto muchas veces provocaba que un código dejase de compilar con otro compilador. En algunos casos, ni siquiera intentando cambiar el código fue posible llegar a compilar algunos benchmarks.

Estos problemas son el motivo por el cual *TORMENT OpenACC2016* contendrá un conjunto de benchmarks preparados específicamente para funcionar con los compiladores que ha sido posible conseguir, de modo que no se produzcan errores de compilación y se pueda ejecutar correctamente la suite independientemente del compilador a usar.

Por otra parte, de los benchmarks que se incluyen en *EPCC Benchmark Suite* y *Rodinia* se observa que muchos de ellos si bien resuelven problemas distintos, en el fondo utilizan siempre la misma estructura en sus algoritmos, por lo que muchos benchmarks no son diferentes unos de otros. Esto ha provocado la decisión de que los benchmarks que se implementen en *TORMENT OpenACC2016* se elijan en base a una serie de características diferentes unos de otros de modo que los resultados sean lo más relevantes posible.

En este capítulo se ha analizado el Estado del Arte, describiendo las herramientas existentes y los diferentes compiladores que implementan el estándar, así como enumerando los problemas encontrados durante el desarrollo del análisis de rendimiento. A continuación se expondrá el análisis de requisitos para la herramienta *TORMENT OpenACC2016*.

Capítulo 4

Análisis de Requisitos

Este capítulo recoge la parte de Análisis de Requisitos de *TORMENT OpenACC2016*. Dado que este Trabajo Fin de Grado consiste en el diseño e implementación de una suite de benchmarks, las herramientas habituales usadas en Ingeniería de Software no son, en algunos casos, suficientemente expresivas. Por este motivo se ha omitido la realización de diagramas de casos de uso o modelos del dominio, ya que en este caso no son adecuados para documentar el proyecto.

Por tanto, a continuación se aporta el análisis de requisitos, según los datos y experiencias recopiladas durante el estudio previo realizado, cuyas conclusiones figuran en el capítulo sobre el Estado del Arte.

4.1. Objetivos

Los objetivos son requisitos funcionales de usuario que se detallan a continuación:

4.1.1. OBJ-01: Obtención de datos de rendimiento de código OpenACC

TORMENT OpenACC2016 deberá permitir la obtención de datos acerca del rendimiento de código generado por compiladores de OpenACC, ofreciendo una puntuación que permita la comparación de diferentes compiladores y máquinas.

4.1.2. OBJ-02: Automatización del proceso de benchmarking

TORMENT OpenACC2016 debe estar automatizado, permitiendo al usuario la ejecución de la suite de benchmarks para todos los compiladores disponibles con una sola llamada al programa.

4.2. Requisitos Funcionales

A continuación se detallan los requisitos funcionales del sistema, que describen los servicios que se proporcionarán.

4.2.1. RF-01: Configuración y detección de compiladores de OpenACC

El sistema deberá permitir al usuario la detección y obtención de rutas a los compiladores de OpenACC presentes en su máquina, permitiendo al usuario la elección de la ruta apropiada en caso de existir varias opciones. Esta detección se efectuara según lo dispuesto en el RNF-01.

4.2.2. RF-02: Compilación de la suite

El sistema deberá permitir que la suite sea compilada por cada uno de los compiladores de forma transparente para el usuario durante la ejecución de la suite. Esta compilación se realizará de acuerdo a los requisitos no funcionales RNF-02, RNF-03, RNF-04 y RNF-05.

4.2.3. RF-03: Ejecución de la suite

El sistema deberá permitir la ejecución de la suite, proceso que incluye la compilación y ejecución de los benchmarks, para la obtención de resultados por cada uno de los compiladores disponibles. Esta ejecución cumple lo dispuesto en los RNF-04, RNF-06, RNF-07 y RNF-08.

4.2.4. RF-04: Obtención de ratios con respecto a tiempos de referencia

El sistema deberá permitir la obtención de ratios con respecto a tiempos de referencia, según lo dispuesto en los RNF-09.

4.2.5. RF-05: Generación de resúmenes de resultados

El sistema deberá permitir la generación de resúmenes de resultados. Estos resúmenes tendrán información sobre: identificación del sistema (RI-04), información sobre la pila de software (RI-05), información del sistema host (RI-06), información de GPU (RI-07). Se cumplirá con lo dispuesto en el RNF-10.

4.3. Requisitos de Información

A continuación se detallan los requisitos de información de *TORMENT OpenACC2016*.

4.3.1. RI-01: Resultados *runtime*

- Tiempo medio
- Tiempo mínimo
- Desviación estándar

4.3.2. RI-02: Resultados en fichero de texto

Para cada benchmark se incluirá:

- Nombre del benchmark
- Tiempo mínimo
- Ratio del tiempo mínimo
- Tiempo medio
- Ratio del tiempo medio
- Desviación estándar

4.3.3. RI-03: Tiempos de referencia

- Tiempo mínimo
- Tiempo medio

4.3.4. RI-04: Identificación del sistema

- Versión del benchmark
- Hostname
- Username

4.3.5. RI-05: Información sobre la pila de software

- Kernel Linux
- Compilador GCC
- Compilador NVCC
- Compiladores de OpenACC usados

4.3.6. RI-06: Información del sistema host

- Modelo
- Arquitectura
- Número de cores
- Frecuencia máxima
- Frecuencia mínima
- Cache
- RAM

4.3.7. RI-07: Información de GPU

Por cada GPU:

- Modelo

4.4. Requisitos No Funcionales

Los requisitos no funcionales definen propiedades del sistema. Para *TORMENT OpenACC2016* son los siguientes:

4.4.1. RNF-01: Detección automática de compiladores

La detección de los compiladores durante la configuración de la suite se hará de forma automática, utilizando los comandos existentes en un entorno UNIX como, por ejemplo, find.

4.4.2. RNF-02: Uso de Makefiles para la compilación

La compilación de la suite se efectuará invocando al comando make, por lo que se dispondrán de los ficheros Makefile necesarios para la correcta compilación del código fuente.

4.4.3. RNF-03: Datos específicos de compilador

Los datos específicos del compilador que sean necesarios para la correcta compilación y ejecución del código fuente serán añadidos automáticamente a un fichero de cabecera de C, para su inclusión donde fuera necesario.

4.4.4. RNF-04: Temporalidad de los binarios de *TORMENT OpenACC2016*

Los binarios producto de la compilación serán eliminados automáticamente tras la ejecución de la suite por cada compilador.

4.4.5. RNF-05: Mensajes generados durante la compilación

Los mensajes y avisos generados durante la compilación serán ocultados de la salida estándar y redireccionados a ficheros temporales dentro de una carpeta también temporal.

4.4.6. RNF-06: Ejecución gestionada mediante Script

La ejecución de la suite será gestionada y lanzada mediante un *script* bash, ocultando toda la complejidad al usuario.

4.4.7. RNF-07: Resultados en *runtime*

La ejecución de la suite mostrará en tiempo de ejecución los resultados que se vayan obteniendo de la ejecución de los benchmarks. Estos resultados están definidos en el RI-01.

4.4.8. RNF-08: Resultados en fichero de texto

Tras la ejecución de *TORMENT OpenACC2016*, se creará un fichero de texto con los resultados definidos en el RI-02. En cada linea del fichero se guardarán los resultados de cada uno de los compiladores.

4.4.9. RNF-09: Tiempos de referencia

Los tiempos de referencia estarán incluidos junto al código fuente y serán obtenidos en una máquina de referencia, ejecutados en modo secuencial y compilados con GCC. Estos tiempos de referencia se definen en el RI-03.

4.4.10. RNF-10: Obtención de información del sistema

La información del sistema recogida en los RI-04, RI-05, RI-06 y RI-07 se recopilará de forma automática mediante el uso de *scripts* bash y comandos UNIX y ficheros localizables en /proc.

4.4.11. RNF-11: Formato del fichero de resumen de resultados

El fichero resumen de resultados generado será un fichero HTML con estilo CSS provisto previamente en una hoja de estilo disponible con *TORMENT OpenACC2016*.

4.4.12. RNF-12: Lenguaje de programación

Las partes de *TORMENT OpenACC2016* que se implementen utilizando un lenguaje de alto nivel en vez de un script deberán utilizar C.

4.5. Reglas de Negocio

Las reglas de negocio describen restricciones del dominio. A continuación se detallan las reglas de negocio para *TORMENT OpenACC2016*.

4.5.1. RN-01: Cancelación de la ejecución ante resultados erróneos

En caso de que se encuentre un resultado erróneo, la ejecución de la suite para el compilador que genera el código que da el resultado erróneo se cancelará y sus resultados serán ignorados.

4.5.2. RN-02: Repeticiones y ejecuciones ignoradas

Los benchmarks serán lanzados una primera vez (cuyo resultado será ignorado), seguido de un bucle de repeticiones para los que se calcularán los resultados.

Con el análisis de requisitos realizado, el siguiente paso consiste en establecer el diseño que dará respuesta a estos requisitos, describiendo los diferentes procesos y estableciendo los benchmarks a implementar en *TORMENT OpenACC2016*, así como la métrica a utilizar.

Capítulo 5

Diseño

En este capítulo se presenta el Diseño planteado para *TORMENT OpenACC2016*. Al igual que en el capítulo relativo al Análisis, el tipo de herramienta que se presenta en este Trabajo Fin de Grado no es adecuado para trabajar con los diagramas y modelos habituales. Los diagrama de secuencia y los diagramas de clase serán sustituidos por diagramas de actividad, que se corresponden con la vista de proceso del sistema, mostrando el comportamiento y las acciones relativas a las diferentes actividades, así como los flujos de datos y control.

Además, este capítulo definirá los benchmarks que serán desarrollados como objetivo para la versión 1.0 de *TORMENT OpenACC2016*. La finalidad de estos benchmarks es dar una visión global del comportamiento del código generado por los compiladores de OpenACC en una arquitectura concreta. Por este motivo, la elección de los benchmarks que serán implementados se basa en un conjunto de características que suponen de algún modo algún tipo de desafío a la capacidad de paralelización automática de los compiladores.

5.1. Diseño del *Wrapper* de Ejecución

La suite principal de *TORMENT OpenACC2016* es el núcleo de la herramienta y la parte encargada de dar respuesta a los requisitos definidos en el capítulo anterior. *TORMENT OpenACC2016* se dividirá en dos partes: una interfaz con el usuario que se realizará con *scripts* bash y el programa principal que se desarrollará en C. De igual forma se definirá la métrica *TORMENT_ACC2016* que permitirá la evaluación numérica del rendimiento del código que cada compilador genera.

5.1.1. Interfaz con el usuario: *Scripts*

Los *scripts* son la parte más cercana al usuario y están pensados para establecer la forma de interacción de *TORMENT OpenACC2016* con dicho usuario. La decisión de escribir *scripts* en lugar de incluir su funcionalidad en el programa C se basa en varios motivos:

En primer lugar, el programa C ha de ser compilado por diferentes compiladores. Dado que se desea simplificar el proceso de benchmarking al usuario, es necesario automatizar estas tareas en un *script*.

Por otra parte, el código fuente ha de ser modificado para incluir información del compilador usado para su compilación. Dado que muchos de los compiladores no siguen el funcionamiento estándar que se puede esperar de compiladores como GCC o similares, que permiten el paso de constantes mediante el argumento -D, o de diferentes rutas de *includes* con el argumento -I. Esta información debe por tanto suministrarse antes de la compilación en un fichero de cabecera. Este proceso puede ser automatizado en un *script*.

El usuario puede pretender realizar tres acciones relativamente independientes que serán implementadas, por tanto, en tres scripts separados. Estas acciones son:

- Configurar la suite
- Ejecutar la suite
- Generar un resumen de una ejecución previa de la suite

Configurar la suite

La compilación y ejecución de la suite de benchmarks, de acuerdo a lo visto durante la etapa de investigación del estado del arte, necesita de cierta información para su correcto funcionamiento.

En concreto, será necesario conocer la ubicación de los ejecutables correspondientes a los diferentes compiladores soportados, si es que están disponibles en la máquina del usuario. Además, el usuario puede tener varias versiones diferentes instaladas en la máquina, por lo que debe darse la opción de seleccionar la más adecuada. Habitualmente, estos compiladores no disponen su ejecutable en una ruta fácil de escribir (o de recordar). Esto supone que pedir al usuario la ruta al ejecutable supondrá, en la mayoría de los casos, una ruta incorrecta.

Se propone el uso del comando find que devuelve todas aquellas rutas en las que se encuentre un fichero con el nombre indicado. Con este comando puede obtenerse todas las rutas relativas a un ejecutable que se conoce de antemano. Mediante el script de configuración se podrá recorrer todos los resultados posibles, preguntando al usuario si es o no la ruta que desea.

Además de las rutas a los ejecutables de los compiladores disponibles, es posible que el usuario desee ejecutar el programa con un comando de ejecución diferente. En el caso de las pruebas realizadas durante la investigación previa realizada en el Grupo Trasgo, los programas se ejecutaban mediante un sistema de colas denominado SLURM. Por esto, se propone que el script de configuración pregunte al usuario si desea utilizar un comando de ejecución diferente.

Finalmente, el script de configuración deberá guardar todos los parámetros relevantes en un fichero de configuración que podrá ser accedido por el script de ejecución cada vez que se invoque.

En la Figura 5.1 se incluye el diagrama de actividad relativo a la configuración de *TORMENT OpenACC2016*.

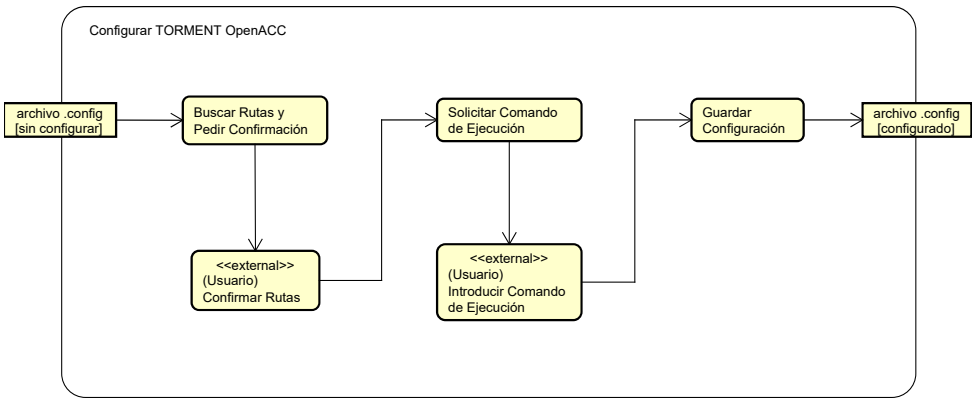


Figura 5.1: Diagrama de Actividad para la configuración de la herramienta

Ejecutar la suite

La parte central de la herramienta es la ejecución de los benchmarks para la obtención de resultados de rendimiento y la métrica *TORMENT_ACC2016* que se definirá más tarde. Para ello se requiere un fichero de configuración realizado por el script comentado en el apartado anterior. Si este fichero no existe deberá lanzarse automáticamente el script de configuración.

Con las rutas a los compiladores y la forma de ejecutar la suite definidas, el script de ejecución deberá proceder, de forma secuencial, a la compilación y ejecución de la suite con los compiladores de los que disponga el usuario.

En el caso de *TORMENT OpenACC2016*, la mejor manera de gestionar la compilación del código fuente es el uso del comando make y ficheros Makefile en cada directorio. Durante las pruebas y la investigación realizada, este método ha demostrado ser el menos problemático para su uso con compiladores en etapas tempranas de desarrollo.

Para compilar correctamente el programa, deberá modificarse un fichero de cabecera con la información del compilador que se está usando. Esto permitirá gestionar las diferencias, en cuanto a sintaxis de código, que tienen los compiladores mediante el uso de compilación condicional. Además, permitirá unificar el código secuencial, OpenACC y CUDA en un único fichero, evitando la duplicación innecesaria de código. Tras generar este fichero de cabecera e invocar al comando make en el directorio de código fuente, se ejecutará la suite de benchmarks.

Una vez finalizada la ejecución de la suite, se invocará de nuevo el comando make para proceder a la limpieza de binarios y código intermedio generado, de modo que el siguiente compilador pueda volver a compilar el código sin interferencia de estos ficheros.

Completada la compilación y ejecución de la suite con todos los compiladores, se procederá a la generación del resumen de resultados. Dado que puede ser útil para el usuario la posibilidad de regenerar resúmenes con datos obtenidos en una ejecución anterior esta funcionalidad se desarrollará en un tercer script.

La Figura 5.2 muestra el diagrama de actividad de la ejecución de la suite *TORMENT OpenACC2016*. La actividad de compilación se puede ver en la Figura 5.3.

Generar resumen

Es común, y útil para el usuario, que una herramienta de benchmarking genere un resumen de resultados con la información más relevante. En el Anexo I se puede ver un resumen de resultados correspondiente a SPEC CPU. El script de generación de resúmenes estará encargado de producir un fichero HTML con la información que se ha especificado en el análisis de requisitos.

Se producirá una plantilla HTML sobre la cual se presentarán los datos adecuadamente formateados. El script se ocupará de utilizar comandos como cat o awk para recuperar la información de ficheros de /proc o similares. Esta información sera procesada y formateada en el fichero HTML resultante.

Con los ficheros de resultados se generarán las tablas correspondientes y se calculará la métrica *TORMENT_ACC2016* para los compiladores de OpenACC.

5.1.2. El núcleo de la herramienta

Una vez definida la parte de *TORMENT OpenACC2016* encargada de interactuar con el usuario, se presentará la parte encargada de ejecutar y recopilar la información del tiempo requerido por los diferentes benchmarks.

Para modularizar lo máximo posible la herramienta, los benchmarks estarán definidos en su propia unidad de compilación. El núcleo central consistirá en la función main(), que estará encargada de la inicialización de todo lo necesario para la obtención y almacenamiento de los resultados de los benchmarks, así como su invocación y procesamiento de resultados intermedios. Además, será necesaria la existencia de funciones auxiliares para el cálculo de la media aritmética, el valor mínimo, y la desviación estándar de un conjunto de resultados.

En la Figura 5.4 se puede ver el diagrama de actividad correspondiente al núcleo de *TORMENT OpenACC2016*, y el proceso de lanzamiento de los diferentes benchmarks, cuya elección y diseño será presentado en el siguiente apartado.

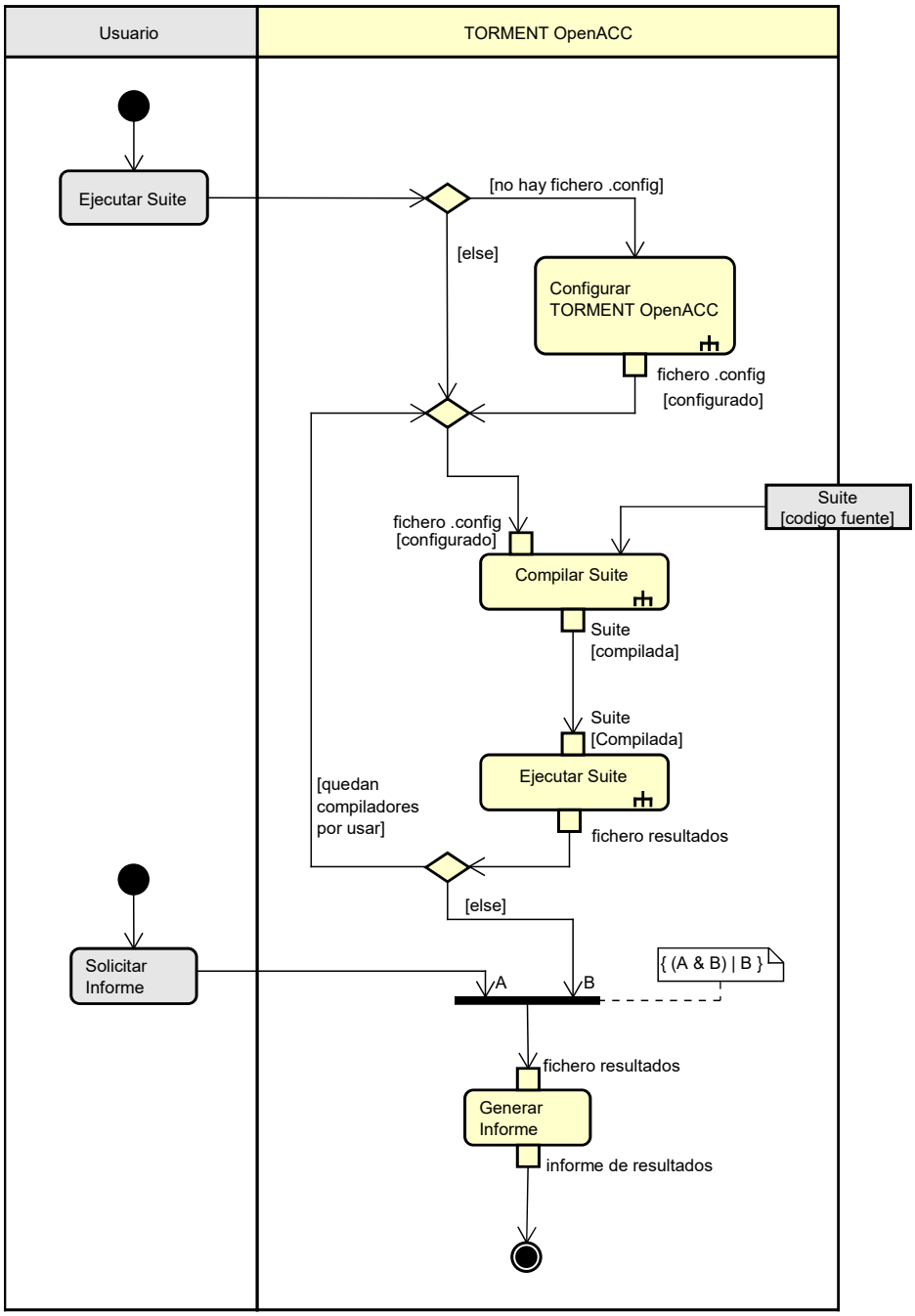


Figura 5.2: Diagrama de Actividad para la ejecución de la herramienta

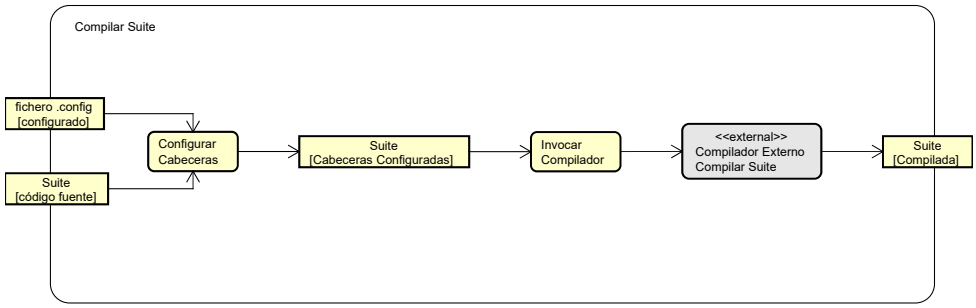


Figura 5.3: Diagrama de Actividad para la compilación de la herramienta

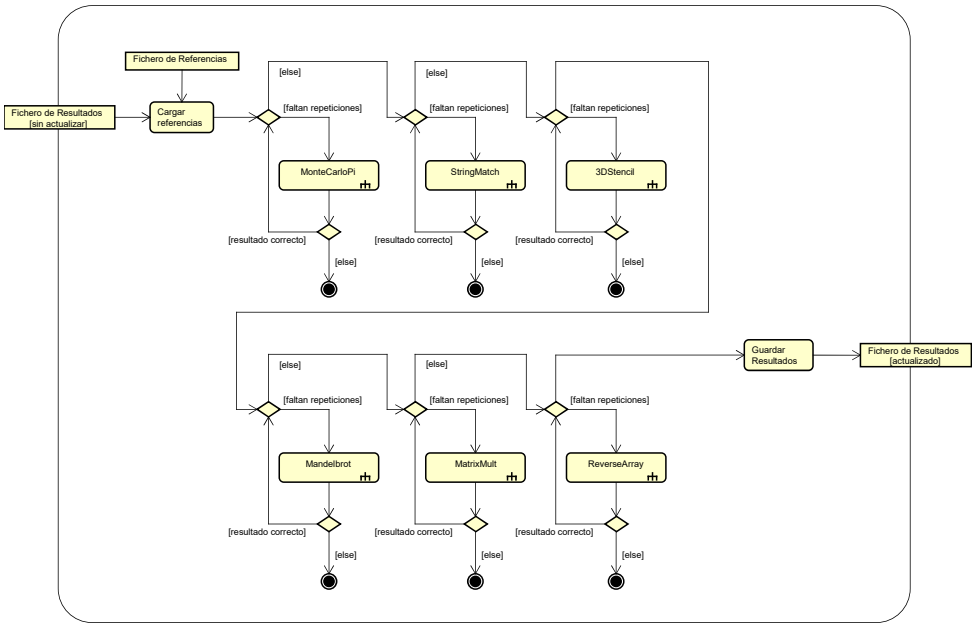


Figura 5.4: Diagrama de Actividad para el núcleo del programa de *TORMENT OpenACC2016*

5.2. Diseño de los Benchmarks

A continuación se enumeran los diferentes benchmarks que se han seleccionado como parte de *TORMENT OpenACC2016*. Para su elección, los criterios utilizados han sido los siguientes:

- Existencia de transferencias de memoria grandes o pequeñas entre el host y la GPU.
- Optimización por uso correcto de *shared memory* de la GPU.
- Efecto del uso de diferentes geometrías de bloque CUDA.
- Uso de offloading de funciones en la GPU para su invocación desde un kernel.
- Generación de números aleatorios en la GPU.
- Modificación de patrones de acceso a memoria para convertir accesos no coalescentes en accesos coalescentes.
- Comportamiento de códigos *embarrassingly parallel* en ausencia de otros factores.

5.2.1. T_MonteCarloPi: Aproximación de Pi por el método de Monte Carlo

Los Métodos de Monte Carlo [12] son una serie de algoritmos basados en el uso de números aleatorios. El nombre es una referencia a los casinos de Monte Carlo y a los juegos de azar.

Una aplicación de estos métodos puede usarse para aproximar el valor del número Pi. Este método se basa en la generación de puntos aleatorios en un cuadrado de lado unitario. Se comprueba si estos puntos se encuentran dentro de un cuarto de círculo de radio unitario y se acumula el total de puntos que cumplen dicha condición, como se muestra en la Figura 5.5. Finalmente, se aplica la siguiente fórmula:

$$\pi \approx \frac{4*P}{T}$$

Donde P es el número de puntos dentro del cuarto de círculo y T es el total de puntos generados.

Este es un benchmark que no tiene prácticamente transferencias de memoria y que realiza un cálculo computacional muy simple, pero puede ser optimizado en CUDA haciendo que cada hilo calcule varios puntos y utilizando la *shared memory* de los bloques para evitar accesos a memoria global. Un buen resultado de los compiladores en este benchmark dependerá de estos factores. Además, dado que hay que implementar una función para la generación de números aleatorios, se tendrá que hacer offloading de dicha función a la GPU.

En el fragmento de código de la Figura 5.6 se puede ver el pseudocódigo para este benchmark y en la Figura 5.7 se incluye el diagrama de actividad correspondiente.

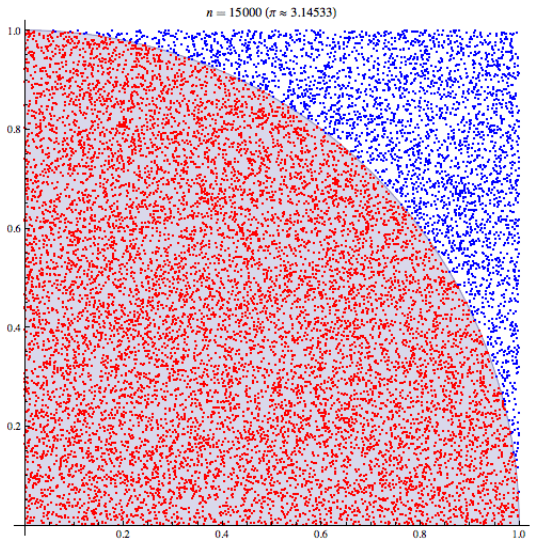


Figura 5.5: Representación gráfica de la aproximación de pi por el método de montecarlo

```
1 Set random seed
2 Set circle_points to 0
3 FOR each of the random coordinates
4     SET x and y random coordinates
5     IF coordinate is inside the circle THEN
6         INCREMENT circle_points
7     END IF
8 END FOR
9 Set Pi to 4 times circle_points divided by number of total coordinates
```

Figura 5.6: Pseudocódigo para el cálculo de Pi por el método de MonteCarlo

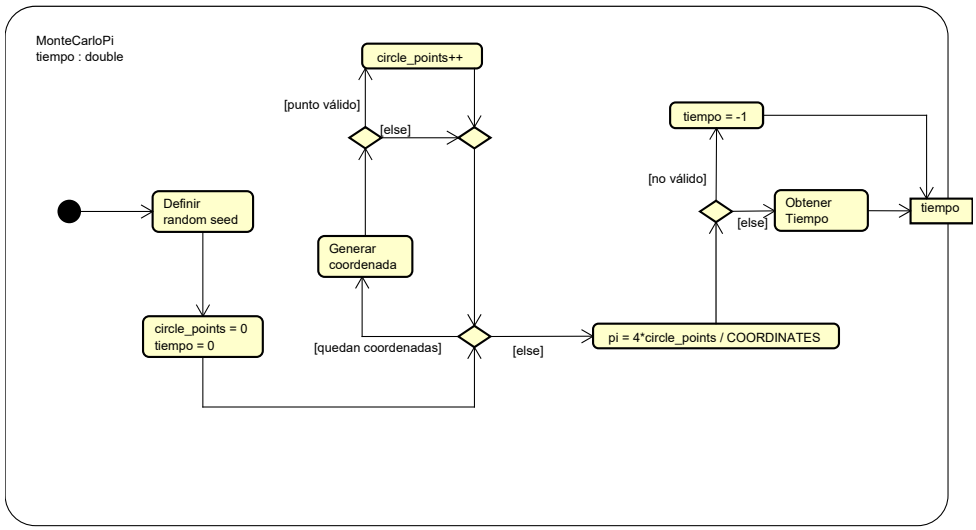


Figura 5.7: Diagrama de Actividad para el benchmark de aproximación de Pi por el método de Monte Carlo

```
1  Set size_b to size of big string
2  Set size_s to size of small string
3  Set result to −1
4  FOR each character in big string
5      FOR each character in small string
6          IF character in small string != character in big string THEN
7              break for loop
8          END IF
9      END FOR
10     IF index of inner for loop == size_s THEN
11         Set result to index of outer for loop
12         break for loop
13     END IF
14 END FOR
```

Figura 5.8: Pseudocódigo para el alineamiento de cadenas

5.2.2. T_StringMatch: Alineamiento de cadenas de caracteres

El algoritmo trivial de alineamiento de cadenas de caracteres consiste en la búsqueda de la primera posición en la cual una subcadena aparece en una cadena mayor, según el pseudocódigo presentado en la Figura 5.8.

Se tiene una cadena de caracteres grande, cuyo tamaño se encuentra en el orden de varios millones de caracteres, y varias cadenas más pequeñas, con tamaños en el orden de los miles de caracteres. En la cadena grande pueden existir varias ocurrencias de estas cadenas más pequeñas. El objetivo del algoritmo es encontrar la posición del primer carácter de la cadena grande a partir del cual existe una ocurrencia de la cadena pequeña que se está buscando.

Este algoritmo es interesante ya que combina una serie de factores que deben ser tenidos en cuenta por los compiladores que generen las versiones paralelas del código secuencial indicado. En primer lugar, tiene lugar la transferencia de memoria de varios elementos que, combinados, suponen varios megabytes de datos. La gestión de estas transferencias puede tener relevancia en el resultado. Por otra parte, deben realizarse reducciones sobre el indice de los posibles resultados, lo cual suele ser una operación bastante compleja de implementar. Además, las constantes comparaciones entre caracteres de una y otra cadenas suponen un importante gasto de tiempo en accesos a memoria global del dispositivo que pueden mejorarse mediante el correcto uso de la *shared memory* de la GPU.

En la Figura 5.9 se puede ver el diagrama de actividad correspondiente.

5.2.3. T_3DStencil: Stencil tridimensional de seis puntos

Los stencil son muy comunes en herramientas de benchmarking y consisten en la actualización de un array o matriz de acuerdo a un patrón predeterminado. En este caso, se ha ideado un Stencil tridimensional sobre una matriz de tres dimensiones en el que cada elemento de la matriz se actualiza con el valor de la media de los puntos adyacentes. La matriz inicial se inicializa con unos y ceros alternados, por lo tanto,

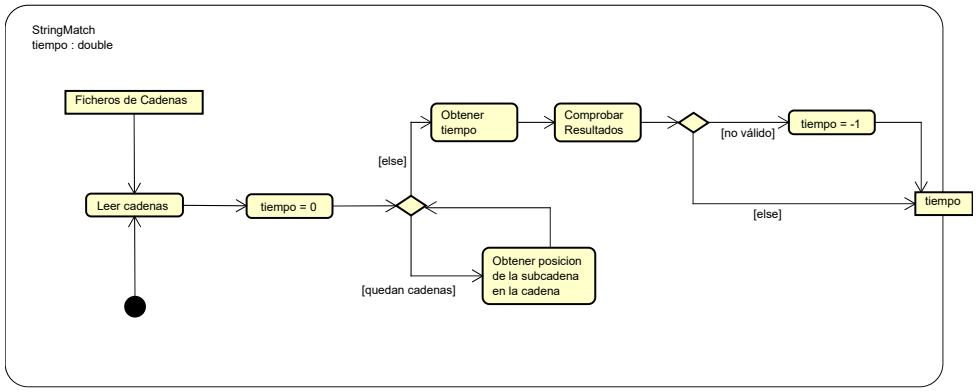


Figura 5.9: Diagrama de Actividad para el benchmark de alineamiento de cadenas

cada iteración de este stencil termina cambiando los ceros por unos y viceversa.

El pseudocódigo para este benchmark se presenta en la Figura 5.10.

Este benchmark es interesante porque es un problema en el que la ejecución en una GPU tradicionalmente ha sido muy eficiente. El problema incluye transferencias de memoria, con los datos de inicio y de fin. Esto puede originar problemas puesto que el calculo planteado es muy simple, buscando potenciar el efecto de los accesos a la memoria global del dispositivo. Debido a esto, se deberán ejecutar un número adecuado de iteraciones para que las transferencias iniciales y finales de memoria no representen la práctica totalidad del tiempo de ejecución del benchmark. La elección del número de iteraciones es complejo, un número muy bajo originaría problemas al calcular las métricas debido a la ausencia de transferencias de memoria en la versión secuencial de referencia, pero un número muy alto despreciaría los efectos de las optimizaciones aplicadas por los compiladores a estas transferencias de memoria. La elección de una correcta geometría de bloque también tiene un importante efecto en el uso de las *caches* en el acceso a datos y será también un factor a tener en cuenta en el resultado.

En la Figura 5.11 se incluye el diagrama de actividad correspondiente.

5.2.4. T_Mandelbrot: Generador del conjunto de Mandelbrot

Este es el más famoso de los conjuntos fractales [15] nombrado en honor de uno de los más importantes investigadores de este campo, Benoit Mandelbrot. Es un conjunto

```
1  Set matrix to alternate ones and zeros
2  Set matrix_copy to zeros
3  FOR each iteration of the benchmark
4      FOR each of the z indices
5          FOR each of the y indices
6              FOR each of the x indices
7                  Set counter to the sum of values of neighbouring cells
8                  Set number to the number of neighbouring cells
9                  Set matrix_copy's cell to counter divided by number
10             END FOR
11         END FOR
12     END FOR
13 END FOR
```

Figura 5.10: Pseudocódigo para el Stencil 3D

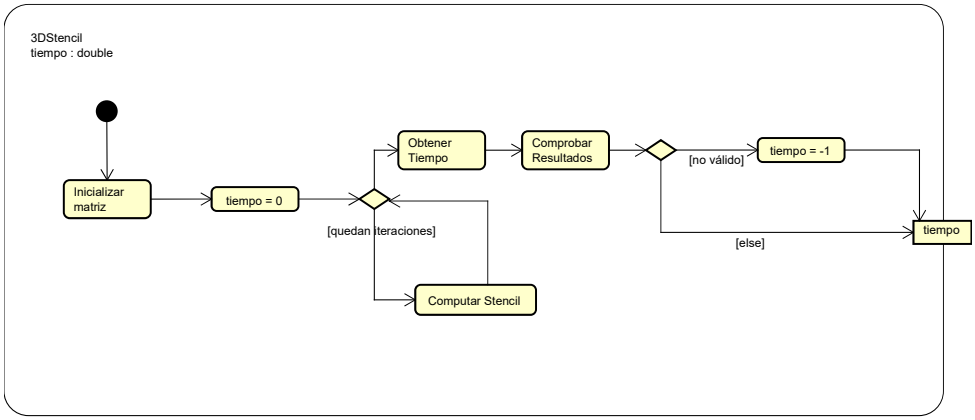


Figura 5.11: Diagrama de Actividad para el benchmark del Stencil 3D

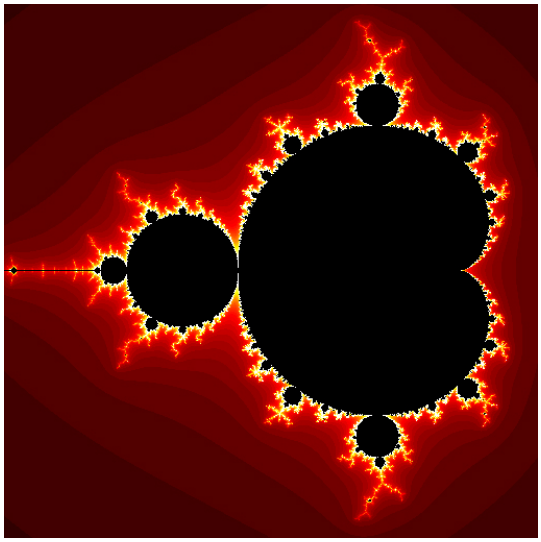


Figura 5.12: Representación gráfica del conjunto de mandelbrot por el algoritmo de tiempo de escape

que se define en el plano complejo siguiendo la siguiente sucesión por recursión:

$$z_0 = 0$$
$$z_{n+1} = z_n^2 + c$$

Si el valor que se le de a c hace que la sucesión quede acotada, entonces c pertenece al conjunto de mandelbrot.

Se propone utilizar el *algoritmo de tiempo de escape* en el benchmark. Mediante este algoritmo se puede obtener la imagen que se presenta en la Figura 5.12.

El algoritmo de tiempo de escape sigue el pseudocódigo que se indica en la Figura 5.13

Este benchmark tiene diversas cualidades que lo han llevado a ser seleccionado como miembro de *TORMENT OpenACC2016*. Cada posición es independiente del resto, por lo que en este caso no existe necesidad de uso de la *shared memory* de la GPU. Los accesos a memoria son también despreciables, puesto que sólo existe un acceso por posición para marcarla como perteneciente o no al conjunto de mandelbrot. La computación en cambio es bastante intensiva, aunque dependiente del número de iteraciones a llevar a cabo. En general, es un benchmark que debería escalar muy bien en la GPU, y que no debería causar problemas en los códigos generados por los diferentes compiladores.

En la Figura 5.14 se puede ver el diagrama de actividad correspondiente.

5.2.5. T_MatrixMult: Multiplicación de matrices

La multiplicación de matrices es una operación muy utilizada en muchos dominios de aplicación. Es una operación potencialmente muy costosa pero con mucho margen

```
1  FOR each pixel/cell in y direction
2      FOR each pixel/cell in x direction
3          Set iter to 0
4          Set max_iter to 10000
5          Set X0 to ( (x_index times 3.5) divided by x_size ) minus 2.5;
6          Set Y0 to ( (y_index times 2.0) divided by y_size ) minus 1.0;
7          Set X1 to 0;
8          Set X2 to 0;
9          WHILE (X1 times X1 plus Y1 times Y1 < 4) and (iter < max_iter)
10             Set tmp to X1 times X1 minus Y1 times Y1 plus X0
11             Set Y1 to 2 times X1 times Y1 plus Y0
12             Set X1 to tmp
13             INCREMENT iter
14         END WHILE
15         IF iter >= max_iter THEN
16             cell is inside mandelbrot set
17         END IF
18     END FOR
19 END FOR
```

Figura 5.13: Pseudocódigo para el conjunto de Mandelbrot por el método de tiempo de escape

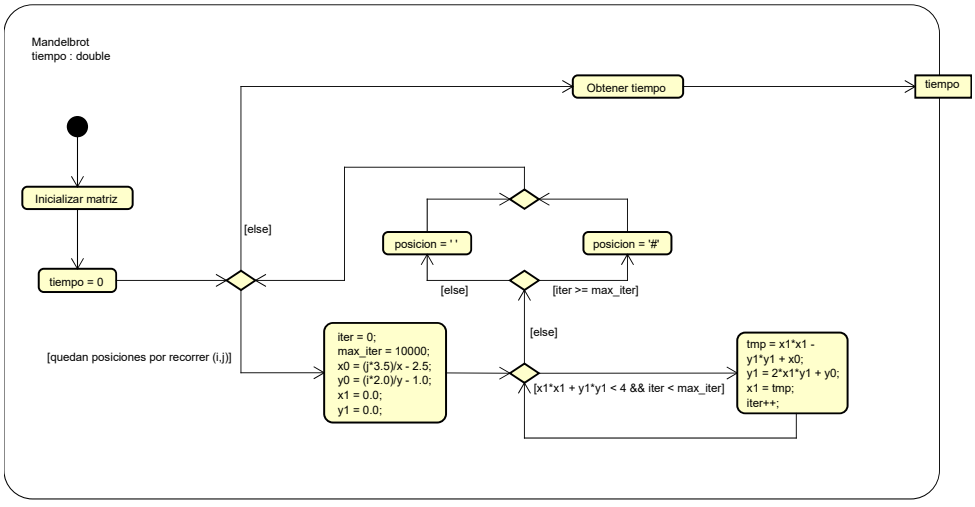


Figura 5.14: Diagrama de Actividad para el benchmark del Conjunto de Mandelbrot

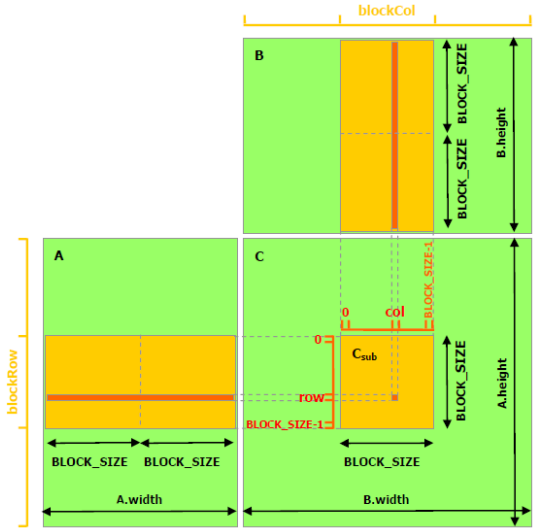


Figura 5.15: Multiplicación de matrices en CUDA usando *shared memory*

```
1 Set matrices a and b to initial values
2 FOR i=0 to size
3     FOR j=0 to size
4         Set c[i,j] to 0
5         FOR k=0 to size
6             Set c[i,j] to c[i,j] + a[i,k] * b[k,j]
7         END FOR
8     END FOR
9 END FOR
```

Figura 5.16: Pseudocódigo para multiplicación de matrices

de optimización utilizando las técnicas de optimización secuencial habituales: orden correcto de bucles, *tiling*, etc.

La multiplicación de matrices mediante el uso de GPU tiene una optimización fundamental en el uso de la *shared memory* para reducir el número de accesos a memoria. Para ello, hace falta además establecer una geometría de bloque adecuada a fin de optimizar las cachés. El procedimiento se muestra en la Figura 5.15.

Este benchmark presenta una cierta complejidad por la existencia de tres bucles anidados. Los compiladores de OpenACC tienen que elegir los diferentes niveles de paralelismo a utilizar para el procesamiento de dichos bucles, así como la geometría de bloque para obtener el mejor tiempo. El uso de la *shared memory* y la optimización de los accesos a memoria son fundamentales para obtener un buen resultado. Por estos motivos la multiplicación de matrices ha sido elegida para su implementación como benchmark de OpenACC. El pseudocódigo del algoritmo a utilizar puede verse en la Figura 5.16, así como el diagrama de actividad correspondiente en la Figura 5.17.

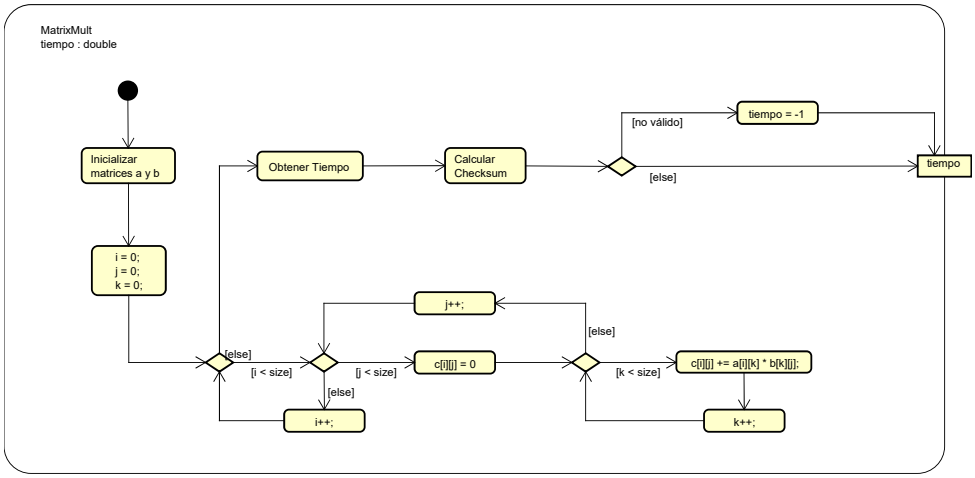


Figura 5.17: Diagrama de Actividad para el benchmark de multiplicación de matrices

5.2.6. T_ReverseArray: Inversión de un array de enteros

El último de los benchmarks elegidos es una operación aparentemente simple, pero que al ser trasladada a una GPU presenta una serie de dificultades que hay que solventar. El concepto en si es simple, darle la vuelta a un array de modo que el último elemento pase a ser el primero, el penúltimo pase a ser el segundo y así sucesivamente.

El principal problema se debe a que el algoritmo ejecutado en una GPU no realiza accesos coalescentes a memoria si se aplica directamente. La solución consiste en utilizar la *shared memory* como buffer intermedio en el que efectuar la reordenación de la parte correspondiente al bloque en ejecución. De este modo se consiguen accesos coalescentes. Puede verse gráficamente en la Figura 5.18.

El interés de este benchmark se basa en que es un fragmento de código en el que no hay realmente computación. El rendimiento se debe únicamente a la forma de realizar los accesos a memoria.

El pseudocódigo a implementar se presenta en la Figura 5.19 y el diagrama de actividad en la Figura 5.20.

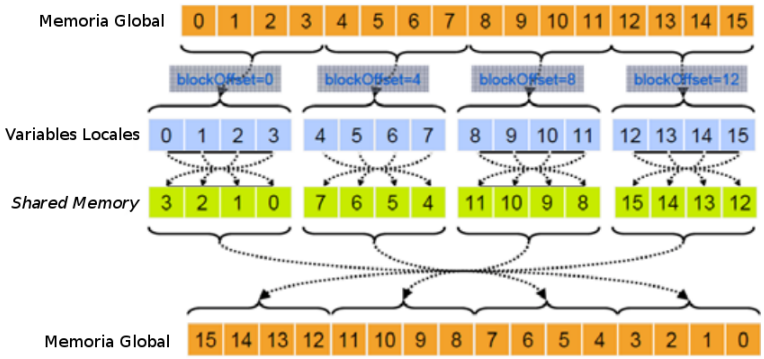


Figura 5.18: Inversión de un array de enteros en GPU

```
1 Set array to initial values
2 FOR i=0 to size
3     Set reversed[size - i -1] to array[i]
4 END FOR
```

Figura 5.19: Pseudocódigo para la inversión de un array

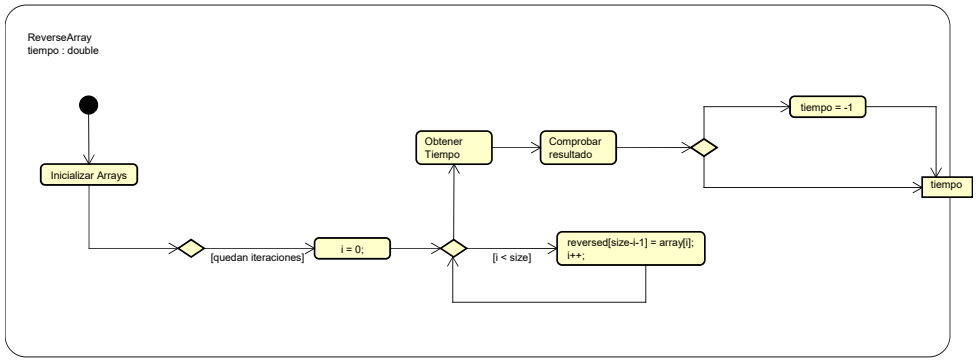


Figura 5.20: Diagrama de Actividad para el benchmark de inversión de un array

5.3. Diseño de la Métrica *TORMENT_ACC2016*

Para la elección de la métrica denominada *TORMENT_ACC2016* hemos decidido optar por la metodología utilizada por SPEC [8]. El *System Performance Evaluation Council*, comúnmente conocido como SPEC, es un referente ampliamente conocido en cuanto a benchmarking y análisis de rendimiento se refiere. Una de sus fortalezas es el reconocer que los benchmarks envejecen en función del paso del tiempo y, en consecuencia, deben ser actualizados.

SPEC utiliza la siguiente metodología. En primer lugar, cada programa devuelve su tiempo de ejecución y se calcula el *SPECratio*, que consiste en el ratio obtenido de dividir un tiempo de ejecución de referencia suministrado por SPEC entre el tiempo de ejecución obtenido. Finalmente, se obtiene la media geométrica de todos los SPECratios del conjunto de benchmarks [14].

Nuestra propuesta sigue una idea similar a la de SPEC, pero con algunas variaciones. En primer lugar, la ejecución de los benchmarks que componen *TORMENT OpenACC2016* devuelven tres valores. *Peak Time* es el mejor tiempo de ejecución obtenido, medido en segundos. *Average Time* es el tiempo medio de todas las ejecuciones del benchmark, también en segundos. Finalmente, *Standard Deviation* es la desviación estándar del conjunto de medidas e indica la variabilidad obtenida en las ejecuciones del benchmark. Con los tiempos *peak* y *average* se calcula un ratio con respecto a los tiempos de referencia suministrados con la herramienta, y que son los tiempos de ejecución secuencial del benchmark en una máquina de referencia. Una vez ejecutados todos los benchmarks, se obtiene la media armónica de todos los ratios, tanto para *peak time* como *average time*. Estos valores que se obtienen son *TORMENT_ACC2016 peak* y *TORMENT_ACC2016 average*.

La principal diferencia entre *TORMENT OpenACC2016* y SPEC, aparte de la metodología de toma de tiempos de ejecución, consiste en el uso de la media armónica en lugar de la media geométrica. Esta decisión se fundamenta en el hecho de que, para los objetivos de *TORMENT OpenACC2016*, la media armónica es más adecuada que la media geométrica. En primer lugar, aunque la media geométrica siempre produce una ordenación consistente, no necesariamente produce la ordenación correcta [14], ya que esta media no es inversamente proporcional al tiempo de ejecución. En cambio, la media armónica si que es inversamente proporcional al tiempo de ejecución, lo que hace que sea una media correcta para expresar ratios. Estas afirmaciones son compartidas por otros trabajos, como por ejemplo [16].

El diseño establece una serie de criterios para la implementación de *TORMENT OpenACC2016*, proceso que será analizado en el siguiente capítulo, resaltando los aspectos más importantes de la codificación de las diferentes partes que componen la herramienta y mostrando fragmentos del código que se escriba.

Capítulo 6

Implementación y pruebas

En este capítulo se documentará el proceso de implementación de la suite *TORMENT OpenACC2016* desde las etapas de análisis y diseño previamente realizadas. El capítulo comienza con la documentación relativa a la creación de los scripts, encargados de ocultar al usuario toda la complejidad de *TORMENT OpenACC2016*. Posteriormente se presentará el programa principal de la suite, a cargo del lanzamiento y recopilación de la información de los diferentes benchmarks. Por último, se presentan los benchmarks implementados.

6.1. Implementación de los Scripts

Los scripts siguen la estructura definida en el capítulo de diseño. A continuación se muestran los detalles de implementación más relevantes de los diferentes scripts.

6.1.1. Configuración: configure.sh

El fichero configure.sh es el script que se encarga de las tareas de configuración de *TORMENT OpenACC2016*. Se necesitan conocer los siguientes datos:

- Ruta a las librerías de CUDA.
- Ruta al compilador de Nvidia nvcc.
- Ruta a los compiladores de OpenACC instalados.
- Comando de ejecución, en caso de usar por ejemplo colas *SLURM*.

Para obtener estos datos, el script hace uso del comando find. Puede verse un ejemplo del código utilizado para localizar las librerías de CUDA en la Figura 6.1. El resto de código para la búsqueda de rutas a compiladores es prácticamente idéntico. En la Figura 6.2 se muestra el código para la solicitud del comando de ejecución.

```
1  ##########################################
2  # Find CUDA library
3  ##########################################
4   printf "$($yel)Searching for CUDA library. "
5   printf "This may take some time...$($ncl)\n"
6
7  #Use find to search cuda libraries
8  cuda=( $(find / -type f -name libcuda.so* 2>/dev/null) )
9
10 #Iterate through all the possibilities and ask for confirmation
11 done=false
12 for path in "${cuda[@]}"
13 do
14     while true;
15     do
16          printf "\t$($blu)$path$($ncl)\n"
17          printf "\tIs this path correct?\t"
18          read yn
19          case $yn in
20               [Yy]* ) done=true; break;;
21               [Nn]* ) break;;
22               * ) printf "Please answer yes or no.\n";;
23          esac
24     done
25     if $done; then
26               cuda=$path
27               break
28     else
29               cuda=""
30     fi
31 done
32
33 #If found, add to configuration file
34 if [ "$cuda" == "" ]; then
35      printf "$($red)CUDA library not found$($ncl)\n\n"
36 else
37      printf "$($grn)CUDA library found$($ncl)\n"
38      printf "$($blu)$cuda$($ncl)\n\n"
39     cuda=$(dirname $cuda)
40     echo "cuda=$cuda" > .config
41 fi
42 # Find CUDA library ##########################
```

Figura 6.1: Código para la busqueda de librerías CUDA en el script de configuración

```
1  ##########################################
2  # Run command: In case the user want to use a special command (slurm, etc)
3  ##########################################
4   printf "$($yel)Run Command\n"
5   printf "If you are going to use another tool for running the program, "
6   printf "like Slurm or similar, please type the required run command:$($ncl)\n"
7   printf "\tDefault :[]\n"
8  done=false
9  while true;
10 do
11      printf "\tPlease type the run command: "
12     read run_command
13      printf "\tis [$($blu)$run_command$($ncl)] correct?\t"
14     read yn
15     case $yn in
16         [Yy]* ) done=true;;
17         [Nn]* ) ;;
18         * ) printf "Please answer yes or no.\n";;
19     esac
20     if $done; then
21         echo "run_command=\"$run_command\"" >> .config
22         break
23     fi
24 done
```

Figura 6.2: Código para la petición del comando de ejecución en el script de configuración

```
# Load configuration variables stored in .config or launch configuration script
if [ ! -f .config ]; then
    ./configure.sh
else
    . .config
fi
```

Figura 6.3: Código para la comprobación del fichero .config

6.1.2. Ejecución: run.sh

Siguiendo la estructura planteada en el diseño del script que se implementa en el fichero run.sh, este script se encarga de la ejecución de la suite *TORMENT OpenACC2016*. Tras ser invocado por el usuario, este script realiza lo siguiente:

- Comprobación de que existe el fichero .config, y de no ser así, lanzar el script de configuración configure.sh.
- Para cada compilador, compilar, ejecutar y eliminar ficheros binarios e intermedios.
- Lanzar el script de generación de resúmenes de resultados con los datos obtenidos tras la ejecución de los benchmarks.

El código correspondiente a la comprobación del fichero .config puede verse en la Figura 6.3. Un fragmento del código para la compilación, ejecución y limpieza de binarios se presenta en la Figura 6.4.

6.1.3. Generación de Resúmenes: gen_report.sh

Finalmente, el script encargado de la generación de los resúmenes HTML con los resultados obtenidos tras la ejecución de la suite se implementa en el fichero gen_report.sh.

En este fichero se combina tanto la plantilla HTML que se generará, como la obtención de datos del sistema y el procesamiento de los resultados numéricos sobre tiempo de ejecución y ratios que se obtienen de la herramienta.

El resultado es un fichero HTML, con una hoja de estilo CSS para su correcta presentación.

El código necesario para obtener datos del sistema que posteriormente serán incluidos en el report se puede ver en la Figura 6.5. Estos datos se muestran en el fichero HTML antes de los datos de benchmarking. El código del script para la generación de esta parte del resumen de resutados se presenta en la Figura 6.6. Finalmente, la parte del script encargada del procesamiento de los datos de benchmarking se puede ver en las Figuras 6.7 y 6.8.

```
1  # Compile and run using PGI, if exists
2  if [ "$pgcc" != "" ]; then
3       printf "\n$($yel)Compiling using PGI$($ncl)\n"
4
5       #Write the PGI header file
6       echo "#ifndef _MAIN_H_" > src/main.h
7       echo "#define _MAIN_H_" >> src/main.h
8       echo "#define PGI" >> src/main.h
9       echo "#define COMPILER \"PGI\"" >> src/main.h
10      echo "#define REF 2" >> src/main.h
11      echo "#endif" >> src/main.h
12
13      #Call make
14      FLAGS="-fast -Minfo=all -ta=tesla" \
15      COMP=PGI \
16      CC=$pgcc \
17      make -C src > temp/pgi.log 2> temp/pgi2.log
18
19      #Wait a bit to make sure NFS systems are correctly updated
20      sleep 5
21
22      #Run the benchmark suite
23      printf "$($yel)Running benchmark suite$($ncl)\n"
24      cd src
25      $run_command ./opentest
26      cd ..
27
28      #Call make clean
29      make clean -C src > /dev/null 2> /dev/null
30 fi
```

Figura 6.4: Código para la compilación, ejecución y limpieza de binarios del script run.sh

```
model_name=$(cat /proc/cpuinfo | grep "model name" | sort -u | cut -d: -f2)
cores=$(cat /proc/cpuinfo | grep " siblings " | sort -u | cut -d: -f2)
arch=$(lscpu | grep "Architecture" | sort -u | cut -d: -f2 | \
    sed -e 's/^[[:space:]]*//')
maxcpu=$(lscpu | grep "CPU max MHz" | sort -u | cut -d: -f2 | \
    sed -e 's/^[[:space:]]*//')
mincpu=$(lscpu | grep "CPU min MHz" | sort -u | cut -d: -f2 | \
    sed -e 's/^[[:space:]]*//')
l1=$(lscpu | grep "L1d cache" | sort -u | cut -d: -f2 | \
    sed -e 's/^[[:space:]]*//')
l2=$(lscpu | grep "L2 cache" | sort -u | cut -d: -f2 | \
    sed -e 's/^[[:space:]]*//')
l3=$(lscpu | grep "L3 cache" | sort -u | cut -d: -f2 | \
    sed -e 's/^[[:space:]]*//')
mem=$(cat /proc/meminfo | grep "MemTotal" | sort -u | cut -d: -f2 | \
    sed -e 's/^[[:space:]]*//')
kernel=$(uname -mrs)

OLDIFS=$IFS
IFS=$'\n'
gpu=($(lspci | egrep "VGA|3D" | grep "NVIDIA" | sort -u | cut -d: -f3))
IFS=$OLDIFS

OLDIFS=$IFS
IFS=$'\n'
results=()
while read line
do
    results=("${results[@]}" "$line")
done < results.txt
IFS=$OLDIFS
```

Figura 6.5: Código para la obtención de datos del host

```
1  echo "<div class='info'>"                                    >> results.html
2  echo "<h3>System Information</h3>"                           >> results.html
3  echo "<div class='sub-info-full'>"                           >> results.html
4  echo "<p>Benchmark version: 0.93</p>"                        >> results.html
5  echo "<p>Hostname: $(hostname)</p>"                          >> results.html
6  echo "<p>Username: $(id -un)</p>"                            >> results.html
7  echo "</div>"                                                >> results.html
8  echo "<div class='sub-info-full'>"                           >> results.html
9  echo "<h4>Software stack info</h4>"                          >> results.html
10 echo "<p>Kernel: "$kernel"</p>"                              >> results.html
11 echo "<p>GCC: "$(gcc --version | grep "gcc")"</p>"           >> results.html
12 echo "<p>NVCC: "$(nvcc --version | grep "release")"</p>"     >> results.html
13 if [ "$pgcc" != "" ]; then
14     echo "<p>PGI Compiler: \
15         "$($pgcc --version | grep "pgcc")"</p>"              >> results.html
16 fi
17 if [ "$pgcc" != "" ]; then
18     echo "<p>OpenUH Compiler: \
19         "$($openuh --version 2>&1 /dev/null|grep "OpenUH")"</p>"
20                                                             >> results.html
21 fi
22 if [ "$accull" != "" ]; then
23     echo "<p>accULL Compiler: "$($accull | grep "Release")"</p>"
24                                                             >> results.html
25 fi
26 echo "</div>"                                                >> results.html
27 echo "<div class='sub-info'>"                                >> results.html
28 echo "<h4>CPU info</h4>"                                     >> results.html
29 echo "<p>Model: $model_name</p>"                             >> results.html
30 echo "<p>Architecture: $arch</p>"                            >> results.html
31 echo "<p>Number of Cores: $cores</p>"                        >> results.html
32 echo "<p>Max MHz: $maxcpu MHz</p>"                           >> results.html
33 echo "<p>Min MHz: $mincpu MHz</p>"                           >> results.html
34 echo "<p>L1 Cache: $l1</p>"                                  >> results.html
35 echo "<p>L2 Cache: $l2</p>"                                  >> results.html
36 echo "<p>L3 Cache: $l3</p>"                                  >> results.html
37 echo "<p>RAM: $mem</p>"                                      >> results.html
38 echo "</div>"                                                >> results.html
39 echo "<div class='sub-info'>"                                >> results.html
40 echo "<h4>GPU(s) info</h4>"                                  >> results.html
41 for (( i=0; i<${#gpu[@]}; i++))
42 do
43     echo "<p>GPU$i:${gpu[$i]}</p>"                           >> results.html
44 done
45 echo "</div>"                                                >> results.html
46 echo "</div>"                                                >> results.html
```

Figura 6.6: Código para la generación de información del host en el resumen

```
for ((I=0; I<${#results[@]}; I++))
do
    elements=(${results[$I]})
    sequential=(${results[0]})
    cuda=(${results[1]})
    echo "<div>"                                                    >> results.html
    echo "<h3>${elements[0]} Compiler Results:</h3>"                >> results.html
    echo "<table border='1'>"                                       >> results.html
    echo "<tr>"                                                     >> results.html
    echo "<th colspan='6'>${elements[0]}</th>"                      >> results.html
    echo "</tr>"                                                    >> results.html
    echo "<tr>"                                                     >> results.html
    echo "<th>Benchmark</th>"                                       >> results.html
    echo "<th>Peak Time (s)</th>"                                   >> results.html
    echo "<th>Peak Ratio</th>"                                      >> results.html
    echo "<th>Avg. Time (s)</th>"                                   >> results.html
    echo "<th>Avg. Ratio</th>"                                      >> results.html
    echo "<th>Std. Deviation</th>"                                  >> results.html
    echo "</tr>"                                                    >> results.html
    for ((i=1; i<$(( ${#elements[@]}-2 )); i+=6))
    do
        echo "<tr>"                                                 >> results.html
        for ((j=0; j<6; j++))
        do
            if [ $j -eq 2 ]
            then
             printf "<td>%0.2f</td>" ${elements[$(( $i+$j ))]}   >> results.html
             elif [ $j -eq 4 ]
            then
             printf "<td>%0.2f</td>" ${elements[$(( $i+$j ))]}   >> results.html
            else
            echo "<td>${elements[$(( $i+$j ))]}</td>"             >> results.html
             fi
        done
        echo "</tr>"                                              >> results.html
    done
    echo "</tr>"                                                  >> results.html
    echo "</table>"                                               >> results.html
```

Figura 6.7: Código para la generación de las tablas de resultados del benchmarking

```
    if [ $l -gt 1 ]
    then
        echo "<div class='score'>"                                    >> results.html
        echo "<div class='sub-score'>"                                >> results.html
        echo "<h4><b>TORMENT_ACC_0.93 Peak</b>:"                      >> results.html
        j=$(( $i+1 ))
        printf " %0.2f</h4>" ${elements[$i]}                          >> results.html
        echo "</div>"                                                 >> results.html
        echo "<div class='sub-score'>"                                >> results.html
        echo "<h4><b>TORMENT_ACC_0.93 Average</b>:"                   >> results.html
        printf " %0.2f</h4>" ${elements[$j]}                          >> results.html
        echo "</div>"                                                 >> results.html
        a=${elements[$i]}
        b=${sequential[$i]}
        temp=$(awk -v a=$a -v b=$b 'BEGIN { print a / b }')
        echo "<div class='sub-score'>"                                >> results.html
        echo "<p><b>Speedup vs Sequential (peak)</b>:"                >> results.html
        printf " %0.2fx</p>\n" $temp                                  >> results.html
        echo "</div>"                                                 >> results.html
        a=${elements[$j]}
        b=${sequential[$j]}
        temp=$(awk -v a=$a -v b=$b 'BEGIN { print a / b }')
        echo "<div class='sub-score'>"                                >> results.html
        echo "<p><b>Speedup vs Sequential (average)</b>:"             >> results.html
        printf " %0.2fx</p>\n" $temp                                  >> results.html
        echo "</div>"                                                 >> results.html
        a=${elements[$i]}
        b=${cuda[$i]}
        temp=$(awk -v a=$a -v b=$b 'BEGIN { print a / b }')
        echo "<div class='sub-score'>"                                >> results.html
        echo "<p><b>Speedup vs CUDA (peak)</b>:"                      >> results.html
        printf " %0.2fx</p>\n" $temp                                  >> results.html
        echo "</div>"                                                 >> results.html
        a=${elements[$j]}
        b=${cuda[$j]}
        temp=$(awk -v a=$a -v b=$b 'BEGIN { print a / b }')
        echo "<div class='sub-score'>"                                >> results.html
        echo "<p><b>Speedup vs CUDA (average)</b>:"                   >> results.html
        printf " %0.2fx</p>\n" $temp                                  >> results.html
        echo "</div>"                                                 >> results.html
        echo "</div>"                                                 >> results.html
    fi
    echo "</div>"                                                     >> results.html
done
```

Figura 6.8: Código para la generación del resumen de resultados del benchmarking

6.2. Implementación del Núcleo de la Herramienta

El programa principal es la parte compilada de *TORMENT OpenACC2016*, y su cometido consiste en gestionar la ejecución de los benchmarks y procesar los tiempos de ejecución de los mismos. Con estos tiempos de ejecución y los tiempos de referencia incluidos con la suite, se calculan los respectivos ratios. Una vez finalizada la ejecución del benchmark se guardarán los datos en el fichero de resultados.

El punto de entrada del programa, la función main(), se encuentra en el fichero main.c. Ya que los benchmarks se encuentran definidos en sus propias unidades de compilación, es necesario disponer de la declaración de las funciones de invocación de los mismos. Estos prototipos de función se encuentran en benchmarks/benchmark.h. En todos los casos, estas funciones devuelven el tiempo de ejecución del benchmark correspondiente.

Como se ha comentado anteriormente, es necesario disponer de información del compilador utilizado y, por motivos de compatibilidad con los compiladores en un estado de desarrollo más temprano, esta información no se puede pasar durante la compilación con los argumentos -D habituales. Debido a esto, el fichero main.c dispone de un #include de un fichero de cabecera con la información necesaria que se edita antes de la compilación. Este fichero es main.h.

En el caso de que uno de los benchmarks de un resultado erróneo, la función devuelve el valor -1.0 y la suite aborta la ejecución de los benchmarks restantes. Esto es una medida de seguridad para evitar resultados parciales que puedan distorsionar los resultados, especialmente la métrica *TORMENT_ACC2016*.

Un fragmento del fichero main.c se puede encontrar en la Figura 6.9. Por motivos de espacio, los includes de la librería estándar de C, así como varias lineas en blanco han sido omitidas. Las funciones auxiliares cuyos prototipos pueden verse al comienzo del programa tampoco han sido incluidas en la Figura. Dentro de la función main(), el código de invocación de uno de los benchmarks se puede ver en la Figura 6.10. Todos los benchmarks se invocan de la misma forma por lo que no se incluyen las invocaciones del resto.

Como puede apreciarse, el código ha sido implementado de la forma más sencilla posible. Esto se debe al estado de desarrollo de algunos de los compiladores a los que se pretende dar soporte. Estos compiladores aún no están suficientemente maduros y los sistemas de traducción fuente a fuente tienen problemas con algunos elementos de la sintaxis de C típica, como por ejemplo, los punteros a función, cuyo uso facilitaría la invocación de los benchmarks y mejoraría la mantenibilidad del código.

```
1  #include "main.h"
2  #include "benchmarks/benchmark.h"
3  #define REPS 10 //Number of repetitions for each benchmark run
4  #define BENCHMARKS 6 //Number of benchmarks in the suite
5
6  double average(double* t, int reps);
7  double peaktime(double*t, int reps);
8  double stddev(double avg, double* t, int reps);
9
10 int main(){
11     double avg, peak, sdev, acum_peak, acum_avg, ratio_peak, ratio_avg;
12     double harm_peak, harm_avg;
13     double t[REPS];
14     double references[BENCHMARKS*2];
15     double* rp = &references[0];
16     char results[BENCHMARKS+2][128];
17     FILE* file;
18     int i;
19     for(i = 0; i < BENCHMARKS+2; i++){
20         memset(results[i], 0, 128);
21     }
22     // Open the references file for ratio calculation
23     file = fopen("data/reference.txt", "r");
24     if (file == NULL){
25         printf("Error opening reference file\n");
26         exit(EXIT_FAILURE);
27     }
28     for(i = 0; i < BENCHMARKS*2; ++i){
29         fscanf(file, "%lf", &references[i]);
30     }
31     // Open the results file where results will be stored
32     file = fopen("results.txt", "a");
33     if (file == NULL){
34         printf("Error opening results file\n");
35         exit(EXIT_FAILURE);
36     }
37     // Write the compiler used in the results file
38     sprintf(results[0], COMPILER);
39     <...Benchmark invocation code skipped...>
40     harm_peak = BENCHMARKS / acum_peak;
41     harm_avg = BENCHMARKS / acum_avg;
42     sprintf(results[BENCHMARKS+1], "\t%f\t%f\n", harm_peak, harm_avg);
43     printf("All benchmarks completed\n");
44
45     for(i = 0; i < BENCHMARKS+2; i++){
46         fprintf(file, "%s", results[i]);
47     }
48     exit(EXIT_SUCCESS);
49 }
```

Figura 6.9: Fragmentos de código pertenecientes al fichero main.c

```
1  /* Run the MonteCarloPi benchmark ********************************/
2   printf ("Running MonteCarloPi benchmark:\n");
3  bench_monteCarloPi();
4  for (i = 0; i < REPS; ++i){
5       t[i] = bench_monteCarloPi();
6       if (t[i] < 0){
7            printf ("Error running the benchmark suite. Aborting.\n");
8            exit (EXIT_FAILURE);
9       }
10 }
11 avg = average(t, REPS);
12 peak = peaktime(t, REPS);
13 sdev = stddev(avg, t, REPS);
14  ratio_peak = *rp / peak;
15 rp++;
16  ratio_avg  = *rp / avg;
17 rp++;
18 acum_peak += 1/ratio_peak;
19 acum_avg  += 1/ratio_avg;
20  printf ("\tPeak Time: %f sec.\n", peak);
21  printf ("\tAvg. Time: %f sec.\n", avg);
22  printf ("\tStd. Dev.: %f sec,\n", sdev);
23  sprintf ( results [1], "\tMonteCarloPi\t %f\t %f\t %f\t %f\t %f",
24          peak, ratio_peak , avg, ratio_avg , sdev);
25 /* MonteCarloPi End *********************************************/
```

Figura 6.10: Fragmento de código para la invocación de los benchmarks, incluido en el fichero main.c

6.3. Implementación de los Benchmarks

En esta sección se detalla la implementación de los benchmarks de *TORMENT OpenACC2016*. En todos los casos, las funciones de invocación de los benchmarks se encuentra en el fichero benchmarks/benchmark.h. Estas funciones ejecutan el código correspondiente a cada benchmark y devuelven el tiempo de ejecución, medido en segundos.

Como *TORMENT OpenACC2016* ofrece resultados de ejecución de código secuencial, CUDA y OpenACC, los ficheros en los que se encuentran los benchmarks contienen código, utilizando compilación condicional, para CUDA y OpenACC (la versión secuencial es igual que la versión OpenACC pero ignorando los pragmas.

Dado que para compilar código CUDA, el compilador NVCC requiere que el fichero tenga extensión .cu, y para evitar duplicación de código, se ha tomado la decisión de generar un fichero .cu por cada fichero .c que simplemente sea un enlace simbólico al fichero en el que se encuentra el código. De esta manera se evita una probable fuente de problemas durante el desarrollo y mantenimiento.

6.3.1. T_MonteCarloPi: Aproximación de Pi por el método de Monte Carlo

El fichero benchmarks/montecarlo.c contiene el código necesario para la aproximación de Pi por el método de Monte Carlo. Este benchmark utiliza una función de generación de números aleatorios que ha sido copiada de la implementación de la función srand() de la librería estándar de C. Esto ha sido necesario ya que no pueden ejecutarse desde la GPU funciones que estén en el host. Puede verse la implementación de la función getRandom() en la Figura 6.11. Para que esta función pueda usarse en la GPU se utiliza el atributo __device__ en CUDA y la directiva #pragma acc routine en OpenACC.

Dado que se utilizan números aleatorios y diferentes formas de establecer el nivel de paralelismo, no es posible asegurar la generación de las mismas semillas entre diferentes compiladores. Debido a esto, la comprobación de que el benchmark ha sido correcto se hace mediante el error cometido en la aproximación de Pi. Aquellas ejecuciones que se alejen de lo esperado serán consideradas como no válidas.

El código secuencial y OpenACC del algoritmo del benchmark se puede ver en la Figura 6.12, así como el código del *kernel* CUDA en la Figura 6.13.

```
1  /*
2   *  Redefinition of GNU's srand method to make possible its use in the GPU.
3   *  Using CURAND is not possible because of the portability across OpenACC
4   *  compilers.
5   */
6  __CUDA_ATTR__ int getRandom(unsigned int* seed)
7  {
8      unsigned int next = *seed;
9      int result;
10
11     next *= 1103515245;
12     next += 12345;
13     result = (unsigned int) (next/65536) % 2048;
14
15     next *= 1103515245;
16     next += 12345;
17     result <<= 10;
18     result ^= (unsigned int) (next/65536) % 1024;
19
20     next *= 1103515245;
21     next += 12345;
22     result <<= 10;
23     result ^= (unsigned int) (next/65536) % 1024;
24
25     *seed = next;
26     return result;
27 }
```

Figura 6.11: Implementación de la función getRandom() en el benchmark MonteCarloPi

```
1      double x, y, d;
2      unsigned int seed;
3      int count = 0;
4      int i;
5      #pragma acc parallel loop private(i, d, x, y, seed) reduction(+:count)
6      for(i = 0; i < COORD_NUM; ++i){
7          // Use a different seed each time, for portability reasons
8          seed = 1987 ^ i*27;
9          x = (double) getRandom(&seed) / (double) RAND_MAX;
10         y = (double) getRandom(&seed) / (double) RAND_MAX;
11
12         d = sqrt(x*x + y*y);
13         if(d <= 1.0){
14             ++count;
15         }
16     }
17
18     pi = (count / (double) COORD_NUM) * 4.0;
```

Figura 6.12: Código secuencial y OpenACC del algoritmo del benchmark MonteCarloPi

```
1  /*
2   *  CUDA Kernel for the estimation of PI using the monte carlo method
3   */
4  __CUDA_GLOBAL__ void piKernel(const unsigned long int triesPerThread,
5                    unsigned long int* hits)
6  {
7      unsigned int seed;
8      int gid, tid, bid;
9      int lhits;
10     float x, y;
11     extern __shared__ unsigned long int sdata[];
12
13     gid = (blockIdx.x*blockDim.x) + threadIdx.x;
14     tid = threadIdx.x;
15     bid = blockIdx.x;
16     lhits = 0;
17
18     seed = 1987 ^ gid*27;
19
20     for (int i = 0; i < triesPerThread; ++i){
21         x = (float) getRandom(&seed) / (float) RAND_MAX;
22         y = (float) getRandom(&seed) / (float) RAND_MAX;
23
24         float d = sqrt(x*x + y*y);
25         if (d <= 1.0f){
26             ++lhits;
27         }
28     }
29     sdata[tid] = lhits;
30     __syncthreads();
31     if (tid == 0){
32         for (int i = 1; i < blockDim.x; ++i){
33             lhits += sdata[i];
34         }
35
36     hits[bid] = lhits;
37     }
38 }
```

Figura 6.13: Código CUDA del algoritmo del benchmark MonteCarloPi

```
1   #pragma acc data copyin(S[0:sizeS], B[0:sizeB]) copy(C[0:sizeB])
2   {
3   #pragma acc parallel loop private(startB) firstprivate (result, sizeS, sizeB)
4   for (startB = 0; startB <= sizeB-sizeS; startB++){
5       C[startB] = 0;
6       if (startB <= result) {
7           int ind;
8           for(ind = 0; ind < sizeS; ind++){
9               if (S[ind] != B[startB+ind]) break;
10          }
11          if (ind == sizeS){
12              result = startB;
13              C[startB] = 1;
14          }
15      }
16  }
17  }
18  result = -1;
19  for (startB = 0; startB <= sizeB-sizeS; startB++){
20      if (C[startB] == 1){
21          result = startB;
22          break;
23      }
24  }
25  free(C);
26  return result;
```

Figura 6.14: Fragmento de código secuencial y OpenACC para StringMatch

6.3.2. T_StringMatch: Alineamiento de cadenas de caracteres

El benchmark StringMatch se implementa en el fichero benchmarks/stringMatch.c, con algunas funciones auxiliares definidas en el fichero de cabecera benchmarks/stringMatch.h. Estas funciones auxiliares sirven para la lectura de los ficheros de datos con las cadenas a utilizar en el benchmark, y que están almacenadas en la carpeta src/data.

Este benchmark tiene un volumen considerable de código para la inicialización. Este código no se incluye en las mediciones de tiempo hasta que aparecen las transferencias de memoria y comienza el cómputo del alineamiento de las cadenas.

Dado que las cadenas se suministran en ficheros que acompañan a la suite, el resultado es conocido y se puede comprobar si la ejecución es errónea.

La versión secuencial y OpenACC de la función de búsqueda de las cadenas se puede ver en la Figura 6.14. La versión CUDA está en la Figura 6.15.

```
1  __global__ void kernel_busqueda(const char* b_idata, const int sizeB,
2                  const char* s_idata, const int sizeS, unsigned int* result)
3  {
4      extern __shared__ unsigned char sdata[];
5      int gid = (blockIdx.x*blockDim.x) + threadIdx.x;
6      int tid = threadIdx.x;
7      int sStart = TS_PER_BLOCK + sizeS;
8      int offset;
9      if (gid == 0){
10         *result = -1;
11     }
12     __syncthreads();
13
14     if (gid < *result){
15         unsigned char* bd = &sdata[tid];
16         b_idata += gid;
17         unsigned char* sd = &sdata[sStart+tid];
18         s_idata += tid;
19         if(tid<TS_PER_BLOCK){
20             for(offset = 0; tid+offset<sStart; offset += TS_PER_BLOCK){
21                 if (offset==0){
22                     *bd = *b_idata;
23                     bd+=TS_PER_BLOCK; b_idata+=TS_PER_BLOCK;
24                     continue;
25                 }
26                 *bd = *b_idata;
27                 bd+=TS_PER_BLOCK; b_idata+=TS_PER_BLOCK;
28                 *sd = *s_idata;
29                 sd+=TS_PER_BLOCK; s_idata+=TS_PER_BLOCK;
30             }
31         }
32         __syncthreads();
33         if (gid <= sizeB-sizeS){
34             unsigned int i;
35             unsigned int b = 1;
36             unsigned char* sd = &sdata[sStart];
37             unsigned char* bd = &sdata[tid];
38             for(i = 0; b && i+15 < sizeS; i+=16){
39                 b &= (*sd == *bd); sd++; bd++;
40                 <...Loop unrolling skipped...>
41                 if (!b) break;
42             }
43             for(; b && i < sizeS ; i++){
44                 b &= (*sd == *bd); sd++; bd++;
45                 if (!b) break;
46             }
47             if ( b ) atomicMin(result, gid);
48         }
49     }
50 }
```

Figura 6.15: Código del kernel CUDA del benchmark StringMatch

```
1   #pragma acc data copyin(cube[0:s*s*s]) copyout(cube2[0:s*s*s])
2   {
3   // Memory transfers affect our metric. Adding computation solves the problem
4   int it;
5   for(it = 0; it < 10; it++){
6       int left, right, up, down, front, back;
7       #pragma acc parallel loop
8       for(z = 0; z < s; ++z){
9       #pragma acc loop
10      for(y = 0; y < s; ++y){
11      #pragma acc loop
12      for(x = 0; x < s; ++x){
13          int idx = z*(s*s) + y*s + x;
14          left  = x == 0     ? -1 : idx - 1;
15          right = x == s-1 ? -1 : idx + 1;
16          up    = y == 0     ? -1 : idx - s;
17          down  = y == s-1 ? -1 : idx + s;
18          front = z == 0     ? -1 : idx - s*s;
19          back  = z == s-1 ? -1 : idx + s*s;
20
21          int i;
22          int valid_sides = 0;
23          int counter = 0;
24          if (left >= 0 && left < s*s*s){
25              ++valid_sides;
26              counter += cube[left];
27          }
28          <...more neighbours skipped...>
29          cube2[idx] = counter / valid_sides ;
30      } } }
31  }
32  } // acc data region
```

Figura 6.16: Fragmento de código secuencial y OpenACC para 3DStencil

6.3.3. T_3DStencil: Stencil de 6 puntos tridimensional

El Stencil 3D que se ha diseñado para *TORMENT OpenACC2016* es un cubo de unos y ceros alternados sobre el que se calcula la media de los valores situados en los seis puntos adyacentes. Esto permite verificar muy fácilmente los resultados correctos en la ejecución de los benchmarks. Este benchmark esta implementado en el fichero benchmarks/3dstencil.c.

Para reducir el peso de las transferencias de memoria se establecen una serie de iteraciones en las que se aplicará el stencil diseñado.

Los fragmentos de código OpenACC y Secuencial y CUDA se pueden ver en las Figuras 6.16 y 6.17 respectivamente.

```
1  /*
2   *  CUDA kernel for the calculation of the new values of the cube
3   */
4  __global__ void kernel_stencil (unsigned int* cube, unsigned int* copy)
5  {
6      int gid = (blockIdx.x*blockDim.x) + threadIdx.x;
7      int z = gid / (SIZE*SIZE);
8      int y = (gid % (SIZE*SIZE)) / SIZE;
9      int x = (gid % (SIZE*SIZE)) % SIZE;
10
11     if (gid < SIZE*SIZE*SIZE) {
12         int sides[6];
13         sides[0] = x == 0      ? -1 : gid - 1;
14         sides[1] = x == SIZE-1 ? -1 : gid + 1;
15         sides[2] = y == 0      ? -1 : gid - SIZE;
16         sides[3] = y == SIZE-1 ? -1 : gid + SIZE;
17         sides[4] = z == 0      ? -1 : gid - SIZE*SIZE;
18         sides[5] = z == SIZE-1 ? -1 : gid + SIZE*SIZE;
19
20         int i;
21         int valid_sides = 0;
22         int counter = 0;
23         for (i = 0; i < 6; ++i){
24             if (sides[i] >= 0 && sides[i] < SIZE*SIZE*SIZE){
25                 ++valid_sides;
26                 counter += cube[sides[i]];
27             }
28         }
29         copy[gid] = counter / valid_sides ;
30     }
31 }
```

Figura 6.17: Código del kernel CUDA del benchmark 3DStencil

```
1   #pragma acc data copyout(grid[0:x*y])
2   {
3
4   int i,j;
5   #pragma acc parallel loop
6   for (i = 0; i < y; ++i){
7   #pragma acc loop
8   for (j = 0; j < x; ++j){
9       double x0, y0, x1, y1;
10      int iter = 0;
11      int max_iter = 10000;
12      x0 = (j*3.5)/x - 2.5;
13      y0 = (i*2.0)/y - 1.0;
14      x1 = 0.0;
15      y1 = 0.0;
16      while (x1*x1 + y1*y1 < 4 && iter < max_iter) {
17          double tmp = x1*x1 - y1*y1 + x0;
18          y1 = 2*x1*y1 + y0;
19          x1 = tmp;
20          iter++;
21      }
22      if (iter >= max_iter){
23          grid[i*x + j] = '#';
24      } else {
25          grid[i*x + j] = ' ';
26      }
27  }
28  }
29
30  } // acc data region
```

Figura 6.18: Fragmento de código secuencial y OpenACC para Mandelbrot

6.3.4. T_Mandelbrot: Generador del conjunto de Mandelbrot

En el fichero benchmarks/mandelbrot.c se implementa el benchmark del conjunto de mandelbrot utilizando el algoritmo de tiempo de escape. El algoritmo es muy sencillo y no existen dependencias entre cálculos, por lo que debería verse una buena escalabilidad.

El código OpenACC y secuencial se puede ver en la Figura 6.18, así como el código CUDA en la Figura 6.19.

```
1  /*
2   *  CUDA Kernel to calculate the mandelbrot set
3   */
4   __global__ void kernel_mandelbrot(char* grid, int x, int y)
5  {
6      int gid = (blockIdx.x*blockDim.x) + threadIdx.x;
7      int i = gid / x;
8      int j = gid % x;
9      double x0, y0, x1, y1;
10     int iter = 0;
11     int max_iter = 10000;
12     x0 = (j*3.5)/x - 2.5;
13     y0 = (i*2.0)/y - 1.0;
14     x1 = 0.0;
15     y1 = 0.0;
16     while (x1*x1 + y1*y1 < 4 && iter < max_iter) {
17         double tmp = x1*x1 - y1*y1 + x0;
18         y1 = 2*x1*y1 + y0;
19         x1 = tmp;
20          iter++;
21     }
22     if ( iter >= max_iter){
23         grid[gid] = '#';
24     } else {
25         grid[gid] = ' ';
26     }
27 }
```

Figura 6.19: Código del kernel CUDA del benchmark Mandelbrot

```
1   #pragma acc data copyin(a[0:size*size], b[0: size*size]) \
2                       copyout(c[0: size*size])
3   {
4
5   #pragma acc parallel loop independent
6   for(i = 0; i < size; i++){
7   #pragma acc loop independent
8   for(j = 0; j < size; j++){
9       c[i*size+j] = 0.0f;
10      for (k = 0; k < size; k++){
11          c[i*size+j] += a[i*size+k] * b[k*size+j];
12      }
13  }
14  }
15
16  } //acc data region
```

Figura 6.20: Fragmento de código secuencial y OpenACC para MatrixMult

6.3.5. T_MatrixMult: Multiplicación de matrices

El fichero benchmarks/matrixMult.c contiene la implementación de la multiplicación de matrices de *TORMENT OpenACC2016*. Este benchmark necesita tres matrices sobre las que trabajar: los dos factores y la matriz de resultados. La comprobación de resultado se realiza mediante la comprobación de un checksum de la matriz de resultados. En caso de ser incorrecto se invalida el resultado y la ejecución de la suite.

El código OpenACC y secuencial de este benchmark se presenta en la Figura 6.20. El código del kernel CUDA puede verse en la Figura 6.21.

```
1  __global__ void kernel_mult(float* a, float* b, float* c, int size)
2  {
3      __shared__ float s_a[THREADS_PER_BLOCK];
4      __shared__ float s_b[THREADS_PER_BLOCK];
5      int tid = threadIdx.y*blockDim.x + threadIdx.x;
6      int gx = blockIdx.x*blockDim.x + threadIdx.x;
7      int gy = blockIdx.y*blockDim.y + threadIdx.y;
8      float acum = 0;
9
10     s_a[tid] = 0.0f;
11     s_b[tid] = 0.0f;
12
13     int bl;
14     for(bl = 0; bl < gridDim.x; ++bl){
15         s_a[tid] = 0.0f;
16         s_b[tid] = 0.0f;
17         int bx = bl * blockDim.x + threadIdx.x;
18         int by = bl * blockDim.y + threadIdx.y;
19         if(gy < size && bx < size){
20             s_a[tid] = a[gy*size + bx];
21         }
22         if(gx < size && by < size){
23             s_b[tid] = b[by*size + gx];
24         }
25
26         __syncthreads();
27
28         int i;
29         for(i = 0; i < blockDim.x; ++i){
30             acum += s_a[threadIdx.y*blockDim.x + i] *
31                     s_b[i*blockDim.x + threadIdx.x];
32         }
33
34         __syncthreads();
35     }
36
37     if(gy < size && gx < size){
38         c[gy*size+ gx] = acum;
39     }
40 }
```

Figura 6.21: Código del kernel CUDA del benchmark MatrixMult

```
1     #pragma acc data copyin(array[0:size]) copyout(reversed[0:size])
2     {
3
4     int it;
5     for(it = 0; it < 4096; it++){
6
7         #pragma acc parallel loop
8         for(i = 0; i < size; i++){
9             reversed[size-i-1] = array[i];
10        }
11
12    }
13
14    }
```

Figura 6.22: Fragmento de código secuencial y OpenACC para ReverseArray

```
1  __global__ void kernel_reverse(int* array, int* reversed, int size)
2  {
3      __shared__ int s_data[THREADS_PER_BLOCK];
4      int gid = blockIdx.x*blockDim.x + threadIdx.x;
5      int tid = threadIdx.x;
6      int block = blockIdx.x;
7      int block_rev = gridDim.x - block - 1;
8
9      int value = array[gid];
10     s_data[THREADS_PER_BLOCK-tid-1] = value;
11     __syncthreads();
12
13     reversed[block_rev*blockDim.x + tid] = s_data[tid];
14 }
```

Figura 6.23: Código del kernel CUDA del benchmark ReverseArray

6.3.6. T_ReverseArray: Inversión de un array de enteros

Este benchmark está implementado en el fichero benchmarks/reverseArray.c. Los datos iniciales del array se corresponden con el indice de cada posición, por lo que la comprobación del resultado es simple. Este benchmark presenta dos problemas de implementación. Como no prácticamente computación, aparte de los cálculos de indices y las lecturas y escrituras de memoria correspondientes, la ejecución es muy rápida. No obstante, aumentar el tamaño del array supone incrementar los tiempos dedicados a la transferencia de memoria entre GPU y Host, por lo que se ven afectados los ratios con respecto a la máquina de referencia. La mejor solución ha sido la de establecer un elevado número de iteraciones en las que se aplica la inversión del array inicial.

El código OpenACC y secuencial se puede ver en la Figura 6.22. El código del kernel CUDA puede verse en la Figura 6.23.

6.4. Plan de Pruebas

A continuación se detallan las pruebas realizadas para la comprobación de la funcionalidad de *TORMENT OpenACC2016*. Las pruebas se dividen en dos partes, por un lado las pruebas relativas al *wrapper* de ejecución y por otro las pruebas de los benchmarks.

Cabe destacar que, dado las limitaciones que impone el uso de compiladores en un nivel muy temprano de desarrollo, no es posible establecer un plan de pruebas más exhaustivo mediante el uso de tests unitarios y *mocking*.

Todas las pruebas se superaron correctamente, obteniendo el resultado esperado en cada caso.

6.4.1. Pruebas relativas al *wrapper* de ejecución

El *wrapper* de ejecución es la parte que se ocupa de la configuración, la ejecución de la suite y la generación de los resultados. La ejecución de la suite conlleva la compilación de los fuentes y la ejecución del fichero binario resultante, que procederá al lanzamiento de los benchmarks y la obtención del resultado.

A continuación, se detallan las pruebas preparadas para comprobar el funcionamiento correcto de las diferentes funcionalidades, así como para ver el comportamiento frente a situaciones incorrectas en alguna de ellas.

Detección y confirmación de rutas

Las siguientes pruebas verifican la funcionalidad recogida en el RF-01 sobre configuración y detección de compiladores de OpenACC.

Prueba 1.1a	
Descripción	Esta prueba está destinada a probar el funcionamiento de la detección de rutas
Entrada	Fichero disponible en una única ruta
Resultado Esperado	El script devuelve el resultado correcto para su confirmación

Prueba 1.1b	
Descripción	Esta prueba está destinada a probar el funcionamiento de la detección de rutas
Entrada	Fichero disponible en varias rutas
Resultado Esperado	El script devuelve los resultados correctos para su confirmación

Prueba 1.1c	
Descripción	Esta prueba está destinada a probar el funcionamiento de la detección de rutas
Entrada	Fichero no disponible en el sistema
Resultado Esperado	El script devuelve que el fichero no está en el sistema

Confirmación de resultados de rutas

Estas pruebas verifican parte de la funcionalidad recogida en el RF-01 sobre configuración y detección de compiladores de OpenACC.

Prueba 1.2a	
Descripción	Esta prueba está destinada a probar el funcionamiento de la confirmación de rutas por parte del usuario
Entrada	Confirmación positiva
Resultado Esperado	El script almacena correctamente la ruta confirmada

Prueba 1.2b	
Descripción	Esta prueba está destinada a probar el funcionamiento de la confirmación de rutas por parte del usuario
Entrada	Confirmación negativa
Resultado Esperado	El script pide confirmación de la siguiente ruta encontrada o, si no hubiera más, informa de que el fichero buscado no está disponible, almacenando correctamente la información

Configuración del comando de ejecución

Estas pruebas están relacionadas con el RF-03 sobre ejecución de la Suite.

Prueba 1.3a	
Descripción	Esta prueba asegura que la configuración del comando de ejecución de la suite es correcta
Entrada	Cadena vacía
Resultado Esperado	El script interpreta la entrada como ausencia de un comando de ejecución no estándar y almacena la configuración.

Prueba 1.3b	
Descripción	Esta prueba asegura que la configuración del comando de ejecución de la suite es correcta
Entrada	Comando de ejecución
Resultado Esperado	El script almacena en la configuración el comando de ejecución insertado por el usuario.

Detección de fichero de configuracion

Estas pruebas están relacionadas con el RF-01 sobre configuración y detección de compiladores de OpenACC.

Prueba 1.4a	
Descripción	Esta prueba asegura que se gestiona correctamente la existencia del fichero de configuración de la suite
Entrada	fichero .config no existe
Resultado Esperado	El script lanza automáticamente la configuración de la suite.

Prueba 1.4b	
Descripción	Esta prueba asegura que se gestiona correctamente la existencia del fichero de configuración de la suite
Entrada	fichero .config existe
Resultado Esperado	El script carga la información del fichero de configuración y continua con la ejecución de la suite

Compilación de los fuentes

Las siguientes pruebas están relacionadas con el RF-02 sobre compilación de la suite.

Prueba 1.5a	
Descripción	Esta prueba asegura que se gestione correctamente la compilación de los fuentes por parte de los diferentes compiladores
Entrada	Fuentes correctos
Resultado Esperado	El script completa correctamente la llamada al comando make, que genera los binarios correspondientes para su ejecución

Prueba 1.5b	
Descripción	Esta prueba asegura que se gestione correctamente la compilación de los fuentes por parte de los diferentes compiladores
Entrada	Fuentes con errores de compilación
Resultado Esperado	El script devuelve un error y almacena la información de compilación en ficheros temporales para su posterior revisión en la carpeta temp

Detección de resultados incorrectos

Estas pruebas están relacionadas con el RF-04 sobre obtención de ratios con respecto a tiempos de referencia

Prueba 1.6a	
Descripción	Esta prueba asegura que la obtención de datos incorrectos invalida la ejecución para el compilador implicado
Entrada	Resultado -1 en cualquier benchmark
Resultado Esperado	El programa aborta la ejecución en curso, omitiendo el volcado de datos en el fichero de resultados

Prueba 1.6b	
Descripción	Esta prueba asegura que la obtención de datos incorrectos invalida la ejecución para el compilador implicado
Entrada	Resultado positivo en todos los benchmarks
Resultado Esperado	El programa realiza los cálculos necesarios para obtener los datos de ejecución y los vuelca en el fichero de resultados.

Generación de resúmenes de resultados

Las pruebas detalladas a continuación se relacionan con el RF-05 sobre generación de resúmenes de resultados.

Prueba 1.7a	
Descripción	Esta prueba asegura la obtención de resúmenes de datos acordes a los resultados obtenidos en la ejecución de la suite
Entrada	Ejecución correcta en todos los compiladores estudiados
Resultado Esperado	El script genera las tablas de todos los compiladores, así como los datos de hardware y software recuperados del sistema.

Prueba 1.7b	
Descripción	Esta prueba asegura la obtención de resúmenes de datos acordes a los resultados obtenidos en la ejecución de la suite
Entrada	Ejecución incorrecta en uno o más de los compiladores
Resultado Esperado	El script genera las tablas para los compiladores cuyo código ha devuelto resultado correcto, dejando fuera del resumen aquellos que hayan generado resultados incorrectos

6.4.2. Pruebas de los benchmarks

En esta sección se detallan las pruebas necesarias para la comprobación de resultados en la ejecución de los benchmarks. Es necesario que durante la ejecución el resultado sea el correcto, de lo contrario las mediciones no serían correctas.

T_MonteCarloPi

El benchmark MonteCarloPi se basa en la generación de números aleatorios. Dado que el resultado final es en si mismo una aproximación, la condición de aceptación del resultado es que el valor de pi sea correcto con un margen de error del 1 %.

Prueba 2.1a	
Descripción	Comprobación del resultado en T_MonteCarloPi
Entrada	Resultado con error dentro del margen del 1 %
Resultado Esperado	El benchmark devuelve el tiempo requerido para su finalización

Prueba 2.1b	
Descripción	Comprobación del resultado en T_MonteCarloPi
Entrada	Resultado con error fuera del margen del 1 %
Resultado Esperado	El benchmark devuelve -1.0

T_StringMatch

Este benchmark utiliza cadenas de caracteres que se suministran junto a la suite. Los casos prueba son conocidos de antemano.

Prueba 2.2a	
Descripción	Comprobación del resultado en T_StringMatch
Entrada	Todos los resultados correctos
Resultado Esperado	El benchmark devuelve el tiempo requerido para su finalización

Prueba 2.2b	
Descripción	Comprobación del resultado en T_StringMatch
Entrada	Al menos un resultado incorrecto
Resultado Esperado	El benchmark devuelve -1.0

T_3DStencil

En este benchmark se intercambian unos por ceros y viceversa, por lo que la comprobación es sencilla.

Prueba 2.3a	
Descripción	Comprobación del resultado en T_3DStencil
Entrada	Resultado correcto
Resultado Esperado	El benchmark devuelve el tiempo requerido para su finalización

Prueba 2.3b	
Descripción	Comprobación del resultado en T_StringMatch
Entrada	La aplicación del Stencil da lugar a un resultado erróneo
Resultado Esperado	El benchmark devuelve -1.0

T_Mandelbrot

La generación del conjunto de Mandelbrot se puede realizar en secuencial y posteriormente comprobarse con la ejecución en paralelo.

Prueba 2.4a	
Descripción	Comprobación del resultado en T_Mandelbrot
Entrada	Todos los resultados correctos
Resultado Esperado	El benchmark devuelve el tiempo requerido para su finalización

Prueba 2.4b	
Descripción	Comprobación del resultado en T_Mandelbrot
Entrada	El algoritmo devuelve un resultado distinto al obtenido en secuencial
Resultado Esperado	El benchmark devuelve -1.0

T_MatrixMult

La multplicación de matrices utiliza valores conocidos de antemano, por lo que es fácil obtener un *checksum* con el que comprobar el resultado.

Prueba 2.5a	
Descripción	Comprobación del resultado en T_MatrixMult
Entrada	Checksum correcto
Resultado Esperado	El benchmark devuelve el tiempo requerido para su finalización

Prueba 2.5b	
Descripción	Comprobación del resultado en T_MatrixMult
Entrada	Checksum incorrecto
Resultado Esperado	El benchmark devuelve -1.0

T_ReverseArray

Este benchmark consiste en la inversión de un array de enteros, la comprobación del resultado es por tanto trivial.

Prueba 2.6a	
Descripción	Comprobación del resultado en T_ReverseArray
Entrada	Array correctamente invertido
Resultado Esperado	El benchmark devuelve el tiempo requerido para su finalización

Prueba 2.6b	
Descripción	Comprobación del resultado en T_ReverseArray
Entrada	Al menos un resultado incorrecto
Resultado Esperado	El benchmark devuelve -1.0

Mostrados los detalles más relevantes de la implementación de *TORMENT OpenACC2016*, a continuación se procederá a analizar los resultados obtenidos que la versión funcional de la herramienta. El próximo capítulo contendrá varios ejemplos de ejecución, así como resúmenes de resultados obtenidos por los benchmarks.

Capítulo 7

Resultados

7.1. Máquina de Referencia

La máquina de referencia permite obtener los tiempos de referencia que serán utilizados por *TORMENT OpenACC2016* para el cálculo de los ratios y, posteriormente, la media armónica que dará como resultado la métrica *TORMENT_ACC2016* presentada anteriormente en esta memoria. El uso de una máquina de referencia permite normalizar la métrica de rendimiento propuesta.

Para elegir una máquina de referencia debe tenerse en cuenta que, con el objetivo de que los ratios sean significativos, es una buena idea que las características técnicas de la máquina no sean demasiado modernas. Como ejemplo, SPEC utiliza una máquina de 1997 [8], la *Ultra Enterprise 2*, de Sun. Esta máquina posee un procesador *UltraSPARC II* de 296 Mhz. Al utilizar esta máquina, los ratios que se obtienen en los benchmarks de SPEC son progresivamente mayores conforme pasan los años debido a la cada vez mayor diferencia de tiempos entre la máquina de referencia y los resultados de las máquinas modernas.

TORMENT OpenACC2016 sigue un principio similar, pero no tan extremo. La máquina de referencia elegida ha sido un portátil ASUS modelo R510J cuyas características más relevantes son las que se indican en la tabla 7.1. Esta máquina posee un procesador con cuatro cores bastante potentes y, por ejemplo, en el caso del cluster de máquinas del Grupo Trasgo de la Universidad de Valladolid, las máquinas utilizadas en la experimentación con *TORMENT OpenACC2016* dan, en secuencial, peores resultados. Esto no es relevante en ningún caso para el estudio, pues la métrica *TORMENT_ACC2016* únicamente se utiliza con las ejecuciones de compiladores de OpenACC.

Para el cálculo de los tiempos de referencia se utiliza únicamente la versión secuencial de la suite, concretamente los tiempos mínimo y medio de 10 repeticiones.

CPU	Intel(R) Core(TM) i5-4200H CPU @ 2.80GHz
Arquitectura	x86_64
Cores	4
Frecuencia Máxima	3400.0000 MHz
Cache L1	32 KB
Cache L2	256 KB
Cache L3	3072 KB
Memoria RAM	4 GB
Kernel Linux	Linux 3.16.0-4-amd64 x86_64
Versión GCC	gcc(Debian 4.9.2-10) 4.9.2

Tabla 7.1: Características de la máquina de referencia

7.2. Ejemplos de Experimentación

Han sido elegidas dos máquinas para la utilización de *TORMENT OpenACC2016* para la evaluación de rendimiento del código generado por compiladores de OpenACC. Una de ellas es precisamente la máquina de referencia. A primera vista puede sorprender que se utilice la máquina de referencia como máquina de estudio, pero esto no es un problema ya que los tiempos de referencia son únicamente tiempos de la ejecución secuencial, mientras que la métrica *TORMENT_ACC2016* utiliza los tiempos de la ejecución en la GPU del código OpenACC generado por los compiladores a estudiar, normalizados con los tiempos de referencia. La otra máquina utilizada es la máquina *Hydra* del cluster de máquinas del Grupo Trasgo de la Universidad de Valladolid.

La información relativa a Hardware y Software relevante de estas máquinas puede encontrarse en los resúmenes de resultados obtenidos para ambas y que están en los Anexos II y III, para el portátil ASUS y para Hydra respectivamente.

7.3. Análisis de Resultados

A continuación se hace un análisis de los resultados obtenidos en los ejemplos de experimentación descritos en el apartado anterior.

Lo primero que se puede ver en los resúmenes de resultados es la información del sistema. Esta información está organizada en:

- Información de la pila de software: Contiene las versiones del kernel linux utilizado, así como de los compiladores que se utilizan en la prueba.
- Información de CPU(s): Contiene información relativa a la CPU como su modelo, arquitectura, frecuencias, etc. Además, incluye el tamaño de la memoria RAM y de las cachés del procesador.
- Información de GPU(s): Contiene información sobre el modelo de las GPU(s) disponibles en el sistema.

Tras la información del sistema encontramos las tablas de resultados de las ejecuciones de código secuencial, compilado con GCC, y las de código CUDA. Estos resultados

son meramente informativos y no se usan en la métrica *TORMENT_ACC2016* ya que no es código que haya sido generado por un compilador OpenACC. No obstante, estos resultados permiten obtener una comparación relativa a la propia máquina, eliminando el factor sistema de los factores de variabilidad. En estas tablas tenemos los resultados para cada Benchmark con sus tiempos mínimo y medio, los ratios con respecto a los tiempos de referencia y la desviación estándar.

Después de los resultados para las ejecuciones secuencial y de código CUDA, se presentan las tablas de resultados para los compiladores de OpenACC disponibles en el sistema del usuario. En esta tabla se tiene la misma información que se ha descrito anteriormente. Además, bajo la tabla se pueden ver los valores que toma la métrica *TORMENT_ACC2016* en sus versiones mínima y media. Es interesante recordar que esta métrica es la media armónica de los ratios obtenidos al dividir el tiempo de referencia mínimo y máximo entre los tiempos resultantes de la ejecución del código generado por el compilador de OpenACC. Además, se incluye la información relativa al propio sistema acerca de los *speedups* con respecto a la versión secuencial y CUDA ejecutadas previamente.

En los dos informes de resultados suministrados en los Anexos II y III se pueden observar los resultados concretos a las ejecuciones en dos máquinas, descritas en el apartado anterior. En estos informes, se puede observar como en ambos casos el mejor resultado lo obtiene el compilador de PGI. Esto es coherente con lo afirmado en el capítulo 3 sobre el Estado del Arte. En las dos máquinas el compilador de PGI obtiene entorno al 60-70 % del rendimiento que se ha conseguido con las implementaciones de CUDA. En los resultados individuales se ve que para el benchmark T_Mandelbrot el compilador de PGI obtiene un mejor rendimiento que en CUDA. Esto indica dos cosas: por una parte, la implementación de CUDA es mejorable, pero también muestra que el compilador de PGI hace un buen trabajo generando el código correspondiente y realizando varias optimizaciones.

El compilador de la Universidad de Houston, OpenUH, queda en una segunda posición en los dos experimentos. El porcentaje de rendimiento frente a la implementación CUDA varía bastante entre las dos máquinas. Esto puede significar un problema a la hora de asignar los diferentes niveles de paralelismo o las geometrías de bloque. En la máquina Hydra alcanza el 41 % del rendimiento que consigue la implementación de CUDA, pero usando una GPU más humilde, este porcentaje aumenta hasta el 63 %.

Finalmente, el compilador accULL, de la Universidad de La Laguna, queda en última posición en ambos informes. De nuevo ocurre lo mismo que con el compilador OpenUH. La experimentación en Hydra muestra un porcentaje del 16 % del rendimiento que consigue la implementación CUDA, mientras que con la GPU de prestaciones inferiores consigue un 44 %.

Con este análisis de resultados se culmina el trabajo realizado en el contexto de este Trabajo de Fin de Grado. En el próximo y último capítulo se presentarán las conclusiones extraídas, así como una lista de puntos a tener en cuenta como trabajo futuro.

Capítulo 8

Conclusiones y Trabajo Futuro

8.1. Conclusiones

TORMENT OpenACC2016 es una herramienta de análisis y comparación de rendimientos de código generado por compiladores de OpenACC. Esta herramienta toma en consideración el nivel de madurez tanto del estándar OpenACC como de los diferentes compiladores. *TORMENT OpenACC2016* desarrolla una suite de benchmarks específicamente diseñados para OpenACC y manteniendo la máxima portabilidad entre compiladores, permitiendo su compilación y ejecución en todos ellos. A continuación se detallan las aportaciones que se presentan en este proyecto:

- Se ha analizado el estado del estándar OpenACC y de los diferentes compiladores que lo implementan, gracias a lo cual se han descubierto las fortalezas y debilidades de cada uno de ellos.
- Se han descubierto herramientas de análisis de rendimiento preparadas, o adaptadas, para su uso con OpenACC. Estas herramientas han dado ideas sobre las técnicas que funcionan y las que no.
- Se ha realizado experimentación con las herramientas existentes, dando una idea general del comportamiento de los diferentes compiladores y completando la información sobre su estado.
- Se ha ideado una nueva herramienta de análisis de rendimiento de código generado por compiladores de OpenACC que se ha bautizado como *TORMENT OpenACC2016* que trata de dar respuesta a las necesidades encontradas.
- Se ha desarrollado la métrica *TORMENT_ACC2016* para que la herramienta pueda dar un valor numérico representativo del nivel de rendimiento obtenido en las pruebas efectuadas.
- Se han enumerado las características fundamentales del trabajo en las GPUs que deben ser tenidos en cuenta en las pruebas de rendimiento y se ha diseñado e implementado un conjunto de seis benchmarks que ponen a prueba estas características.

- Se ha diseñado una hoja resumen de resultados que ofrece de forma clara y centralizada los datos obtenidos en las pruebas, ofreciendo al usuario además de la métrica *TORMENT_ACC2016*, una comparativa relativa a su propia máquina para las versiones secuencial, CUDA y OpenACC con los compiladores de los que disponga.

8.1.1. Publicaciones

Este Trabajo de Fin de Grado ha dado origen a las siguientes publicaciones:

- **Una herramienta de benchmarking para compiladores de OpenACC**: Esta publicación presenta una versión preliminar de *TORMENT OpenACC2016* en las XXVII Jornadas de Paralelismo, englobadas en las Jornadas Sarteco 2016. El artículo se puede ver en el Anexo IV.
- **Comparative Analysis of OpenACC Compilers**: Esta publicación realiza un análisis comparativo de los compiladores de OpenACC con las herramientas disponibles durante la fase de investigación de este Trabajo. Ha sido enviada al congreso internacional ICA3PP 2016 (Core B).
- **TORMENT OpenACC2016: A benchmarking tool for OpenACC compilers**: Esta publicación describe la versión final de la suite *TORMENT OpenACC2016* en el estado en que se presenta en esta memoria. Ha sido enviada al congreso internacional HiPC 2016 (Core A).

8.2. Trabajo Futuro

Este Trabajo de Fin de Grado abre las puertas a un gran número de posibilidades de trabajo futuro. OpenACC es un estándar de reciente desarrollo y que evoluciona con rapidez. Los compiladores que implementan el estándar están aún en una fase muy temprana de su desarrollo y conforme pasen los meses irán apareciendo nuevas versiones con un mayor número de funcionalidades.

Las circunstancias que rodean a *TORMENT OpenACC2016* no son inmutables y la suite debe evolucionar con ellas. La versión de la herramienta que se presenta con este Trabajo de Fin de Grado está pensada para el momento actual. Existe un gran número de funcionalidades que no han sido probadas porque en el momento actual no están implementadas completamente. Cuando esto cambie, *TORMENT OpenACC2016* deberá ser actualizado para continuar siendo completo.

Además, cualquier cambio en las implementaciones del estándar OpenACC puede tener consecuencias en la portabilidad que *TORMENT OpenACC2016* intenta conseguir, haciendo que alguno de los benchmarks deje de funcionar. La labor de mantenimiento es igualmente importante.

Por otra parte, el estándar OpenACC no está sólo diseñado para C, también existe para FORTRAN. Actualmente no existe un número suficiente de compiladores que permitan compilar código OpenACC en FORTRAN. Si en un futuro esta circunstancia cambiase, sería muy interesante portar los benchmarks a FORTRAN y analizar posibles diferencias de rendimiento.

Referencias

[1] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The nas parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[2] David H Bailey and John T Barton. The nas kernel benchmark program. 1985.

[3] Mike Bayer. Mako Templates. `http://www.makotemplates.org/`, nov 2015.

[4] David M. Beazley. PLY (Python Lex-Yacc). `http://www.dabeaz.com/ply`, nov 2015.

[5] Mike Berry, D Chen, P Koss, D Kuck, S Lo, Yingxin Pang, Lynn Pointer, R Roloff, A Sameh, E Clementi, et al. The perfect club benchmarks: Effective performance evaluation of supercomputers. *International Journal of High Performance Computing Applications*, 3(3):5–40, 1989.

[6] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. (IISWC), 2009 IEEE International Symposium on*, pages 44–54. IEEE, 2009.

[7] Shuai Che, Jeremy W Sheaffer, Michael Boyer, Lukasz G Szafaryn, Liang Wang, and Kevin Skadron. A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, pages 1–11. IEEE, 2010.

[8] Kaivalya M Dixit. The spec benchmarks. *Parallel computing*, 17(10):1195–1209, 1991.

[9] Jack J Dongarra, Piotr Luszczek, and Antoine Petitet. The linpack benchmark: past, present and future. *Concurrency and Computation: practice and experience*, 15(9):803–820, 2003.

[10] EPCC. Epcc OpenACC benchmark suite. `https://github.com/EPCCed/epcc-openacc-benchmarks`, sep 2013.

[11] L. Grillo, F. de Sande, and R. Reyes. Performance evaluation of OpenACC compilers. In *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, pages 656–663, Feb 2014.

[12] John Hammersley. *Monte carlo methods*. Springer Science & Business Media, 2013.

[13] Philippe Kruchten. *The rational unified process: an introduction*. Addison-Wesley Professional, 2004.

[14] David J Lilja. *Measuring computer performance: a practitioner's guide*. Cambridge University Press, 2005.

[15] Benoît Mandelbrot. *Fractals and chaos: the Mandelbrot set and beyond*. Springer Science & Business Media, 2013.

[16] John R Mashey. War of the benchmark means: time for a truce. *ACM SIGARCH Computer Architecture News*, 32(4):1–14, 2004.

[17] Frank H McMahon. The livermore fortran kernels: A computer test of the numerical performance range. Technical report, Lawrence Livermore National Lab., CA (USA), 1986.

[18] University of Houston. Open-source UH compiler. `http://web.cs.uh.edu/~openuh/download/`, nov 2015.

[19] OpenACC-standard.org. About OpenACC.

[20] OpenACC-standard.org. OpenACC 2.5 draft for public comment, oct 2015.

[21] OpenACC-Standard.org. The OpenACC application programming interface version 2.5, oct 2015.

[22] Pathscale. Rodinia benchmark suite 2.1 with OpenACC port. `https://github.com/pathscale/rodinia`, apr 2014.

[23] PGI. Pgi accelerator compilers with OpenACC directives. `https://www.pgroup.com/resources/accel.htm`, nov 2015.

[24] Louis-Noël Pouchet. Polybench: The polyhedral benchmark suite. *URL: http://www. cs. ucla. edu/˜ pouchet/software/polybench/[cited July,]*, 2012.

[25] Ruymán Reyes, Iván López-Rodríguez, Juan J Fumero, and Francisco de Sande. accULL: an OpenACC implementation with CUDA and OpenCL support. In *Euro-Par 2012 Parallel Processing*, pages 871–882. Springer, 2012.

[26] Cloyce D Spradling. Spec cpu2006 benchmark tools. *ACM SIGARCH Computer Architecture News*, 35(1):130–134, 2007.

[27] Xiaonan Tian, Rengan Xu, Yonghong Yan, Zhifeng Yun, Sunita Chandrasekaran, and Barbara Chapman. Compiling a high-level directive-based programming model for GPGPUs. In *Languages and Compilers for Parallel Computing*, pages 105–120. Springer, 2014.

[28] Cheng Wang, Rengan Xu, S. Chandrasekaran, B. Chapman, and O. Hernandez. A validation testsuite for OpenACC 1.0. In *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 1407–1416, May 2014.

Anexos

Anexo I

Ejemplo de Resumen de Resultados: SPEC

SPEC es un referente a la hora de realizar análisis de rendimiento. Ha desarrollado una serie de herramientas de benchmarking y es utilizado por un gran número de personas. Una de sus señas de identidad es la inclusión de una métrica propia que permite reducir a un único número los resultados de rendimiento obtenidos y, al utilizar un sistema de referencia, permite la comparación de rendimientos entre máquinas independientes mediante el uso de estas métricas.

Este anexo contiene un ejemplo de resumen de resultados obtenido de la página web de SPEC. La metodología SPEC ha sido una de las principales fuentes de referencia a la hora de desarrollar *TORMENT OpenACC2016* y, en concreto, este documento es una referencia esencial a la hora de diseñar el resumen de resultados que ofrece, tal y como se ha comentado en varios puntos del Capítulo 1.

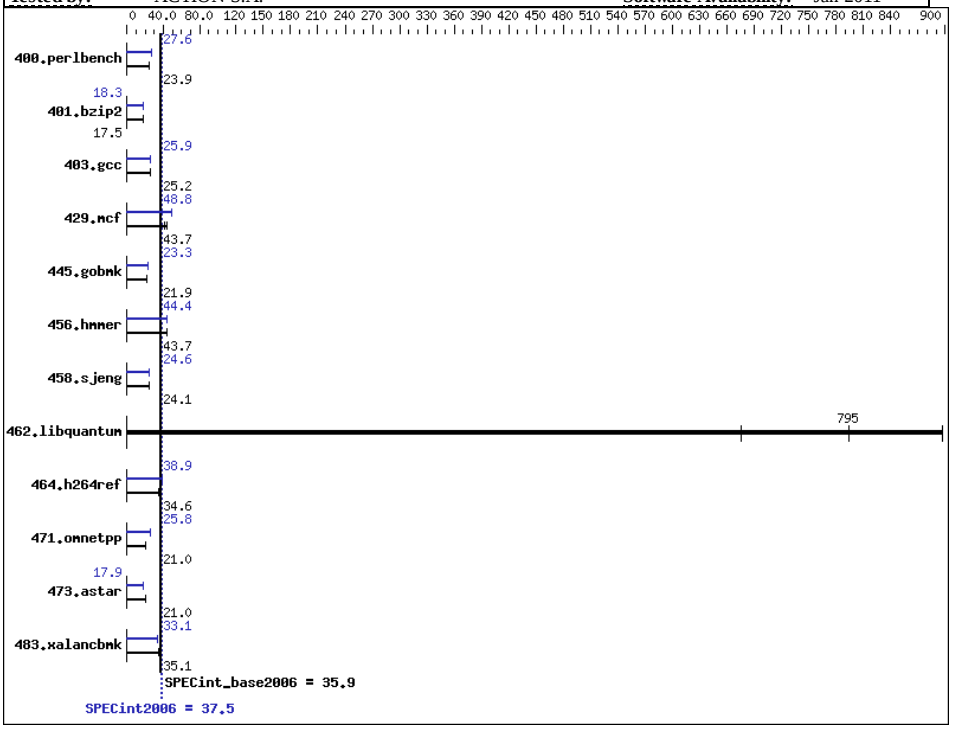
SPEC® CINT2006 Result

ACTION S.A.
ACTINA SOLAR 220 X3 (Intel Xeon X5650)

SPECint®2006 = 37.5
SPECint_base2006 = 35.9

CPU2006 license: 9008
Test sponsor: ACTION S.A.
Tested by: ACTION S.A.

Test date: Jun-2011
Hardware Availability: Apr-2010
Software Availability: Jan-2011



Hardware

CPU Name:	Intel Xeon X5650
CPU Characteristics:	Intel Turbo Boost Technology up to 3.06 GHz
CPU MHz:	2667
FPU:	Integrated
CPU(s) enabled:	12 cores, 2 chips, 6 cores/chip, 2 threads/core
CPU(s) orderable:	1,2 chips
Primary Cache:	32 KB I + 32 KB D on chip per core
Secondary Cache:	256 KB I+D on chip per core
L3 Cache:	12 MB I+D on chip per chip
Other Cache:	None
Memory:	48 GB (6 x 8 GB 2Rx4 PC3-10600R-9, ECC)
Disk Subsystem:	1 x 500 GB SATA, 7200 RPM

Software

Operating System:	SuSe Linux Enterprise Server 11 (x86_64), Kernel 2.6.27.19-5-default
Compiler:	Intel C++ Compiler XE for applications running on IA-32 Version 12.0.1.116 Build 20101116
Auto Parallel:	Yes
File System:	ext3
System State:	Run level 3 (multi-user)
Base Pointers:	32/64-bit
Peak Pointers:	32/64-bit
Other Software:	Microquill SmartHeap V9.01

Other Hardware:	None

Results Table

Benchmark	Base						Peak					
	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio	Seconds	Ratio
400.perlbench	410	23.8	408	24.0	**408**	**23.9**	353	27.7	**354**	**27.6**	354	27.6
401.bzip2	552	17.5	**552**	**17.5**	552	17.5	**527**	**18.3**	527	18.3	527	18.3
403.gcc	320	25.2	321	25.1	**320**	**25.2**	311	25.9	311	25.9	**311**	**25.9**
429.mcf	**208**	**43.7**	217	42.0	208	43.8	**187**	**48.8**	187	48.8	187	48.7
445.gobmk	**478**	**21.9**	473	22.2	486	21.6	**450**	**23.3**	450	23.3	450	23.3
456.hmmer	213	43.7	**214**	**43.7**	214	43.7	210	44.4	**210**	**44.4**	210	44.4
458.sjeng	501	24.1	502	24.1	**502**	**24.1**	492	24.6	493	24.6	**492**	**24.6**
462.libquantum	**26.1**	**795**	30.7	675	23.1	898	**26.1**	**795**	30.7	675	23.1	898
464.h264ref	**639**	**34.6**	640	34.6	639	34.6	569	38.9	569	38.9	**569**	**38.9**
471.omnetpp	**297**	**21.0**	297	21.0	298	21.0	242	25.8	242	25.9	**242**	**25.8**
473.astar	**334**	**21.0**	334	21.0	336	20.9	393	17.9	392	17.9	**393**	**17.9**
483.xalancbmk	199	34.7	196	35.3	**197**	**35.1**	**208**	**33.1**	208	33.1	209	33.1

Results appear in the order in which they were run. Bold underlined text indicates a median measurement.

Operating System Notes

```
'nodev   /mnt/hugepages   hugetlbfs   defaults   0 0' added to /etc/fstab
'ulimit -s unlimited' was used to set the stacksize to unlimited prior to run
echo 10800 > /proc/sys/vm/nr_hugepages
export HUGETLB_MORECORE=yes
export LD_PRELOAD=/usr/lib64/libhugetlbfs.so
```

General Notes

```
Binaries compiled on RHEL5.5 with
binutils-2.17.50.0.6-14.el5
OMP_NUM_THREADS set to number of cores
```

Base Compiler Invocation

C benchmarks:

icc -m64

C++ benchmarks:

icpc -m64

Base Portability Flags

400.perlbench: -DSPEC_CPU_LP64 -DSPEC_CPU_LINUX_X64
401.bzip2: -DSPEC_CPU_LP64
403.gcc: -DSPEC_CPU_LP64
429.mcf: -DSPEC_CPU_LP64
445.gobmk: -DSPEC_CPU_LP64
456.hmmer: -DSPEC_CPU_LP64
458.sjeng: -DSPEC_CPU_LP64
462.libquantum: -DSPEC_CPU_LP64 -DSPEC_CPU_LINUX
464.h264ref: -DSPEC_CPU_LP64
471.omnetpp: -DSPEC_CPU_LP64
473.astar: -DSPEC_CPU_LP64
483.xalancbmk: -DSPEC_CPU_LP64 -DSPEC_CPU_LINUX

Base Optimization Flags

C benchmarks:

-xSSE4.2 -ipo -O3 -no-prec-div -parallel -opt-prefetch -auto-p32
-B /usr/share/libhugetlbfs/ -Wl,-melf_x86_64 -Wl,-hugetlbfs-link=BDT

C++ benchmarks:

-xSSE4.2 -ipo -O3 -no-prec-div -opt-prefetch -Wl,-z,muldefs -L/smartheap -lsmartheap64 -B /usr/share/libhugetlbfs/ -Wl,-melf_x86_64 -Wl,-hugetlbfs-link=BDT

Base Other Flags

C benchmarks:

403.gcc: -Dalloca=_alloca

Peak Compiler Invocation

C benchmarks (except as noted below):

icc -m64

400.perlbench: icc -m32

429.mcf: icc -m32

445.gobmk: icc -m32

464.h264ref: icc -m32

C++ benchmarks (except as noted below):

icpc -m32

473.astar: icpc -m64

Peak Portability Flags

400.perlbench: -DSPEC_CPU_LINUX_IA32
401.bzip2: -DSPEC_CPU_LP64
403.gcc: -DSPEC_CPU_LP64
456.hmmer: -DSPEC_CPU_LP64
458.sjeng: -DSPEC_CPU_LP64
462.libquantum: -DSPEC_CPU_LP64 -DSPEC_CPU_LINUX
473.astar: -DSPEC_CPU_LP64
483.xalancbmk: -DSPEC_CPU_LINUX

Peak Optimization Flags

C benchmarks:

400.perlbench: -xSSE4.2(pass 2) -prof-gen(pass 1) -ipo(pass 2) -O3(pass 2) -no-prec-div(pass 2) -prof-use(pass 2) -opt-prefetch -ansi-alias -B /usr/share/libhugetlbfs/ -Wl,-hugetlbfs-link=BDT

401.bzip2: -xSSE4.2(pass 2) -prof-gen(pass 1) -ipo(pass 2) -O3(pass 2) -no-prec-div -prof-use(pass 2) -auto-ilp32 -opt-prefetch -ansi-alias

403.gcc: -xSSE4.2 -ipo -O3 -no-prec-div -inline-calloc -opt-malloc-options=3 -auto-ilp32 -B /usr/share/libhugetlbfs/ -Wl,-melf_x86_64 -Wl,-hugetlbfs-link=BDT

429.mcf: -xSSE4.2(pass 2) -prof-gen(pass 1) -ipo(pass 2) -O3(pass 2) -no-prec-div(pass 2) -prof-use(pass 2) -auto-ilp32 -ansi-alias -B /usr/share/libhugetlbfs/ -Wl,-hugetlbfs-link=BDT

445.gobmk: -xSSE4.2(pass 2) -prof-gen(pass 1) -prof-use(pass 2) -auto-ilp32 -ansi-alias -B /usr/share/libhugetlbfs/ -Wl,-hugetlbfs-link=BDT

456.hmmer: -xSSE4.2 -ipo -O3 -no-prec-div -unroll2 -auto-ilp32 -ansi-alias -B /usr/share/libhugetlbfs/ -Wl,-melf_x86_64 -Wl,-hugetlbfs-link=BDT

458.sjeng: -xSSE4.2(pass 2) -prof-gen(pass 1) -ipo(pass 2) -O3(pass 2) -no-prec-div(pass 2) -prof-use(pass

2) -unroll4

462.libquantum: basepeak = yes

464.h264ref: -xSSE4.2(pass 2) -prof-gen(pass 1) -ipo(pass 2) -O3(pass 2) -no-prec-div(pass 2) -prof-use(pass 2) -unroll2 -ansi-alias -B /usr/share/libhugetlbfs/ -Wl,-hugetlbfs-link=BDT

C++ benchmarks:

471.omnetpp: -xSSE4.2(pass 2) -prof-gen(pass 1) -ipo(pass 2) -O3(pass 2) -no-prec-div(pass 2) -prof-use(pass 2) -opt-ra-region-strategy=block -ansi-alias -Wl,-z,muldefs -L/smartheap -lsmartheap -B /usr/share/libhugetlbfs/ -Wl,-hugetlbfs-link=BDT

473.astar: -xSSE4.2(pass 2) -prof-gen(pass 1) -ipo(pass 2) -O3(pass 2) -no-prec-div(pass 2) -prof-use(pass 2) -opt-ra-region-strategy=routine -Wl,-z,muldefs -L/smartheap -lsmartheap64

483.xalancbmk: -xSSE4.2 -ipo -O3 -no-prec-div -opt-prefetch -ansi-alias -Wl,-z,muldefs -L/smartheap -lsmartheap -B /usr/share/libhugetlbfs/ -Wl,-hugetlbfs-link=BDT

Peak Other Flags

C benchmarks:

403.gcc: -Dalloca=_alloca

The flags files that were used to format this result can be browsed at
http://www.spec.org/cpu2006/flags/ACTION-platform-linux64.html
http://www.spec.org/cpu2006/flags/Intel-ic12.0-linux64-revB.html

You can also download the XML flags sources by saving the following links:
http://www.spec.org/cpu2006/flags/ACTION-platform-linux64.xml
http://www.spec.org/cpu2006/flags/Intel-ic12.0-linux64-revB.xml

For questions about this result, please contact the tester.
For other inquiries, please contact webmaster@spec.org

Tested with SPEC CPU2006 v1.1.
Report generated on Wed Jul 23 21:48:32 2014 by SPEC CPU2006 HTML formatter v6932.
Originally published on 5 July 2011.

Anexo II

Resumen de Resultados *TORMENT OpenACC2016*: Máquina de Referencia

En este anexo se presenta el resumen de resultados generado tras la ejecución de *TORMENT OpenACC2016* en la máquina de referencia. Este resumen contiene los datos de benchmarking obtenidos en una GPU modesta como resultado de la ejecución de código generado por los tres compiladores de OpenACC soportados. La información relativa al análisis de los resultados se puede encontrar en el Capítulo 7.

TORMENT OpenACC2016 Result page

Grupo Trasgo - Universidad de Valladolid

System Information

Benchmark version: 1.00
Hostname: corikLaptop
Username: dani

Software stack info
Kernel: Linux 3.16.0-4-amd64 x86_64
GCC: gcc (Debian 4.9.2-10) 4.9.2
NVCC: Cuda compilation tools, release 7.5, V7.5.17
PGI Compiler: pgcc 15.7-0 64-bit target on x86-64 Linux -tp haswell
OpenUH Compiler: OpenUH 3.1.0 (based on Open64 Compiler Suite: Version 5.0)
accULL Compiler: Release 0.4alpha

CPU info
Model: Intel(R) Core(TM) i5-4200H CPU @ 2.80GHz
Architecture: x86_64
Number of Cores: 4
Max MHz: 3400.0000 MHz
Min MHz: 800.0000 MHz
L1 Cache: 32K
L2 Cache: 256K
L3 Cache: 3072K
RAM: 3939364 kB

GPU(s) info
GPU0: NVIDIA Corporation GM107M [GeForce GTX 850M] (rev a2)

GCC_Sequential Compiler Results:

GCC_Sequential					
Benchmark	**Peak Time (s)**	**Peak Ratio**	**Avg. Time (s)**	**Avg. Ratio**	**Std. Deviation**
MonteCarloPi	6.200410	1.02	6.200819	1.02	0.000204
StringMatch	11.294680	1.02	11.306193	1.02	0.016588
3DStencil	11.059049	1.01	11.089780	1.01	0.023238
Mandelbrot	2.231308	1.02	2.231413	1.02	0.000092
MatrixMult	16.823954	0.98	16.995129	0.99	0.113361
ReverseArray	4.930743	1.02	5.001434	1.03	0.049034

NVCC_CUDA Compiler Results:

NVCC_CUDA					
Benchmark	**Peak Time (s)**	**Peak Ratio**	**Avg. Time (s)**	**Avg. Ratio**	**Std. Deviation**
MonteCarloPi	0.057815	109.01	0.060548	104.16	0.003332
StringMatch	0.335617	34.17	0.335792	34.27	0.000078
3DStencil	1.161615	9.66	1.162313	9.67	0.000619
Mandelbrot	0.291438	7.77	0.292700	7.75	0.000834
MatrixMult	0.148669	110.91	0.148916	112.59	0.000141
ReverseArray	1.264766	3.97	1.265037	4.08	0.000125

PGI Compiler Results:

PGI					
Benchmark	**Peak Time (s)**	**Peak Ratio**	**Avg. Time (s)**	**Avg. Ratio**	**Std. Deviation**
MonteCarloPi	0.344547	18.29	0.379062	16.64	0.034461
StringMatch	1.893914	6.06	2.083713	5.52	0.189430
3DStencil	0.994550	11.28	1.099986	10.22	0.099708
Mandelbrot	0.278717	8.13	0.308121	7.36	0.028050
MatrixMult	0.428552	38.48	0.471653	35.55	0.042872
ReverseArray	1.270164	3.95	1.397407	3.69	0.127038

TORMENT_ACC2016 Peak Score: 8.45
Speedup vs Sequential (peak): 8.36x
Speedup vs CUDA (peak): 0.75x

TORMENT_ACC2016 Average Score: 7.76
Speedup vs Sequential (average): 7.65x
Speedup vs CUDA (average): 0.68x

OpenUH Compiler Results:

OpenUH					
Benchmark	**Peak Time (s)**	**Peak Ratio**	**Avg. Time (s)**	**Avg. Ratio**	**Std. Deviation**
MonteCarloPi	0.701839	8.98	0.701957	8.98	0.000060
StringMatch	2.195887	5.22	2.196697	5.24	0.000457
3DStencil	0.828224	13.55	0.833254	13.49	0.002065
Mandelbrot	0.352985	6.42	0.355509	6.38	0.001635
MatrixMult	0.861674	19.14	0.862472	19.44	0.000442
ReverseArray	1.310527	3.83	1.310622	3.94	0.000076

TORMENT_ACC2016 Peak Score: 7.09
Speedup vs Sequential (peak): 7.03x
Speedup vs CUDA (peak): 0.63x

TORMENT_ACC2016 Average Score: 7.16
Speedup vs Sequential (average): 7.06x
Speedup vs CUDA (average): 0.63x

accULL Compiler Results:

accULL					
Benchmark	**Peak Time (s)**	**Peak Ratio**	**Avg. Time (s)**	**Avg. Ratio**	**Std. Deviation**
MonteCarloPi	0.679098	9.28	0.679183	9.29	0.000052
StringMatch	2.298331	4.99	2.299382	5.00	0.000435
3DStencil	2.899589	3.87	2.907701	3.87	0.003876
Mandelbrot	0.485486	4.67	0.488898	4.64	0.002212
MatrixMult	2.070534	7.96	2.070807	8.10	0.000191
ReverseArray	1.504273	3.34	1.505290	3.43	0.000375

TORMENT_ACC2016 Peak Score: 4.98
Speedup vs Sequential (peak): 4.93x
Speedup vs CUDA (peak): 0.44x

TORMENT_ACC2016 Average Score: 5.01
Speedup vs Sequential (average): 4.95x
Speedup vs CUDA (average): 0.44x

Peak score is the harmonic mean of the ratios of the best of 10 results. Average score is the harmonic mean of the ratios of the arithmetic mean of 10 results. In both cases higher is better.

Anexo III

Resumen de Resultados *TORMENT OpenACC2016*: Hydra

En este anexo se presenta el resumen de resultados generado tras la ejecución de *TORMENT OpenACC2016* en la máquina Hydra del cluster del Grupo Trasgo. Este resumen contiene los datos de benchmarking obtenidos en una GPU de altas prestaciones como resultado de la ejecución de código generado por los tres compiladores de OpenACC soportados. La información relativa al análisis de los resultados se puede encontrar en el Capítulo 7.

TORMENT OpenACC2016 Result page

Grupo Trasgo - Universidad de Valladolid

System Information

Benchmark version: 1.00
Hostname: hydra
Username: daniel

Software stack info
Kernel: Linux 3.10.0-229.4.2.el7.x86_64 x86_64
GCC: gcc (GCC) 4.8.3 20140911 (Red Hat 4.8.3-9)
NVCC: Cuda compilation tools, release 7.5, V7.5.17
PGI Compiler: pgcc 15.7-0 64-bit target on x86-64 Linux -tp haswell
OpenUH Compiler: OpenUH 3.1.0 (based on Open64 Compiler Suite: Version 5.0)
accULL Compiler: Release 0.4alpha

CPU info
Model: Intel(R) Xeon(R) CPU E5-2609 v3 @ 1.90GHz
Architecture: x86_64
Number of Cores: 6
Max MHz: MHz
Min MHz: MHz
L1 Cache: 32K
L2 Cache: 256K
L3 Cache: 15360K
RAM: 65687144 kB

GPU(s) info
GPU0: NVIDIA Corporation GK110B [GeForce GTX Titan Black] (rev a1)
GPU1: NVIDIA Corporation GK110B [GeForce GTX Titan Black] (rev a1)
GPU2: NVIDIA Corporation GK110B [GeForce GTX Titan Black] (rev a1)
GPU3: NVIDIA Corporation GK110B [GeForce GTX Titan Black] (rev a1)

GCC_Sequential Compiler Results:

GCC_Sequential					
Benchmark	**Peak Time (s)**	**Peak Ratio**	**Avg. Time (s)**	**Avg. Ratio**	**Std. Deviation**
MonteCarloPi	14.760615	0.43	14.760659	0.43	0.000032
StringMatch	20.121974	0.57	20.137021	0.57	0.013539
3DStencil	19.474451	0.58	19.477503	0.58	0.002025
Mandelbrot	3.987377	0.57	3.987397	0.57	0.000015
MatrixMult	6.411346	2.57	6.500626	2.58	0.076947
ReverseArray	3.166268	1.59	3.228853	1.60	0.030246

NVCC_CUDA Compiler Results:

NVCC_CUDA					
Benchmark	**Peak Time (s)**	**Peak Ratio**	**Avg. Time (s)**	**Avg. Ratio**	**Std. Deviation**
MonteCarloPi	0.017013	370.44	0.018084	348.73	0.000757
StringMatch	0.105946	108.25	0.106084	108.48	0.000086
3DStencil	0.620855	18.08	0.646922	17.38	0.051729
Mandelbrot	0.054516	41.55	0.054896	41.33	0.000344
MatrixMult	0.067772	243.30	0.067976	246.65	0.000104
ReverseArray	0.193948	25.89	0.194152	26.59	0.000096

PGI Compiler Results:

PGI					
Benchmark	**Peak Time (s)**	**Peak Ratio**	**Avg. Time (s)**	**Avg. Ratio**	**Std. Deviation**
MonteCarloPi	0.067162	93.84	0.076087	82.88	0.007608
StringMatch	0.688423	16.66	0.758181	15.18	0.068915
3DStencil	0.743150	15.10	0.817918	13.75	0.074316
Mandelbrot	0.050256	45.08	0.056223	40.36	0.005165
MatrixMult	0.171354	96.23	0.188604	88.90	0.017137
ReverseArray	0.199197	25.21	0.219295	23.54	0.019945

TORMENT_ACC2016 Peak: 28.69
Speedup vs Sequential (peak): 41.17x
Speedup vs CUDA (peak): 0.64x

TORMENT_ACC2016 Average: 26.18
Speedup vs Sequential (average): 37.49x
Speedup vs CUDA (average): 0.59x

OpenUH Compiler Results:

OpenUH					
Benchmark	**Peak Time (s)**	**Peak Ratio**	**Avg. Time (s)**	**Avg. Ratio**	**Std. Deviation**
MonteCarloPi	0.159748	39.45	0.165055	38.21	0.006223
StringMatch	1.513772	7.58	1.529313	7.53	0.009267
3DStencil	0.729787	15.38	0.732878	15.34	0.001414
Mandelbrot	0.063368	35.75	0.065915	34.42	0.001012
MatrixMult	0.413113	39.91	0.413493	40.55	0.000271
ReverseArray	0.249520	20.12	0.251464	20.53	0.000648

TORMENT_ACC2016 Peak: 18.46
Speedup vs Sequential (peak): 26.49x
Speedup vs CUDA (peak): 0.41x

TORMENT_ACC2016 Average: 18.37
Speedup vs Sequential (average): 26.31x
Speedup vs CUDA (average): 0.41x

accULL Compiler Results:

accULL					
Benchmark	**Peak Time (s)**	**Peak Ratio**	**Avg. Time (s)**	**Avg. Ratio**	**Std. Deviation**
MonteCarloPi	0.135661	46.46	0.137047	46.02	0.003250
StringMatch	1.186638	9.66	1.188907	9.68	0.005512
3DStencil	5.644383	1.99	5.884403	1.91	0.088936
Mandelbrot	0.089364	25.35	0.090154	25.17	0.000442
MatrixMult	0.475147	34.70	0.478946	35.01	0.003708
ReverseArray	0.653959	7.68	0.655895	7.87	0.001687

TORMENT_ACC2016 Peak: 7.26
Speedup vs Sequential (peak): 10.42x
Speedup vs CUDA (peak): 0.16x

TORMENT_ACC2016 Average: 7.11
Speedup vs Sequential (average): 10.18x
Speedup vs CUDA (average): 0.16x

Peak score is the harmonic mean of the ratios of the best of 10 results. Average score is the harmonic mean of the ratios of the arithmetic mean of 10 results. In both cases higher is better.

Anexo IV

"Una herramienta de benchmarking para compiladores de OpenACC"

Jornadas de Paralelismo 2016

Este artículo ha sido desarrollado como consecuencia del trabajo realizado en este Trabajo de Fin de Grado. En el se presenta *TORMENT OpenACC2016* 0.91, una versión preliminar con solo dos benchmarks implementados, para las XXVII Jornadas de Paralelismo que se celebrarán este año en Salamanca. Este artículo es uno de los tres que han surgido como consecuencia de este Trabajo de Fin de Grado, como se indica en las conclusiones del Capítulo 8, y el único que puede ser incluido en esta memoria debido a que, a fecha de redacción de esta memoria, es el único que ha sido aceptado.

Una herramienta de benchmarking para compiladores de OpenACC

Daniel Barba[1] , Arturo González-Escribano[2] y Diego R. Llanos[3]

Resumen— **OpenACC es un modelo de programación paralela para aceleradores de tipo GPU y Xeon PHI que lleva en desarrollo algunos años. Durante este tiempo han aparecido distintos compiladores, tanto comerciales como de código abierto, que se encuentran aún en un estado temprano de desarrollo. Dado que tanto el estándar como sus implementaciones son relativamente recientes, disponer de una suite de benchmarks diseñada para soportar varios compiladores puede faciliar enormemente el análisis comparativo de rendimiento de los distintos códigos generados. En este artículo presentamos *TORMENT OpenACC*, una suite de benchmarks preparada para comparar tanto arquitecturas como el código generado por diferentes compiladores. Junto a esta herramienta hemos desarrollado unas métricas adecuadas para la comparación del rendimiento de los pares compilador-máquina y que hemos denominado *TORMENT_ACC Score*. Aquí se presenta la version 0.91 de la suite, desarrollada como prueba de concepto, y que incluye dos benchmarks. La version 1.0, considerada estable, incluirá seis benchmarks y estará disponible próximamente.**

Palabras clave— **OpenACC, compiladores, benchmarking.**

I. Introducción

OpenACC es un estándar abierto que define una serie de directivas de compilación o pragmas para ejecución de fragmentos de código en aceleradores de tipo GPU y Xeon Phi. Su objetivo es facilitar la paralelización de código secuencial en este tipo de aceleradores reduciendo el tiempo necesario tanto en la programación como en el aprendizaje [1]. La especificación se encuentra en su versión 2.5 [2].

La responsabilidad de la paralelización automática del código secuencial recae sobre los diferentes compiladores que soportan OpenACC: PGI Compiler [3], de *The Portland Group*, empresa ahora perteneciente a Nvidia, es (según los estudios que hemos realizado) el compilador con un grado de madurez más alto. Se encuentra disponible como parte del *Nvidia OpenACC Toolkit*, gratuito durante tres meses y con la posibilidad de adquirir licencias comerciales o de investigación. OpenUH [4], de la Universidad de Houston y accULL [5] de la Universidad de La Laguna son dos alternativas de código abierto desarrolladas en el ámbito académico y que pueden descargarse libremente para su uso.

Además de los compiladores señalados, existen otros compiladores que soportan OpenACC, pero debido a que en la fecha de redacción de este artículo fue imposible obtener licencias de prueba o académicas, han sido dejados fuera de nuestros estudios. Estos compiladores están siendo desarrollados por *CRAY Inc.* y *Pathscale Inc.*

Nuestro trabajo ha consistido en el desarrollo de una herramienta de análisis de rendimiento del código generado por estos compiladores, que hemos denominado *TORMENT OpenACC* (Trasgo perfORMance and EvaluatioN Tool for OpenACC). El motivo que nos ha movido a comenzar este trabajo es la escasez de herramientas similares para OpenACC y las carencias que presentan las existentes. Estas herramientas son, a fecha de redacción de este artículo, EPCC OpenACC Benchmark Suite [6] y Rodinia [7]. El primero ha sido desarrollado por el *Edinburgh Parallel Computing Centre* y consiste en una serie de microbenchmarks destinados a medir el overhead de la implementación de los distintos pragmas, y benchmarks para medir el rendimiento del código generado. Rodinia [7] es una traducción de los benchmarks originales de la suite del mismo nombre [8] para OpenACC desarrollado por *Pathscale*.

Nuestro trabajo busca dar respuesta a la necesidad de análisis del rendimiento y comparación del código generado por los distintos compiladores, intentando mantener un equilibrio entre las fortalezas y debilidades de los diferentes compiladores y tratando de obtener una herramienta que sea posible utilizar con los compiladores disponibles para la comunidad académica. Debido a que los compiladores se encuentran aún en fase de desarrollo, es imprescindible mantener el código de los benchmarks lo más sencillo posible.

Además del desarrollo de la herramienta como tal, hemos desarrollado también una métrica para poder ofrecer al usuario de esta herramienta una puntuación relativa que permita la comparación entre diferentes sistemas y compiladores analizados. Esta métrica ha sido denominada *TORMENT_ACC Score*.

El resto del artículo se organiza del siguiente modo: La sección II describe con más detenimiento las herramientas de benchmarking existentes. La sección III describe los compiladores actualmente soportados por *TORMENT OpenACC*. En la sección IV se describe la herramienta, sus objetivos y características a medir y enumera los benchmarks actualmente implementados, así como la métrica utilizada. Finalmente, la sección V concluye el artículo.

II. Herramientas existentes

Como se indicaba en la Introducción, a fecha de redacción de este artículo existen dos herramientas

[1]Dpto. de Informática, Universidad de Valladolid, España. e-mail: daniel@infor.uva.es.
[2]Dpto. de Informática, Universidad de Valladolid, España. e-mail: arturo@infor.uva.es.
[3]Dpto. de Informática, Universidad de Valladolid, España. e-mail: diego@infor.uva.es.

de análisis de rendimiento para OpenACC. A continuación describimos más en profundidad las mismas y las carencias detectadas.

A. EPCC OpenACC Benchmark Suite

La suite de benchmarks desarrollada por el *EPCC* ha sido diseñada específicamente para OpenACC. Se compone de tres partes diferenciadas y que se denominan *Nivel 0*, *Nivel 1* y *Nivel de Aplicaciones*.

En el *Nivel 0* se pueden encontrar una serie de microbenchmarks cuyo objetivo es medir el overhead generado por las implementaciones de los pragmas. Estos microbenchmarks dan un resultado que es la diferencia de dos tiempos de ejecución en función del pragma que están analizando, o bien el tiempo de transferencia de datos en una directiva de tipo `#pragma acc data`. Estos resultados son interesantes para el desarrollo de los compiladores, pero tal y como están diseñados no ofrecen una idea clara de rendimiento y los resultados pueden ser difíciles de interpretar.

En el *Nivel 1* se encuentran un conjunto de benchmarks típicos de tipo BLAS basados en *Polybench* y *Polybench/GPU* [9]. Los resultados ofrecidos por estos benchmarks son tiempos de ejecución de los diferentes códigos, por lo que son un buen indicador del rendimiento del código generado por los diferentes compiladores.

En el *Nivel de Aplicaciones* hay tres benchmarks de mayor entidad que los anteriores. Estos benchmarks también ofrecen tiempos de ejecución, pero al no ser problemas sintéticos dan resultados más interesantes.

En general, la idea del *EPCC OpenACC Benchmark Suite* está bien planteada y diseñada. No obstante, los resultados obtenidos en los microbenchmarks no son adecuados para un análisis de rendimiento. Los demás benchmarks, si bien podrían ser útiles, siempre han dado problemas en nuestros estudios. En unos casos no podían compilarse con algunos de los compiladores utilizados, en otros casos daban error de ejecución. Uno de los mayores problemas ha sido la limitación en los tamaños de los datos a utilizar debido a problemas en la implementación que llevaban a que se intentase asignar mucha más memoria de la deseada, originando errores en la ejecución y limitando a 10MB el tamaño de los datos.

B. Rodinia para OpenACC

La empresa *Pathscale Inc.* ha desarrollado una versión [7] de la suite de benchmarks Rodinia [8], [10] para su uso con compiladores de OpenACC. Con la versión existente en GitHub a día 25 de Abril de 2014 (versión más reciente durante el desarrollo de nuestro trabajo), en general los benchmarks disponibles no compilan con ninguno de los compiladores existentes, salvo contadas excepciones. El compilador de PGI es el único que logra compilar un número significativo de benchmarks.

La suite se compone de benchmarks que devuelven el tiempo de ejecución, por lo que serían un buen punto de partida para un análisis de rendimiento. Por desgracia, la escasa madurez del código y la poca compatibilidad con los compiladores disponibles hacen imposible su uso para establecer una comparativa de rendimientos.

III. Compiladores Soportados

En la Introducción hemos enumerado brevemente los compiladores disponibles. Su elección está motivada por la existencia de licencias gratuitas o académicas, o bien por ser de código abierto. A continuación explicaremos algunos detalles de los compiladores soportados en la versión preliminar de *TORMENT OpenACC*.

A. PGI Compiler

El compilador de PGI [3], desarrollado por *The Portland Group* y Nvidia, es a fecha de redacción de este artículo el compilador con un grado de madurez más alto. Su uso está muy extendido en los diferentes talleres y conferencias que se realizan sobre OpenACC. Actualmente está disponible como parte del *Nvidia OpenACC toolkit*, que incluye una licencia gratuita de tres meses y la posibilidad de adquirir una licencia comercial o académica. En nuestros estudios, el compilador de PGI ha demostrado ser el más sólido y el que más alto grado de madurez tiene. Se ajusta bien a la especificación de OpenACC. Su instalación es simple y dispone de abundante documentación.

B. OpenUH

El compilador OpenUH, desarrollado por la Universidad de Houston, es una alternativa de código abierto. Comparado con el compilador de PGI, su madurez y solidez son menores y carece de algunas funcionalidades, como reducciones sobre mínimos o máximos, lo cual dificulta ligeramente la programación de aplicaciones. Su instalación, a fecha de redacción de este artículo, no es tan sencilla como cabría esperar. La versión pre-compilada disponible en su web [4] carece de algunas librerías que deben ser compiladas aparte u obtenidas de otra forma.

C. accULL

El compilador accULL [5], desarrollado por la Universidad de La Laguna, es (al igual que OpenUH) un compilador de código abierto. De los tres compiladores soportados, es el que tiene una menor robustez, teniendo problemas para la traducción fuente a fuente de algunas características de C, como punteros a función, o de funcionalidades de OpenACC, como algunos tipos de reducciones. La instalación de la versión disponible en su web [11] es bastante simple, siguiendo las instrucciones que pueden encontrarse adjuntas al compilador en ficheros de texto.

IV. TORMENT OpenACC

Nuestra propuesta en desarrollo se denomina *TORMENT OpenACC* y es el acrónimo para Trasgo perfORMance and EvaluatioN Tool for OpenACC.

El proyecto ha nacido para intentar dar una solución a la necesidad de dar respuesta a los intentos de realizar comparativas de rendimiento de código OpenACC.

A. Motivación

Las herramientas actuales han demostrado en nuestros estudios previos no ser suficientes para dar una respuesta sencilla a la evaluación del código generado por los diferentes compiladores de OpenACC.

El primero de los problemas es la escasa compatibilidad del código de los benchmarks entre los diferentes compiladores. Es cierto que si el código se ajusta al estándar debería compilarse y ejecutarse correctamente en cualquier implementación del estándar OpenACC, pero en la actual etapa en la que se encuentran tanto el estándar como los compiladores esto no se cumple. Debido a esto, hemos desarrollado *TORMENT OpenACC* teniendo en cuenta tanto la especificación del estándar como las capacidades actuales de los diferentes compiladores.

Otro problema detectado es que muchos de los benchmarks no ofrecen una imagen clara de rendimiento, como es el caso de los microbenchmarks del *EPCC OpenACC Benchmark Suite*. Si bien es cierto que estos microbenchmarks son útiles para el desarrollo de los compiladores, creemos que el overhead de la implementación de los distintos pragmas no es realmente relevante en una comparativa de rendimientos. Por este motivo, en nuestra propuesta dejamos fuera ese tipo de pruebas.

Finalmente, hemos observado también que, en muchos casos, benchmarks que parecen compilar correctamente luego generan errores en tiempo de ejecución o resultados incorrectos. Esta última situación puede suponer que, si no se detecta el resultado erróneo, los resultados de rendimiento obtenidos sean también incorrectos. Para evitar esto, además de incorporar comprobación de resultados a los benchmarks, también hemos analizado los resultados durante el desarrollo de nuestra propuesta, para que los benchmarks que incorporamos en nuestra herramienta compilen y ejecuten correctamente con cualquiera de los compiladores.

Otro aspecto importante que hemos tenido en cuenta a la hora de llevar a cabo el desarrollo de *TORMENT OpenACC* es la ofrecer a la comunidad una herramienta que, además de facilitar el análisis de rendimiento del código generado por los compiladores de OpenACC ejecutados en distintas máquinas, ofrezcan una medida del rendimiento que permita una comparación fácil entre máquinas y compiladores. Esta idea nos ha llevado al desarrollo de las métricas *TORMENT_ACC Score* que describiremos posteriormente.

B. Objetivos

El objetivo principal de *TORMENT OpenACC* es la de permitir un análisis de rendimiento de código OpenACC generado por los distintos compiladores de forma sencilla, generando un resumen de resultados fácilmente analizable y ofreciendo unas métricas, denominadas *TORMENT_ACC Score*, que permiten la comparación de los pares máquina-compilador.

Nuestra propuesta pretende facilitar a la comunidad una herramienta preparada específicamente para OpenACC que sea consciente del estado de desarrollo temprano de los compiladores existentes, de forma que se eviten problemas en compilación o ejecución como sucede con otras herramientas de benchmarking existentes. *TORMENT OpenACC* estará preparado para compilarse y ejecutarse con la menor intervención posible del usuario. Los scripts que acompañan a la suite recopilan la información del proceso y ofrecen finalmente un informe en formato HTML con los datos relevantes.

Además, *TORMENT OpenACC* utiliza los compiladores GCC y NVCC para obtener datos de ejecuciones de código secuencial y código CUDA respectivamente. De este modo, nuestra propuesta ofrece al usuario información del *speedup* con respecto al código secuencial y CUDA ejecutado en la misma máquina.

C. Estructura de la herramienta

TORMENT OpenACC se compone de una serie de *scripts* que se encargan de todo el proceso de compilación, ejecución y obtención de resultados, eliminando esta carga al usuario. Estos *scripts* se dividen en tres categorías. Los scripts de configuración guían al usuario en la obtención de las rutas correctas a librerías CUDA (necesarias por algunos de los compiladores), rutas de compiladores y comandos de ejecución (en caso de que el usuario utilice, por ejemplo, sistemas de colas tipo `slurm`). El *script* de ejecución se encarga de la compilación y ejecución usando todos los compiladores que hayan sido hallados en el sistema del usuario. Finalmente, el *script* de generación de resultados procesa los resultados obtenidos y genera un fichero HTML con el informe final.

Cada benchmark está desarrollado de modo que sea lo más sencillo posible, para evitar problemas de compilación. Por este motivo se procura evitar el uso de niveles de indirección que en otras situaciones serían aconsejables. Cada benchmark está definido en su propia unidad de compilación y el código OpenACC y CUDA está incorporado en el mismo fuente, utilizando compilación condicional y evitando tener de forma simultanea ficheros `.c` y `.cu` con código duplicado. Los ficheros `.cu` son necesarios para que el compilador NVCC genere el código CUDA correspondiente, pero esto se ha resuelto mediante el uso de enlaces simbólicos a los ficheros `.c` correspondientes.

El programa principal se encarga del lanzamiento de los benchmarks. Cada uno se lanza con diez repeticiones, con una repetición extra al comienzo cuyos resultados son desechados. Al finalizar estas diez repeticiones se obtienen los valores denominados *peak* y *average* y que corresponden a la mejor de las ejecuciones y a la media aritmética de todas ellas. Estos valores servirán para calcular la métrica de la que

```
CUDA_ATTR__ int getRandom(unsigned int* seed)
{
    unsigned int next = *seed;
    int result;

    next *= 1103515245;
    next += 12345;
    result = (unsigned int) (next/65536) % 2048;

    next *= 1103515245;
    next += 12345;
    result <<= 10;
    result ^= (unsigned int) (next/65536) % 1024;

    next *= 1103515245;
    next += 12345;
    result <<= 10;
    result ^= (unsigned int) (next/65536) % 1024;

    *seed = next;
    return result;
}
```

Fig. 1
Código para la generación de números aleatorios.

hablaremos posteriormente.

D. Benchmarks implementados

La versión preliminar de *TORMENT OpenACC* contiene únicamente dos benchmarks. El desarrollo de la herramienta sigue en proceso y otros benchmarks se irán añadiendo progresivamente.

D.1 MonteCarloPi

Este benchmark consiste en una aproximación de Pi por el método de Monte Carlo, que se basa en la generación de puntos aleatorios en un cuadrado de lado unitario. Se comprueba si estos puntos se encuentran dentro de un cuarto de círculo de radio unitario y se acumula el total de puntos que cumplen dicha condición. Finalmente, se aplica la siguiente fórmula:

$$\pi \approx \frac{4*P}{T}$$

Donde P es el número de puntos dentro del cuarto de círculo y T es el total de puntos generados.

MonteCarloPi es un benchmark que no tiene prácticamente transferencias de memoria y que realiza un cálculo computacional muy simple, pero puede ser optimizado en CUDA haciendo que cada hilo calcule varios puntos y utilizando la *shared memory* de los bloques para evitar accesos a memoria global. Un buen resultado de los compiladores en este benchmark dependerá de estos factores.

Dado que no se puede hacer uso de la función srand de C en el código ejecutado en la GPU, y el uso de la librería curand se limita a código CUDA, hemos decidido replicar la función srand para permitir su ejecución en la GPU, como se indica en la Fig. 1.

El código utilizado para las versiones de OpenACC y CUDA se muestra en las Figs. 2 y 3.

```
#pragma acc parallel loop \
    private(i, d, x, y, seed) \
    reduction(+:count)
for(i = 0; i < COORD_NUM; ++i){
    seed = 1987 ^ i*27;
    x = (double)getRandom(&seed)/(double)RAND_MAX;
    y = (double)getRandom(&seed)/(double)RAND_MAX;

    d = sqrt(x*x + y*y);
    if(d <= 1.0){
        ++count;
    }
}
```

Fig. 2
Código de MonteCarloPi para OpenACC.

```
__CUDA_GLOBAL__ void piKernel(
        const unsigned long int triesPerThread,
        unsigned long int* hits)
{
    unsigned int seed;
    int gid, tid, bid;
    int lhits;
    float x, y;
    extern __shared__ unsigned long int sdata[];

    gid = (blockIdx.x*blockDim.x) + threadIdx.x;
    tid = threadIdx.x;
    bid = blockIdx.x;
    lhits = 0;

    seed = 1987 ^ gid*27;

    for (int i = 0; i < triesPerThread; ++i){
        x = (float)getRandom(&seed)/(float)RAND_MAX;
        y = (float)getRandom(&seed)/(float)RAND_MAX;

        float d = sqrt(x*x + y*y);
        if (d <= 1.0f){
            ++lhits;
        }
    }
    sdata[tid] = lhits;
    __syncthreads();
    if (tid == 0){
        for (int i = 1; i < blockDim.x; ++i){
            lhits += sdata[i];
        }

    hits[bid] = lhits;
    }
}
```

Fig. 3
Código de MonteCarloPi para CUDA.

```
#pragma acc data copyin(S[0:sizeS],B[0:sizeB]) \
    copy(C[0:sizeB])
{
#pragma acc parallel loop private(startB) \
    firstprivate(result,sizeS,sizeB)
for (startB = 0; startB <= sizeB-sizeS; startB++){
    C[startB] = 0;
    if (startB <= result) {
        int ind;
        for(ind = 0; ind < sizeS; ind++){
            if (S[ind] != B[startB+ind]) break;
        }
        if (ind == sizeS){
            result = startB;
            C[startB] = 1;
        }
    }
}
}

result = -1;
for (startB = 0; startB <= sizeB-sizeS; startB++){
    if (C[startB] == 1){
        result = startB;
        break;
    }
}
```

Fig. 4
Código de StringMatch para OpenACC.

D.2 StringMatch

El benchmark *StringMatch* es un programa de alineamiento de cadenas de caracteres. El programa consiste en la búsqueda de la primera ocurrencia de una cadena pequeña en una cadena grande, utilizando un algoritmo *naïve*. En este benchmark, la cadena grande tiene una longitud de 10 millones de caracteres, es decir, aproximadamente 10MB. Las cadenas pequeñas, que son cuatro, tienen una longitud de 1000 caracteres.

Este algoritmo es interesante porque combina transferencia de datos con la necesidad de un uso eficiente de la memoria, especialmente de la *shared memory*. Los códigos de las versiones de OpenACC y CUDA pueden verse en las Figs. 4 y 5.

E. Métrica utilizada

Para la elección de la métrica denominada *TORMENT_ACC Score* hemos decidido optar por la metodología utilizada por SPEC [12]. El *System Performance Evaluation Council*, comúnmente conocido como SPEC, es un referente ampliamente conocido en cuanto a benchmarking y análisis de rendimiento se refiere. Una de sus fortalezas es el reconocer que los benchmarks envejecen en función del paso del tiempo y, en consecuencia, deben ser actualizados.

SPEC utiliza la siguiente metodología. En primer lugar, cada programa devuelve su tiempo de ejecución y se calcula el *SPECratio*, que consiste en el ratio obtenido de dividir un tiempo de ejecución de referencia suministrado por SPEC entre el tiempo de ejecución obtenido. Finalmente, se obtiene la media geométrica de todos los SPECratios del conjunto de

```
__global__ void kernel_busqueda(const char* b_idata,
        const int sizeB, const char* s_idata,
        const int sizeS, unsigned int* result)
{
    extern __shared__ unsigned char sdata[];
    int gid = (blockIdx.x*blockDim.x) + threadIdx.x;
    int tid = threadIdx.x;
    int sStart = THREADS_PER_BLOCK + sizeS;
    int offset;

    if (gid == 0){
        *result = -1;
    }

    __syncthreads();

    if (gid < *result){
        unsigned char* bd = &sdata[tid];
        b_idata += gid;
        unsigned char* sd = &sdata[sStart+tid];
        s_idata += tid;
        if(tid<THREADS_PER_BLOCK){
            for(offset = 0; tid+offset<sStart;
                    offset += THREADS_PER_BLOCK)
            {
                if (offset==0){
                    *bd = *b_idata;
                    bd+=THREADS_PER_BLOCK;
                    b_idata+=THREADS_PER_BLOCK;
                    continue;
                }
                *bd = *b_idata;
                bd+=THREADS_PER_BLOCK;
                b_idata+=THREADS_PER_BLOCK;
                *sd = *s_idata;
                sd+=THREADS_PER_BLOCK;
                s_idata+=THREADS_PER_BLOCK;
            }
        }

        __syncthreads();

        if (gid <= sizeB-sizeS){
            unsigned int i;
            unsigned int b = 1;
            unsigned char* sd = &sdata[sStart];
            unsigned char* bd = &sdata[tid];
            for(i = 0; b && i+15 < sizeS; i+=16){
                b &= (*sd == *bd); sd++; bd++;
                b &= (*sd == *bd); sd++; bd++;
                b &= (*sd == *bd); sd++; bd++;
                b &= (*sd == *bd); sd++; bd++;
                b &= (*sd == *bd); sd++; bd++;
                b &= (*sd == *bd); sd++; bd++;
                b &= (*sd == *bd); sd++; bd++;
                b &= (*sd == *bd); sd++; bd++;
                b &= (*sd == *bd); sd++; bd++;
                b &= (*sd == *bd); sd++; bd++;
                b &= (*sd == *bd); sd++; bd++;
                b &= (*sd == *bd); sd++; bd++;
                b &= (*sd == *bd); sd++; bd++;
                b &= (*sd == *bd); sd++; bd++;
                b &= (*sd == *bd); sd++; bd++;
                b &= (*sd == *bd); sd++; bd++;
                if (!b) break;
            }
            for(; b && i < sizeS ; i++){
                b &= (*sd == *bd); sd++; bd++;
                if (!b) break;
            }
            if ( b ) atomicMin(result, gid);
        }
    }
}
```

Fig. 5
Código de StringMatch para CUDA.

benchmarks [13].

Nuestra propuesta sigue una idea similar a la de SPEC, pero con algunas variaciones. En primer lugar, la ejecución de los benchmarks que componen *TORMENT OpenACC* devuelven tres valores. *Peak Time* es el mejor tiempo de ejecución obtenido, medido en segundos. *Average Time* es el tiempo medio de todas las ejecuciones del benchmark, también en segundos. Finalmente, *Standard Deviation* es la desviación estándar del conjunto de medidas e indica la variabilidad obtenida en las ejecuciones del benchmark. Con los tiempos *peak* y *average* se calcula un ratio con respecto a los tiempos de referencia suministrados con la herramienta, y que son los tiempos de ejecución secuencial del benchmark en una máquina de referencia. Una vez ejecutados todos los benchmarks, se obtiene la media armónica de todos los ratios, tanto para *peak time* como *average time*. Estos valores que se obtienen son *TORMENT_ACC Peak Score* y *TORMENT_ACC Average Score*.

La principal diferencia entre *TORMENT OpenACC* y SPEC, aparte de la metodología de toma de tiempos de ejecución, consiste en el uso de la media armónica en lugar de la media geométrica. Esta decisión se fundamenta en el hecho de que, para los objetivos de *TORMENT OpenACC*, la media armónica es más adecuada que la media geométrica. En primer lugar, aunque la media geométrica siempre produce una ordenación consistente, no necesariamente produce la ordenación correcta [13], ya que esta media no es inversamente proporcional al tiempo de ejecución. En cambio, la media armónica si que es inversamente proporcional al tiempo de ejecución, lo que hace que sea una media correcta para expresar ratios. Estas afirmaciones son compartidas por otros trabajos, como por ejemplo [14].

Como ejemplo de uso, las figuras 6 y 7 muestran dos informes generados para sistemas con diferente configuración. En el primero puede verse que el compilador de PGI obtiene el *TORMENT_ACC Score* más alto, tanto *Peak* como *Average*, con un valor de 28.56 y 25.71 respectivamente. Los resultados con respecto a la implementación en CUDA de los benchmarks, muestran que los tres compiladores generan código cuyo rendimiento queda muy lejos del conseguido usando CUDA. En el segundo puede observarse que a pesar de utilizar una GPU mucho más modesta en comparación, los resultados son consistentes con los del primer informe. En ambos resultados se aprecia como el compilador de PGI obtiene los valores más grandes de la desviación típica. Esto parece indicar que al generar el código se está modificando la semántica del programa y la primera ejecución de cada benchmark que debería ser desechada se está incluyendo en las medidas.

V. Conclusiones y Trabajo Futuro

TORMENT OpenACC es una herramienta de análisis y comparación de rendimientos de código generado por compiladores de OpenACC, teniendo en consideración el nivel de madurez de tanto el estándar OpenACC como de los diferentes compiladores. *TORMENT OpenACC* desarrolla una suite de benchmarks específicamente diseñados para OpenACC y manteniendo la máxima portabilidad entre compiladores, permitiendo su compilación y ejecución en todos ellos.

Los resultados ofrecidos por *TORMENT OpenACC* incluyen la denominada *TORMENT_ACC Score*, diseñada para la comparación de pares máquina-compilador. Además, se ofrece un resumen de la ejecución de los benchmarks con una tabla de tiempos y ratios, tanto mínimos como medios e incluyendo la desviación estándar para un análisis de variabilidad de los tiempos de ejecución del código generado.

Junto con los resultados de los compiladores de OpenACC se incluyen también resultados de ejecución de código generado por los compiladores GCC y NVCC para código secuencial y CUDA. De este modo, se puede ofrecer al usuario una comparativa a nivel de máquina del rendimiento del código generado por los compiladores de OpenACC con respecto a versiones secuenciales y CUDA de los benchmarks.

El trabajo futuro a desarrollar consiste en la ampliación de la suite de benchmarks, para lograr resultados que sean verdaderamente relevantes. Aquí se presenta la versión 0.91 de la suite, desarrollada como prueba de concepto, y que incorpora dos benchmarks. La versión 1.0, considerada estable, incluirá seis benchmarks y estará disponible próximamente, Tratando de cubrir los aspectos más interesantes de la ejecución de código en las GPUs. Además, una parte importante del trabajo restante consiste en mantener la compatibilidad entre compiladores y asegurar el correcto funcionamiento de la herramienta en distintas máquinas.

Agradecimientos

En memoria de Agustín de Dios Hernández.

Esta investigación ha sido parcialmente financiada por el MICINN y el programa ERDF de la Unión Europea: proyecto HomProg-HetSys (TIN2014-58876-P), la red CAPAP-H5 (TIN2014-53522-REDT) y el COST Program Action IC1305: Network for Sustainable Ultrascale Computing (NESUS).

Referencias

[1] OpenACC-standard.org, "About OpenACC," .
[2] OpenACC-standard.org, "OpenACC 2.5 draft for public comment," oct 2015.
[3] PGI, "Pgi accelerator compilers with OpenACC directives," `https://www.pgroup.com/resources/accel.htm`, nov 2015.
[4] University of Houston, "Open-source UH compiler," `http://web.cs.uh.edu/~openuh/download/`, nov 2015.
[5] Ruymán Reyes, Iván López-Rodríguez, Juan J Fumero, and Francisco de Sande, "accULL: an OpenACC implementation with CUDA and OpenCL support," in *Euro-Par 2012 Parallel Processing*, pp. 871–882. Springer, 2012.
[6] EPCC, "Epcc OpenACC benchmark suite," `https://github.com/EPCCed/epcc-openacc-benchmarks`, sep 2013.
[7] Pathscale, "Rodinia benchmark suite 2.1 with OpenACC

port," `https://github.com/pathscale/rodinia`, apr 2014.

[8] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. (IISWC), 2009 IEEE International Symposium on*. IEEE, 2009, pp. 44–54.

[9] Louis-Noël Pouchet, "Polybench: The polyhedral benchmark suite," *URL: http://www. cs. ucla. edu/˜ pouchet/software/polybench/[cited July,]*, 2012.

[10] Shuai Che, Jeremy W Sheaffer, Michael Boyer, Lukasz G Szafaryn, Liang Wang, and Kevin Skadron, "A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads," in *Workload Characterization (IISWC), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–11.

[11] Universidad de La Laguna, "accULL," `http://cap.pcg.ull.es/es/accULL`, nov 2015.

[12] Kaivalya M Dixit, "The spec benchmarks," *Parallel computing*, vol. 17, no. 10, pp. 1195–1209, 1991.

[13] David J Lilja, *Measuring computer performance: a practitioner's guide*, Cambridge University Press, 2005.

[14] John R Mashey, "War of the benchmark means: time for a truce," *ACM SIGARCH Computer Architecture News*, vol. 32, no. 4, pp. 1–14, 2004.

TORMENT OpenACC Result page

Grupo Trasgo - Universidad de Valladolid

UVa

System Information

Benchmark version: 0.91
Hostname: hydra
Username: daniel

Software stack info
Kernel: Linux 3.10.0-229.4.2.el7.x86_64 x86_64
GCC: gcc (GCC) 4.8.3 20140911 (Red Hat 4.8.3-9)
NVCC: Cuda compilation tools, release 7.5, V7.5.17
PGI Compiler: pgcc 15.7-0 64-bit target on x86-64 Linux -tp haswell
OpenUH Compiler: OpenUH 3.1.0 (based on Open64 Compiler Suite: Version 5.0)
accULL Compiler: Release 0.4alpha

CPU info
Model: Intel(R) Xeon(R) CPU E5-2609 v3 @ 1.90GHz
Architecture: x86_64
Number of Cores: 6
Max MHz: MHz
Min MHz: MHz
L1 Cache: 32K
L2 Cache: 256K
L3 Cache: 15360K
RAM: 65687144 kB

GPU(s) info
GPU0: NVIDIA Corporation GK110B [GeForce GTX Titan Black] (rev a1)
GPU1: NVIDIA Corporation GK110B [GeForce GTX Titan Black] (rev a1)
GPU2: NVIDIA Corporation GK110B [GeForce GTX Titan Black] (rev a1)
GPU3: NVIDIA Corporation GK110B [GeForce GTX Titan Black] (rev a1)

GCC_Sequential Compiler Results:

GCC_Sequential					
Benchmark	**Peak Time (s)**	**Peak Ratio**	**Avg. Time (s)**	**Avg. Ratio**	**Std. Deviation**
MonteCarloPi	14.763701	0.43	14.764007	0.43	0.000336
StringMatch	20.118267	0.57	20.136980	0.57	0.021079

NVCC_CUDA Compiler Results:

NVCC_CUDA					
Benchmark	**Peak Time (s)**	**Peak Ratio**	**Avg. Time (s)**	**Avg. Ratio**	**Std. Deviation**
MonteCarloPi	0.016409	383.96	0.017840	354.36	0.001254
StringMatch	0.105644	108.48	0.105889	109.21	0.000162

PGI Compiler Results:

PGI					
Benchmark	**Peak Time (s)**	**Peak Ratio**	**Avg. Time (s)**	**Avg. Ratio**	**Std. Deviation**
MonteCarloPi	0.067226	93.72	0.077228	81.86	0.008222
StringMatch	0.688718	16.64	0.758494	15.25	0.068944

TORMENT_ACC_0.91 Peak Score: 28.26
Speedup vs Sequential (peak): 57.92x
Speedup vs CUDA (peak): 0.17x

TORMENT_ACC_0.91 Average Score: 25.71
Speedup vs Sequential (average): 52.40x
Speedup vs CUDA (average): 0.15x

OpenUH Compiler Results:

OpenUH					
Benchmark	**Peak Time (s)**	**Peak Ratio**	**Avg. Time (s)**	**Avg. Ratio**	**Std. Deviation**
MonteCarloPi	0.153154	41.14	0.165321	38.24	0.006447
StringMatch	1.574083	7.28	1.578459	7.33	0.002537

TORMENT_ACC_0.91 Peak Score: 12.37
Speedup vs Sequential (peak): 25.35x
Speedup vs CUDA (peak): 0.07x

TORMENT_ACC_0.91 Average Score: 12.30
Speedup vs Sequential (average): 25.07x
Speedup vs CUDA (average): 0.07x

accULL Compiler Results:

accULL					
Benchmark	**Peak Time (s)**	**Peak Ratio**	**Avg. Time (s)**	**Avg. Ratio**	**Std. Deviation**
MonteCarloPi	0.135701	46.43	0.139198	45.42	0.007257
StringMatch	1.173044	9.77	1.184555	9.76	0.004565

TORMENT_ACC_0.91 Peak Score: 16.14
Speedup vs Sequential (peak): 33.08x
Speedup vs CUDA (peak): 0.10x

TORMENT_ACC_0.91 Average Score: 16.07
Speedup vs Sequential (average): 32.76x
Speedup vs CUDA (average): 0.10x

Peak score uses the best of 10 results. Average score uses the arithmetic mean of 10 results. In both cases higher is better.

Fig. 6
Página de Resultados TORMENT OpenACC

TORMENT OpenACC Result page

Grupo Trasgo - Universidad de Valladolid

UVa

System Information

Benchmark version: 0.91
Hostname: corikLaptop
Username: dani

Software stack info
Kernel: Linux 3.16.0-4-amd64 x86_64
GCC: gcc (Debian 4.9.2-10) 4.9.2
NVCC: Cuda compilation tools, release 7.5, V7.5.17
PGI Compiler: pgcc 15.7-0 64-bit target on x86-64 Linux -tp haswell
OpenUH Compiler: OpenUH 3.1.0 (based on Open64 Compiler Suite: Version 5.0)
accULL Compiler: Release 0.4alpha

CPU info
Model: Intel(R) Core(TM) i5-4200H CPU @ 2.80GHz
Architecture: x86_64
Number of Cores: 4
Max MHz: 3400.0000 MHz
Min MHz: 800.0000 MHz
L1 Cache: 32K
L2 Cache: 256K
L3 Cache: 3072K
RAM: 3939364 kB

GPU(s) info
GPU0: NVIDIA Corporation GM107M [GeForce GTX 850M] (rev a2)

GCC_Sequential Compiler Results:

GCC_Sequential					
Benchmark	**Peak Time (s)**	**Peak Ratio**	**Avg. Time (s)**	**Avg. Ratio**	**Std. Deviation**
MonteCarloPi	6.196389	1.02	6.196760	1.02	0.000310
StringMatch	11.292974	1.01	11.298524	1.02	0.006537

NVCC_CUDA Compiler Results:

NVCC_CUDA					
Benchmark	**Peak Time (s)**	**Peak Ratio**	**Avg. Time (s)**	**Avg. Ratio**	**Std. Deviation**
MonteCarloPi	0.057900	108.81	0.062234	101.58	0.004161
StringMatch	0.335251	34.19	0.335390	34.48	0.000085

PGI Compiler Results:

PGI					
Benchmark	**Peak Time (s)**	**Peak Ratio**	**Avg. Time (s)**	**Avg. Ratio**	**Std. Deviation**
MonteCarloPi	0.343979	18.32	0.378440	16.70	0.034409
StringMatch	1.925690	5.95	2.120944	5.45	0.192720

TORMENT_ACC_0.91 Peak Score: 8.98
Speedup vs Sequential (peak): 8.84x
Speedup vs CUDA (peak): 0.17x

TORMENT_ACC_0.91 Average Score: 8.22
Speedup vs Sequential (average): 8.05x
Speedup vs CUDA (average): 0.16x

OpenUH Compiler Results:

OpenUH					
Benchmark	**Peak Time (s)**	**Peak Ratio**	**Avg. Time (s)**	**Avg. Ratio**	**Std. Deviation**
MonteCarloPi	0.697705	9.03	0.698031	9.06	0.000198
StringMatch	2.224097	5.15	2.226121	5.19	0.003758

TORMENT_ACC_0.91 Peak Score: 6.56
Speedup vs Sequential (peak): 6.46x
Speedup vs CUDA (peak): 0.13x

TORMENT_ACC_0.91 Average Score: 6.60
Speedup vs Sequential (average): 6.46x
Speedup vs CUDA (average): 0.13x

accULL Compiler Results:

accULL					
Benchmark	**Peak Time (s)**	**Peak Ratio**	**Avg. Time (s)**	**Avg. Ratio**	**Std. Deviation**
MonteCarloPi	0.675184	9.33	0.676132	9.35	0.000704
StringMatch	2.298089	4.99	2.299703	5.03	0.003703

TORMENT_ACC_0.91 Peak Score: 6.50
Speedup vs Sequential (peak): 6.40x
Speedup vs CUDA (peak): 0.12x

TORMENT_ACC_0.91 Average Score: 6.54
Speedup vs Sequential (average): 6.40x
Speedup vs CUDA (average): 0.13x

Peak score uses the best of 10 results. Average score uses the arithmetic mean of 10 results. In both cases higher is better.

Fig. 7
Página de Resultados TORMENT OpenACC

Anexo V

Manual de instalación y uso

Instalación

TORMENT OpenACC2016 no requiere ningún tipo de instalación especial y basta con extraer el contenido del fichero tar.gz en un directorio en el que se posean los adecuados permisos de lectura y escritura.

Dependencias

Para el correcto funcionamiento de *TORMENT OpenACC2016*, se debe diponer de las siguientes dependencias:

- Sistema Linux con disponibilidad de los comandos: cat, grep, sort, cut, sed, awk y make.
- Shell Bash o equivalente.
- Compilador GCC
- Compilador NVCC con el toolkit CUDA correctamente instalado.
- Al menos un compilador de los siguientes: PGI, OpenUH o accULL.

Uso

Para usar *TORMENT OpenACC2016*, basta con ejecutr el script run.sh que se encuentra en el directorio raiz de la herramienta. Este script guiará al usuario en el proceso de ejecución de *TORMENT OpenACC2016*, incluyendo la configuración si fuera necesario.

Configuración Para configurar *TORMENT OpenACC2016*, ejecutar el script configure.sh. Se procederá a buscar las rutas necesarias, pidiendo confirmación al usuario.

Generación de resúmenes Si se desea volver a generar los resúmenes de resultados de una ejecución anterior de *TORMENT OpenACC2016*, sin volver a ejecutar los benchmarks, ejecutar el script gen_reports.sh. El proceso es automático y como resultado generara un nuevo fichero HTML con los resultados.