Encapsulated Synchronization and Load-Balance in Heterogeneous Programming

Conference Paper · August 2012

DOI: 10.1007/978-3-642-32820-6_50

Yuri Torres
Universidad de Valladolid
37 PUBLICATIONS **252** CITATIONS
SEE PROFILE

Arturo Gonzalez-Escribano
Universidad de Valladolid
145 PUBLICATIONS **803** CITATIONS
SEE PROFILE

Diego R. Llanos
Universidad de Valladolid
156 PUBLICATIONS **918** CITATIONS
SEE PROFILE

Encapsulated synchronization and load-balance in heterogeneous programming

Yuri Torres, Arturo Gonzalez-Escribano, and Diego Llanos

Departamento de Informatica, Universidad de Valladolid
{yuri.torres,arturo,diego}@infor.uva.es

Abstract. Programming models and techniques to exploit parallelism in accelerators, such as GPUs, are different from those used in traditional parallel models for shared- or distributed-memory systems. It is a challenge to blend different programming models to coordinate and exploit devices with very different characteristics and computation powers. This paper presents a new extensible framework model to encapsulate run-time decisions related to data partition, granularity, load balance, synchronization, and communication for systems including assorted GPUs. Thus, the main parallel code becomes independent of them, using internal topology and system information to transparently adapt the computation to the system. The programmer can develop specific functions for each architecture, or use existent specialized library functions for different CPU-core or GPU architectures. The high-level coordination is expressed using a programming model built on top of message-passing, providing portability across distributed- or shared-memory systems. We show with an example how to produce a parallel code that can be used to efficiently run on systems ranging from a Beowulf cluster to a machine with mixed GPUs. Our experimental results show how the run-time system, guided by hints about the computational-power ratios of different devices, can automatically part and distribute large computations across heterogeneous systems, improving the overall performance.

1 Introduction

Currently, heterogeneous systems provide computing power using mixed types of devices and architectures, such as CPU-cores, GPUs or FPGAs [6, 10]. General-Purpose Programming for devices such as GPUs (GP-GPU) has been simplified by the introduction of higher level data parallel languages, such as CUDA or OpenCL. However, to obtain efficient codes the programmer needs knowledge about the underlying target architecture, and how it relates to the programming model. The intrinsic complexity of the code generation for heterogeneous systems increases every time we add any different hardware device. Thus, it is an important goal to devise abstractions and tools that allow the programmer to blend the different programming models involved, also simplifying the tasks of data-distribution and device coordination across an heterogeneous system.

In previous works we presented Hitmap [3, 5], a library to support both data and task parallelism in distributed-memory environments, through manipulation and mapping of hierarchical tiling arrays Hitmap features an extensible plug-in system that allows the programmer to choose among different data-partition and distribution techniques, or easily program and reuse new ones. It provides functionalities for tile communication, allowing to build complex and scalable communication patterns in terms of the results of the mapping functions.

This paper presents a new framework model to encapsulate run-time decisions related to data partition, granularity, load balance, synchronization, and communication for heterogeneous systems. It introduces a new abstraction layer in the conceptual structure of Hitmap. More precisely, in this work we present the following contributions:

(1) We propose a new plug-ins layer to encapsulate the decisions related to map tile computations to specific accelerator devices. We discuss how load-balancing techniques relate to the different plug-ins layers.

(2) We introduce a high-level API that selects the proper kernel for a given device, and hides all details of synchronization and communication between logical processes and accelerators.

(3) We discuss an implementation of this framework model currently supporting distributed-memory clusters of multicore CPUs and NVIDIA GPUs.

(4) We show with an example how to produce a single parallel code that adapts the computation to efficiently run on systems ranging from a Beowulf cluster to a machine with mixed GPUs. Our experimental results show how the run-time system can automatically part and distribute large computations across very different devices, improving the performance of a homogeneous approach.

The rest of the paper is organized as follows. Section 2 discusses some previous approaches and their limitations. Section 3 introduces our conceptual approach. Section 4 describes the architecture of our solution and the design problems faced. Section 5 shows a case study. In section 6 we present a performance evaluation of the case study with a load-balancing strategy in different scenarios. Finally, section 7 discusses some conclusions and future work.

2 Related work

Several research groups are working in the problem of simplifying heterogeneous programming without sacrificing hardware accelerators performance.

Quintana-Ortí *et al.* [11] presented the FLAME programming model. It focus on programming dense linear algebra operations on complex platforms, including multicore processors, and hardware accelerators such as GPUs, and Cell B.E. FLAME abstracts the target accelerator architecture. It divides the parallelism in two levels, the first one considering each accelerator device as a computation unit (coarse-grain parallelism), and the second one considering each hardware accelerator as a set of multiple cores (fine-grain parallelism). They rely on the BLAS library to exploit this second level. Besides the limited application domain, global configuration parameters are fixed, while it has been shown that it is important to adapt them to the particular thread memory access pattern [14].

MCUDA [13] is a framework to mix CPU and GPU programming. In MCUDA it is mandatory to define kernels for all available devices. No data distribution policy is provided, and the toolkit can not make any assumption about the relative performance of the supported devices. Introducing any of these features would involve a redesign of the framework. Other works [8, 15] try to exploit at the same time CPU and GPU devices, attempting to obtain good load balancing with the help of heuristics. Data structures partition and manipulation is not abstracted and they do not support flexible mechanisms to add new partition and layout policies. Finally, papers as [9, 2] use MPI and CUDA parallel programming model in order to exploit all GPUs devices in heterogeneous systems. However, the authors do not abstract the use of both models and the target underlying hardware details.

Chapel [1], a PGAS language, proposes a transparent plug-in system for domain partitions in generic systems. The PGAS approach tries to hide the communication issues to the programmer. Thus, efficient aggregated communications can not be directly expressed, and most of the times can not be automatically derived from generic codes. Contributors to Chapel are currently working in prototyped layouts to generate array allocations, data transfers, and parallel operations in CUDA. However, they do not offer a different layer for accelerator partition policies, or synchronization among different CPU and GPU devices.

3 Conceptual approach

Heterogeneous systems can be built with very different hardware devices (CPU-cores, accelerators) in several nodes interconnected in a distributed environment. Portable codes for such systems should implement parallel algorithms abstracting them from the mapping activities that adapt the computation to the platform. Thus, the programming model should encapsulate the mapping techniques and the CPU/accelerator synchronization with appropriate abstractions.

We propose a programming framework based on: (1) Several layers of plug-in modules that encapsulate the mapping functions; and (2) functionalities to build the coordination (synchronization and communication) structures of the algorithms, which are transparently adapted at run-time in terms of the results of the mapping functions.

Hitmap [3, 5] is a parallel programming library where partition policies are implemented through a set of plug-ins with a common interface. The programmer may select, or change the chosen plug-in in the program code, using only its name. Hitmap automatically associates logical processes to processing nodes. The data-partition plug-in interface returns an object containing information about which parts of the data are mapped to the logical process, taking into account the neighbourhood relationships of a virtual topology. Coordination patterns are built with high-level point-to-point or collective tile communications, using the results stored in the map objects. If partition details change, the communication structure will reflect the changes automatically. Thus, the

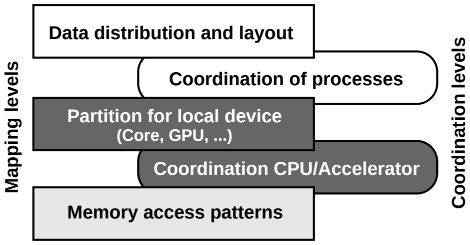
Data distribution and layout
Coordination of processes
Partition for local device
(Core, GPU, ...)
Coordination CPU/Accelerator
Memory access patterns
Mapping levels
Coordination levels

Fig. 1. Mapping/Coordination levels. White boxes show the original Hitmap approach.

coordination among processes may be programmed in an abstract form, independently of the target system topology.

Our work extends the Hitmap approach. Figure 1 shows the different mapping levels of the original Hitmap, and our proposed extension. Hitmap has a single level of data-partition and layout. It is designed to encapsulate coarse-grain mapping techniques, appropriate for distributed-memory nodes.

We propose to add a second, middle-grain partitioning level that allows to adapt the local part of data to the specific characteristics and architecture of the actual device associated to the logical process by the virtual topology. The programmer naturally introduces a third level of mapping inside the kernel code by implementing specific thread-level memory access patterns.

The second-level mapping plug-ins use information about the device and the global memory access pattern of the kernel, to generate domain partitions that exploit locality, maximum occupancy, coalescence, or other device properties that affects performance. The result is an object encapsulating information about a partition of the local computation in a grid of blocks. The same abstraction can be used for techniques of very different nature: CPU-cores, GPUs, or other kind of accelerator. Finally, the coordination, data movement between the CPU and accelerators, and kernel launch, can be automatized by a run-time system, using the second-level partition results. Padding can be automatically added to tiles if needed to properly align data to the memory banks of the particular device, alleviating memory bottlenecks, and improving cache use.

This approach can be used together with techniques to automatically generate kernels for different architectures from common specifications (see e.g. [12, 4]), avoiding the current need to supply optimized kernels for all the architectures that compose the target heterogeneous system, By encapsulating the CPU/accelerator coordination in a transparent system, we also allow to integrate as kernels libraries specifically optimized for a given architecture, such as CUBLASfor GPUs. We also promote the abstraction of hierarchical tiles to specific programming languages for accelerators (in this work we use CUDA as a proof of concept, doing this exercise for more generic languages such as OpenCL is straightforward). Thus, we introduce a common array abstraction, simplifying the porting of code between CPU cores and accelerators.

This conceptual framework has been implemented adding new functionalities to the Hitmap library, without modifying the original structure. This imposes a minimal impact on the original Hitmap codes. The original Hitmap code takes care of the coordination of processes in the higher level. The new extension takes care of adapting the local parts to the device automatically assigned to the logical process. Plug-ins with new mapping techniques may be included and tested without modifying the framework implementation.

4 Design and implementation

We have developed a prototype implementation of this framework extending Hitmap. In this section we describe some design and implementation considerations, and problems solved.

The original Hitmap library was written in C language. Nevertheless, it has and object-oriented design, and future releases could provide a neater C++ interface. Hitmap is designed to manipulate hierarchical tiling arrays. The *HitShape* class implements tile domains. A shape object represents a subspace of array indexes defined as an n-dimensional rectangular parallelotope. Its limits are determined by n *Signature* objects. Each *Signature* is a tuple of three integer numbers (begin, end, and stride), representing the indexes in a domain axis.

Hitmap defines an API for data-partition modules, named *Layout* plug-ins. It defines a wrapper function that links the main code with the chosen plug-in. The Layout plug-ins receive as parameters: (1) a virtual topology object (*HitTopology*), (2) a domain to be mapped (a HitShape object), and (3) optional parameters for the specific technique. They return a *HitLayout* object containing a local domain (another HitShape), information about neighbor relations, and other mapping details. These objects are used as parameters in the constructors of *HitComm* objects that express tile communications across logical processes.

Partitions We follow the same approach for the new second-level *Partition* plug-ins. The wrapper function is similar, but also selects different implementations of the same plug-in name depending on the architecture of the target device. Our current wrapper differences between CPU-cores, and different Nvidia's CUDA supported architectures.

The Hitmap initialization function gathers information about the particular system devices and builds an internal physical topology object. The virtual topology constructors attach each logical process to one device, storing its information. The Partition plug-ins receive as parameters: (1) the attached device data; (2) a HitLayout object with information of the local domain to be mapped. Optional parameters indicating the memory-access patters of the low-level threads can be supplied. The result is a new *HitPartition* containing information about block shapes, grid sizes, and information to generate tile paddings if needed.

As example, we have implemented a trivial partition plug-in. The CPU-cores implementation simply creates a grid with one element containing the full local shape. The GPU implementations split the local domain in rectangular blocks with 1 × 512 threads. This is appropriate for computations that access data linearly in both Nvidia's architectures [14]. For specific grid sizes an extra block is added at the end of each row to alleviate GPU memory contention effects. More sophisticated policies can be integrated as new plug-ins.

Assigning several logical processes to the same device We have introduced a new technique in the virtual topology modules of Hitmap. It allows to assign more than one logical process to the same device. It has two purposes. As a potential load-balancing technique (see section 6), and to transparently use accelerators to perform large computations whose data do not directly fit in the accelerator global memory. Thus, the full computation is done in smaller parts, coordinated by the Hitmap upper-level communication structures.

Kernel definition and launch We provide a macro function to define with a common interface the function headers of different kernel versions for different architectures. The following example shows the headers of two implementations (one for CPU-cores, another for pre-Fermi GPUs) of the same kernel:

```
hit_kernelDefinition( CORE, mmult, HitTile_float *A, HitTile_float *B, HitTile_float *C ) {
hit_kernelDefinition( GPU_R1, mmult, HitTile_float *A, HitTile_float *B, HitTile_float *C ) {
```

We have developed one function that transparently do the coordination with the assigned device. It receives as parameters the kernel name, a partition object, and the kernel parameters, indicating which ones are inputs and outputs. See the example in Fig. 2. This function deals with linking issues of kernels written if specific languages. For example, the launch of a kernel for an Nvidia GPU needs a special syntax and the launching code has to be compiled with the CUDA compiler. We use internal wrapper functions with different implementations for different architectures. Each implementation is compiled with the proper tools before linking. A selection mechanism checks at run-time the assigned device architecture and calls the appropriate implementation for the local device.

For CPU-cores the wrapper simply calls the proper C function passing the indicated arguments. For accelerators the process is more complex, and involves communication between the main system memory and the device memory. We have implemented the synchronization with Nvidia's GPUs with the following stages: (1) Move to the GPU memory the input tiles (the data and the tile handler structure). Padding restrictions expressed in the HitPartition object are applied to the memory allocation in this step. (2) Launch the kernel, using the grid parameters from the Partition object, passing the pointers to the new tile handlers in GPU memory. (3) Copy data from output GPU-memory tiles to the CPU, eliminating padding if needed. Finally, a mutual exclusion mechanisms has been added in the kernel launch function to allow several processes assigned to the same device to coordinate themselves for the use of the device.

These abstractions completely encapsulate the synchronization and coordination between CPU and different devices, such as cores and accelerators. The same primitive call automatically invokes CPU-core functions written in plain C language, or launches CUDA kernels.

Running the programs Hitmap programs are started like any MPI program, using the *mpiexec* command. The MPI hosts file is used to select the machines where the processes are started. Processes in the same machine are automatically attached to CPU-cores or GPU devices. If data do not fit into the memory of an accelerator device, more MPI processes are required to obtain a finer partition.

5 Case study

In this section we show with an example how Hitmap abstractions lead to codes which are independent of the encapsulated mapping techniques. We have chosen the Cannon's algorithm for matrix multiplication (see e.g. [7]). It is a task-parallel algorithm focused on reducing local memory usage for distributed systems. Thus, it shows the interaction of different levels of parallelism.

In Cannon's algorithm the available processes are organized in a perfect square topology to generate neighbor relations. Each matrix A, B and C is divided into rectangular blocks, distributing them across processes. It starts with an initial communication stage to relocate A and B blocks in a circular shift ($A_{ij} = A_{i(j-i)}$, and $B_{ij} = B_{(i-j)j}$). On each step, every process multiply its local blocks of A and B, accumulating the partial results in the local block of C. It then sends the used block of A to the leftward process, and the used block of B to the upward process, both in a circular shift. There are as many communication-computation steps as the square-root of the number of total processes.

Figure 2 shows the Cannon's matrix multiplication algorithm implemented with the Hitmap library for heterogeneous systems. We use float base elements. The code is the same used in previous versions of Hitmap for distributed-memory systems except lines 40–41 (that encapsulate the low-level partition for the assigned device), and lines 47 and 50, that encapsulates the coordination between the CPU and the accelerators.

Lines 3–6 declare the full domain of the three matrices with a global-view approach. Memory is not yet allocated. Line 9 builds a virtual topology enforcing a perfect square of processes, as required by the algorithm. Lines 12–14 create layout objects that distribute the matrices domains across the virtual topology. The layout plug-in modules used are different for the three matrices. Figure 3 shows a diagram of the resulting layouts. Matrix B uses a classical block data partition, with evenly sized parts. Matrices C and A use a load-balancing plug-in technique. The rows dimension is split unevenly according to a *Balance factor*, decided in terms of the relative computing power of the device types as recorded in the low-level topology description. Currently, it is experimentally determined.

In lines 17–21 each logical process creates and allocates the local part of the matrices. Thanks to the *maxShape* padding function, n and m do not need to be exact multiples of the number of processes in a given axis. Lines 24–26 read in parallel the tiles of the input matrices. The C matrix is initialized with 0 values.

Lines 29–32 perform the initial relocating stage prescribed by the Cannon's algorithm, shifting A and B tiles. Lines 35–37 build the shifting communication pattern that will be used between the computation stages. The layout objects

```
void cannonsMM( int n, int m, int p ) {
    /* 1. DECLARE FULL MATRICES WITHOUT MEMORY */
    HitTile_double A, B, C;
    hit_tileDomain( &A, float, 2, n, m );
    hit_tileDomain( &B, float, 2, m, p );
    hit_tileDomain( &C, float, 2, n, p );

    /* 2. CREATE VIRTUAL TOPOLOGY */
    HitTopology topo = hit_topology( plug_topSquare );

    /* 3. COMPUTE PARTITIONS */
    HitLayout layC = hit_layout( plug_layBlocksLB, topo, C, 0 );
    HitLayout layA = hit_layoutWrap( plug_layBlocksLB, topo, A, 0 );
    HitLayout layB = hit_layoutWrap( plug_layBlocks, topo, B );

    /* 4. CREATE AND ALLOCATE TILES */
    HitTile_double tileA, tileB, tileC;
    hit_tileSelectNoBoundary( &tileA, &A, hit_layMaxShape(layA,1) );
    hit_tileSelectNoBoundary( &tileB, &B, hit_layMaxShape(layB,0) );
    hit_tileSelect( &tileC, &C, hit_layShape(layC) );
    hit_tileAlloc( &tileA ); hit_tileAlloc( &tileB ); hit_tileAlloc( &tileC );

    /* 5. INITIALIZE MATRICES */
    hit_tileFileRead( &tileA, "matrixA.dat" );
    hit_tileFileRead( &tileB, "matrixB.dat" );
    float aux=0; hit_tileFill( &tileC, &aux );

    /* 6. INITIAL ALIGNMENT PHASE */
    HitComm commRow = hit_comShiftDim( layA, 1, -hit_layRank(layA,0), &tileA );
    HitComm commCol = hit_comShiftDim( layB, 0, -hit_layRank(layB,1), &tileB );
    hit_comDo( commRow ); hit_comDo( commCol );
    hit_comFree( commRow ); hit_comFree( commCol );

    /* 7. REUSABLE COMM PATTERN */
    HitPattern shift = hit_pattern( HIT_PAT_UNORDERED );
    hit_patternAdd( &shift, hit_comShiftDim( layA, 1, 1, &tileA ) );
    hit_patternAdd( &shift, hit_comShiftDim( layB, 0, 1, &tileB ) );

    /* 8. COMPUTE DEVICE PARTITION USING ACCESS PATTERN INFO */
    HitPartition parts = hit_partition( plug_partBlocks, hit_layShape(layC),
                   2, hit_shape( 2, ALL, THIS ), hit_shape( 2, THIS, ALL ) );

    /* 9. DO COMPUTATION */
    int loopIndex;
    int loopLimit = max( hit_layNumActives(layA,0), hit_layNumActives(layB,1) );
    for (loopIndex = 0; loopIndex < loopLimit-1; loopIndex++) {
        hit_kernelLaunch( mmult, parts, 3, IN, tileA, IN, tileB, INOUT, tileC );
        hit_patternDo( shift );
    }
    hit_kernelLaunch( mmult, parts, 3, IN, tileA, IN, tileB, INOUT, tileC );

    /* 11. WRITE RESULT */
    hit_tileFileWrite( &tileC, "matrixC.dat" );

    /* 12. FREE RESOURCES */
    hit_partitionFree( parts );
    hit_layFree( layA ); hit_layFree( layB ); hit_layFree( layC );
    hit_patternFree( &shift );
    hit_topFree( topo );
}
```

Fig. 2. Heterogeneous Hitmap implementation of Cannon's matrix multiplication

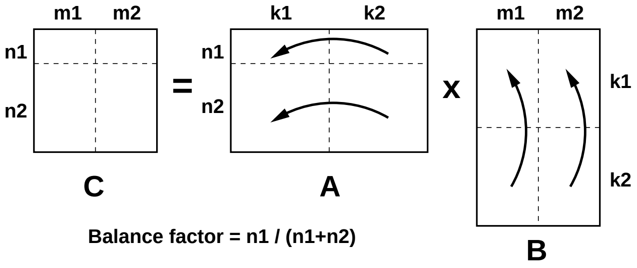

Fig. 3. Load balancing layout scheme in the Cannon's matrix multiplication example.

and the tiles provide all the information needed to internally find neighbors and build MPI derived data types to optimize the communications. Thus, communications are adapted to the partition transparently. For this example we choose synchronous communication to avoid the need of double buffers, exploiting our full system memory to do larger computations.

Lines 40–41 generate a partition object tailored to the device assigned to the logical process. Line 41 is a shape expression that represents the global memory access pattern; indicating, in relative coordinates, which elements are accessed by a thread. Lines 44–50 implement the main loop of the algorithm. The computation stage of the last iteration has been unrolled to avoid the last unneeded communication stage. The computation is launched by the *hit_kernelLaunch* primitive, independently of the actual device. The shifting communication pattern is activated by the *hit_patterDo* primitive. Line 53 writes the output matrix tiles to a file in parallel. Lines 56–59 free all the Hitmap resources before finishing.

6 Experimental work

We have designed experimental work to show that: (1) Our new abstractions do not impose a significant overhead on the computation; and (2) this framework allows to easily exploit different devices to obtain performance benefits.

In order to show the efficiency of the Hitmap codes, we have manually developed and optimized reference codes for matrix multiplication: (a) A direct MPI implementation of the Cannon's algorithm (see [7]); and (b) a direct CUDA implementation that may also split and multiply the matrices block by block if they are too big to fit in the GPU device memory.

For our experiments we have used two different platforms. The first one is a Beowulf cluster with up to 18 dual-core PC computers. The second one is an Intel(R) Core(TM) i7 CPU 960, 3.20GHz with active hyper-threading. This system has two GPUs: a GeForce 8500 GT, and a GerForce 9600 GT, both managed by the CUDA driver included in the 4.0 toolkit. From now on, we identify the different available devices in this machine with the following letters: (A) GeForce 9600 GT; (B) GeForce 8500 GT; (C) Cores of the CPU.

We select several matrix sizes: $N = M = 2048, 8192, 12288$. The first size is small enough to allocate the three matrices in any of the devices of both systems.

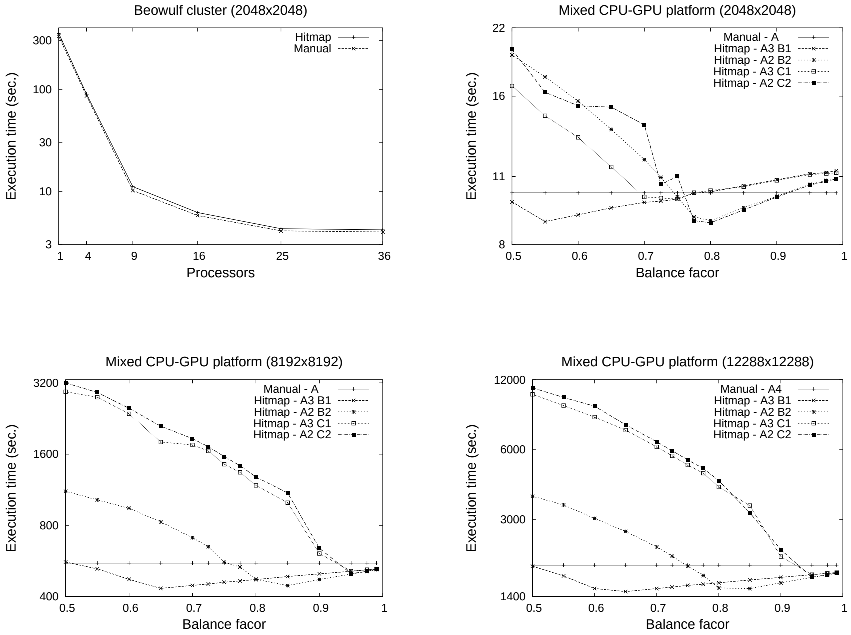

Fig. 4. Experimental work results

The second size cannot be fully allocated on the second GPU (device B) of the mixed CPU-GPU machine. The last size cannot be allocated in any of the GPUs. We also test using modified sizes (e.g. $N = M = 2039, 2057$), that our padding mechanisms do not impose a significant performance effect on the results.

Figure 4 presents execution times obtained in different scenarios. Notice that all y-axis are in logarithmic scale. The experiments in the Beowulf cluster show, even for the small matrix size, that Hitmap implementations have the same scalability and overall performance than the manually optimized MPI code. A minimal Hitmap performance overhead is observed in all our experimental work.

In the more heterogeneous machine the best performance results are obtained for a small number of processes. Remind that Cannon's algorithm forces more synchronization stages when the number of processes increase. Thus, for Cannon's algorithm, more MPI processes lead to bigger communication overhead, while reducing the computation load of each task. In this machine, our experiments show the best results for four MPI processes. We show results for the following scenarios. *Reference code:* (A) Manually developed CUDA code, executing the whole computation with only one kernel launch in device A, the fastest GPU; (A4) For matrices that do not fit in the GPU device memory, the reference code parting the matrices in four even parts and executing the computation in several kernel launches. *Hitmap code:* Changing the topology module we can easily experiment with different assignments of devices to logical processes. (A3-B1) Mixed GPUs: 3 processes mapped to device A, and 1 process to device B; (A2-B2) Mixed GPUs: 2 processes mapped to device A, and 2 process to device B; (A3-C1) GPU and core: 3 processes mapped to device A, and 1 process to one CPU-core; (A2-C2) GPU and cores: 2 processes mapped to device A, and 2 processes, each one mapped to a different CPU-core. For all the experiments with GPUs and Hitmap we have manipulated the partition plug-in to experiment with different load-balance factors, between 0.5 and 0.975.

Consider the execution time of the reference CUDA code (A and A4). The results show that it is always possible to improve these performance results with the Hitmap code, exploiting heterogeneity with more than one device. The results for the small matrix size are more unstable, and impacted by the kernel initialization times, including the communication between CPU and GPU. However, as the matrix size increases, the results are more stable, and show exactly the same trends. We obtain performance improvements of up to 10% for the small matrices, and a consistent best improvement of 20.5% for medium and big input data sizes. Traces of the executions show that the MPI communication times are always less than 10% of the total execution time for the small matrix size. And their impact quickly decreases as the data input size grows.

On the left part of the plots (load-balance factor 0.5), the load is evenly distributed, not taking into account the different computing powers of the devices. The critical path is dominated by the slower devices. As the load-balance factor grows, the balance is improved proportionally reducing the total execution time. After the optimum balance point is found, an increase of the factor leads to too few computation on the slower devices. Thus, the critical path is dominated by the fastest device, proportionally reducing performance again.

An important question is: Is it possible to predict the best load-balance factor for a given set of devices? Profiling tests with simple benchmarks show that the relative computing power between devices A and B is approximately $r = 3.826$; and between device A and a core (device C) it is $r = 14.153$. In order to assign to each device a computation proportional to its relative computing power, the load-balance factor may be calculated as $LB = r/(r+1)$ for the A2 scenarios, and $LB = (r-1)/(r+1)$ for the A3 scenarios. The experimental results show that, for big enough matrices, this estimation is always a little lower than the value that leads to the best performance: 10% in both A3 scenarios, 2% and 6% on A2 scenarios. A more sophisticated model, taking into account the synchronization stages, is needed to automatically predict the best factor in the Layout plug-ins.

7 Conclusion

In this paper we present a new framework for heterogeneous programming. It encapsulates the mapping techniques into plug-ins at two different layers of abstraction: one related to logical processes coordination, and another related to adapting the computations to the inherent parallelism and architecture details of the actual device associated to each logical process. We propose a high-level

API that transparently deals with all the details of communication and synchronization between logical processes and accelerator devices, such as GPUs. This framework allows to generate codes which are transparently adapted to heterogeneous systems with mixed types of accelerator devices.

Current on-going work involves: (1) Introducing in the framework more sophisticated mapping policies that better exploit CPU-cores and GPU architecture information; and (2) test the applicability of these techniques to more types of programs, including well-know benchmarks and real applications.

References

1. Chamberlain, B., Deitz, S., Iten, D., Choi, S.E.: User-defined distributions and layouts in Chapel: Philosophy and framework. In: 2nd USENIX Workshop on Hot Topics in Parallelism (June 2010)
2. kui Chen, Q., kang Zhang, J.: A stream processor cluster architecture model with the hybrid technology of mpi and cuda. In: ICISE'2009. pp. 86 –89 (dec 2009)
3. de Blas Cartón, C., Gonzalez-Escribano, A., Llanos, D.R.: Effortless and Efficient Distributed Data-Partitioning in Linear Algebra. In: HPCC'2011. pp. 89–97. IEEE (Sep 2010)
4. Farooqui, N., Kerr, A., Diamos, G.F., Yalamanchili, S., Schwan, K.: A framework for dynamically instrumenting GPU compute applications within GPU Ocelot. In: GPGPU. p. 9 (2011)
5. Fresno, J., Gonzalez-Escribano, A., Llanos, D.R.: Automatic Data Partitioning Applied to Multigrid PDE Solvers. In: PDP'2011. pp. 239–246. IEEE (Feb 2011)
6. Gelado, I., Stone, J.E., Cabezas, J., Patel, S., Navarro, N., mei, W.: An asymmetric distributed shared memory model for heterogeneous parallel systems. In: ASPLOS'2010. pp. 347–358. ACM, New York, NY, USA (2010)
7. Grama, A., Gupta, A., Karypis, G., Kumar, V.: Introduction to Parallel Computing. Addison Wesley, 2nd edn. (2003)
8. Hong, C., Chen, D., Chen, W., Zheng, W., Lin, H.: MapCG: writing parallel program portable between CPU and GPU. In: PACT'2010. pp. 217–226. PACT '10, ACM, New York, NY, USA (2010)
9. Karunadasa, N., Ranasinghe, D.: Accelerating high performance applications with cuda and mpi. In: ICIIS'2009. pp. 331 –336 (dec 2009)
10. Luk, C.K., Hong, S., Kim, H.: Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In: MICRO-42. pp. 45 –55 (dec 2009)
11. Quintana-Ortí, G., Igual, F.D., Quintana-Ortí, E.S., van de Geijn, R.A.: Solving dense linear systems on platforms with multiple hardware accelerators. In: PPoPP'2009. pp. 121–130. PPoPP '09, ACM, New York, NY, USA (2009)
12. Singh, S.: Computing without processors. Commun. ACM 54, 46–54 (August 2011)
13. Stratton, J.A., Stone, S.S., Hwu, W.M.W.: Mcuda: An efficient implementation of cuda kernels for multi-core cpus. In: Amaral, J.N. (ed.) LCPC'2008. pp. 16–30. Springer-Verlag, Berlin, Heidelberg (2008)
14. Torres, Y., Gonzalez-Escribano, A., Llanos, D.R.: Using Fermi architecture knowledge to speed up CUDA and OpenCL programs. In: Proc. ISPA'12. Leganes, Madrid, Spain (2012)
15. Yao, P., An, H., Xu, M., Liu, G., Li, X., Wang, Y., Han, W.: CuHMMer: A load-balanced CPU-GPU cooperative bioinformatics application. In: HPCS'2010. pp. 24 –30 (July 2010)