TuCCompi: A Multi-Layer Programing Model for Heterogeneous Systems with Auto-Tuning Capabilities

Hector Ortega-Arranz  Yuri Torres  Diego R. Llanos  Arturo Gonzalez-Escribano
Departamento de Informática, Universidad de Valladolid, Spain
{hector | yuri.torres | diego | arturo}@infor.uva.es

Abstract
During the last decade, parallel processor architectures have become a powerful tool to deal with massively-parallel problems that require High Performance Computing (HPC). The last trend of HPC is the use of heterogeneous environments, that combine different computational power units, such as CPU-cores and GPUs. Performance maximization of any GPU parallel implementation of an algorithm requires an in-depth knowledge about its underlying architecture, becoming a tedious task only suited for experienced programmers. In this paper we present TuCCompi, a multi-layer framework that not only transparently exploits heterogeneous systems, but automatically tunes the GPU capabilities by choosing the optimal values for their configuration parameters, using the kernel characterization provided by the programmer. This model is very useful to tackle problems characterized by independent, high computational-load tasks with none or few communications, such as embarrassingly-parallel problems. We have evaluated TuCCompi in different, real-world heterogeneous environments using the APSP problem as a case study.

Categories and Subject Descriptors D. Software [Programming techniques]: Concurrent Programming

Keywords APSP, Auto-Tuning, CUDA, GPU, Heterogeneous system, HPC framework, MPI, OpenMP, Parallel model

1. Introduction
During the last decade, parallel processor architectures have become a powerful tool to handle massively-parallel problems. These computing-intensive problems are divided into many independent tasks that can be executed in parallel, and that do not require any communication among them. They are called embarrassingly-parallel problems [7]. Many real problems are included in this category, such as index processing in web search [8], bag-of-tasks applications [3], traffic simulations [20] or some molecular physics computations [2].

Although the parallelization of embarrassingly-parallel problems does not require a very complex algorithm to take profit of parallel computing environments, their high amount of computational work requires High Performance Computing (HPC). In order to give support to the massive demand of HPC, the last trends focus on the use of heterogeneous environments that include computational units of different nature. These computational units include common CPU-cores, graphic processor units (GPUs) and other hardware accelerators. The exploitation of these environments offers a higher peak performance and a better efficiency compared to the traditional homogeneous cluster systems [1]. Due to these advantages and to the low cost of building heterogeneous systems, they are being incorporated into many different computational environments, from small academic research clusters, to supercomputing centers.

Despite of the wide use of heterogeneous environments to execute massively-parallel algorithms, there are two issues that limit the usability of these environments. The first one is the lack of global computing frameworks that easily schedules the workload in such complex environments. Some works have tried to ease the joint use of parallel programming languages, such as MPI or OpenMP, by the creation of different tools. For example, a source-to-source compiler that translates C annotated code to MPI + OpenMP or CUDA code is presented in [18]. However, in this work CUDA can not be jointly used with the other parallel models. Another example is OMPICUDA [12], a framework to develop parallel applications on heterogeneous clusters by mixing OpenMP and MPI. In this work OpenMP code is translated to CUDA, however, this code has serious programming limitations. Moreover, these works do not exploit all computational capabilities of the GPUs. There is not a known parallel model that automatically selects the optimal values for CUDA configuration parameters, such as the threadBlock size-shape or the state of L1 cache memory, of each kernel. These optimization techniques significantly enhance the GPU powerful performance.

The second limitation is the lack of a tuning methodology that efficiently unleashes the power of GPU devices. Although languages such as CUDA aim to reduce the programmer’s burden in writing parallel applications, it is a difficult task to correctly tune the code in order to efficiently exploit all underlying GPU resources. Some configuration parameters, such as the thread-block size and shape, and cache L1 size, have a significant impact on the GPU performance. Several studies have shown that in some cases the values recommended by CUDA do not lead to the optimum performance, leaving to programmers the task of searching for the best values through time-consuming, trial-and-error tests.

In this paper we present TuCCompi (Tuned, Concurrent CUDA, OpenMP and MPI), a multi-layer computing framework that transparently exploits heterogeneous systems and squeezes the GPU capabilities by automatically choosing the optimal values for the configuration parameters. Each layer represents a level of parallelism. The first layer handles the distributed-memory environment coordinating the nodes (so-called shared-memory systems) that compose...
Communication and synchronization mechanisms in order to coordinate these machines for the parallel resolution of the problem. The first layer of TuCCompi (see Fig. 1) is responsible of managing the coordination of these nodes without taking into account the specific hardware details and features of each machine. In order to communicate and synchronize these nodes, we use MPI (Message Passing Interface) as message passing tool.

The 2nd layer (shared-memory systems) Most computers nowadays are composed by several processing units (we will name them CPU-cores) that share a global address space. Additionally, there are other devices, such as GPUs, FPGAs and Xeon Phi among others, that are also able to perform computational actions at the same time but usually they have to be governed by a CPU-core. Although the use of these devices implies the computational sacrifice of a CPU-core, their performance is higher than the one obtained by this CPU-core. In this layer of TuCCompi we use the concept of "computational unit" for any CPU-core or device that shares the global memory hosted in a node. This second layer is responsible of the coordination of all computational units inside the node. If there were devices in the machine, like the GPUs present in Fig. 1, this layer would automatically deploy the parallel version of the algorithm to the CPU-cores responsible of the device management, and the sequential version to the rest of CPU-cores. In order to manage these resources we use OpenMP as thread-management environment.

The 3rd layer (GPU devices) An emerging way of parallel computing includes the use of hardware accelerators, such as GPUs. Their powerful capability have triggered their massive use to speed up high-level parallel computations. For certain problems, the use of a parallel implementation of an algorithm in these hardware accelerators can offer huge speedups against the sequential algorithms deployed in the CPU-cores. However, their management is much more complicated than any multi-core system. If these kind of devices are found in a shared-memory system, the third layer automatically deploys the parallel-algorithm execution into them additionally to the sequential execution of the remaining node CPU-cores. In order to manage these devices, we use CUDA as programming environment.

The 4th layer (concurrent GPU kernel execution) The most recently NVIDIA GPUs support concurrent-kernel execution [14, 15], where different kernels of the same application context can be executed on the GPU at the same time (See Fig. 2). If the number of resources needed to execute a kernel does not reach the total available resources, the remaining ones can be used to concurrently execute another kernel. Thus, the number of kernels that can be executed at the same time, depends on the total hardware resources required by each kernel and the corresponding GPU hardware characteristics. This feature is very helpful when small kernels are launched, allowing a concurrent execution that exploits all device resources. Although at first sight this feature seems to be profitable only when small kernels are launched, the concurrent execution for bigger kernels also gives performance improvements. This occurs because the launch of several kernels of the same application context takes advantages of the L1 data-cache, originating less number of cache-misses and therefore alleviating the global memory bottlenecks. Additionally, the threadBlock-warp dispatcher works faster scheduling kernels if they have been previously launched [16].

The fourth layer of TuCCompi (see Fig. 1) is responsible of the automatic launching of many concurrent kernels in modern GPUs, squeezing their computational resources. The different tasks that are scheduled to these kind of accelerators can be executed in parallel in the same device, adding a new level of parallelism.

The Tuning layer. While correctness of an NVIDIA CUDA program is easy to achieve, the optimal exploitation of the GPU computational capabilities is much more complicated than traditional

Figure 1. Layer deployment of TuCCompi model in a heterogeneous cluster.
3. TuCCompi Model Usage

TuCCompi users should provide to the model some elements for its use (see Fig. 3): (a) the sequential-CPU and the parallel-GPU code application, that we define as CPU\_PLUG-IN and GPU\_PLUG-IN respectively, (b) the kernel characterizations used in the GPU code, and (c) the main program with TuCCompi macros.

3.1 Kernel characterization

The user has to provide a general characterization of his kernels in its definition. This information is easily expressed in our prototype implementation through the TuCCompi\_KERNELCHAR(kernel, num\_dim, A, B, C, D) primitive. The values for parameters A, B, C and D have to be chosen from the kernel-characterization classification shown in Table 1. TuCCompi model automatically optimizes the use of all underlying hardware resources of GPU devices, following the guidelines and optimizations proposed in [21] for each of the possible combination of these parameters.

Table 1. TuCCompi kernel-characterization classification. The def choice can be used when the user does not know the kernel characterization.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Global memory-access pattern</td>
<td>scatter/medium-coalesced/medium-coalesced/def</td>
</tr>
<tr>
<td>B</td>
<td>Ratio of arithmetic instructions per thread compared to the global-memory accesses</td>
<td>high/low/none/def</td>
</tr>
<tr>
<td>C</td>
<td>Ratio of L1 cache memory lines evictions compared to the size of this memory</td>
<td>high/medium/low/def</td>
</tr>
<tr>
<td>D</td>
<td>Ratio of global memory data reutilization compared to the number of arithmetic instruction per thread</td>
<td>high/medium/low/def</td>
</tr>
</tbody>
</table>

The def choice can be used whenever the user does not know the kernel characterization.

3.2 User-code Plug-ins

Figure 5 shows the interface of the sequential code that will be executed in a CPU computational unit. The user is responsible of

![Figure 5. Plugin_cpu interface. The programmer adds to his code the boxed arguments to deploy the Cpu plugin in TuCCompi.](image-url)
the algorithm implementation that solves a single task (line C01, Cpu user code).

Figure 6 shows the code that will be executed in a CPU to manage the associated GPU. The user should define the code that handles the logic control of the algorithm that comprises the use of one or several GPU kernels. This code will be the responsible of launching the corresponding kernels into the GPU. Line G02 shows the TuCComp macro that carries out a kernel launch, with the name of the kernel as first parameter, and followed by other user variables that have been previously allocated in the GPU. Transpar-

tently for the user, the model executes as many kernel instances as the hardware nature of the computational unit. 

The needed parameters are: The variable name; the type of element; and the number of elements that compounds the MK value defined in the fourth layer by the user (see line M03 of Fig. 7). Every concurrent kernel launched will need the use of its own workspace to compute its results. The macro of line G03 gives to the kernel one memory pointer for each data structure needed. The needed parameters are: The variable name; the type of element; and the number of elements that compounds the MK value defined in the fourth layer by the user (see line M03 of Fig. 7). Every concurrent kernel launched will need the use of its own workspace to compute its results. The macro of line G03 gives to the kernel one memory pointer for each data structure needed. The needed parameters are: The variable name; the type of element; and the number of elements that compounds

Figure 7. User implementation of the TuCComp macro instance. In this section we will discuss the internals of the TuCComp framework.

4.1 Cluster Node Communication: TuCComp_COMM

Once a heterogeneous cluster is defined and the TuCComp program is in execution, each process initializes its MPI-identification variables, and enters in a global communication step carried out by exchanging a few MPI messages. The parent process adopts a listening posture, receiving from the remaining processes the number of the computational resources they are able to manage. Afterwards, in order to have a global identifier for each computational unit, the parent process sends each process an identification number for each resource, avoiding conflicts with other computational units. Additionally, the parent process sends more information about the whole heterogeneous environment, such as the number of resources of each node and the particular numeration of each computational unit, among others.

Fig. 8 shows the implementation of this first phase. We will now review the data structures involved. The \( v_{\text{cu}} \) vector stores the number of computational units from each process. The \( v_{\text{ld}} \) vector stores the number from which the numeration of computational units should start for the process \( i \). The \( t_{\text{total cu}} \) variable stores the total number of computational units. The \( t_{\text{mpi}} \) variable stores the identifier of the MPI process. The \( n_{\text{proc}} \) variable stores the total number of MPI processes. Finally, the \( \text{PARENT} \) constant is the identifier of the MPI process that coordinates the communication, whose value is zero. In this first phase, lines 02-04 initialize some values and ask to the second layer how many computational units has the machine. Lines 05-09 receive information from the rest of

3.3 TuCComp main program implementation

Figure 7 shows an example of the code that the user has to implement in order to start the execution of our model. The macro TuCComp_COMM in Line M01 initializes the system. Afterwards, the user can introduce his code, including variable declarations, initializations and the sequential code needed for application. Line M03 shows the macro that the user should use to set the number of tasks that the GPU devices have to concurrently execute. Line M04 shows the macro used to initialize and execute the work-task execution of the functions described in the corresponding plug-ins, for CPUs and GPUs, using all existing computational units in parallel. The first parameter of this macro represents the kind of scheduling policy desired by the user (see Sect. 3.4). In line M05 the programer specifies that the process has to wait until all computational units of the cluster node have finished. The user is free to insert more code with the aim to execute other parallelization instance, if needed, before the finalization of the heterogeneous environment communication shown in Line M07.

3.4 Workload scheduling

TuCComp model includes three different policies to distribute the computational load between all available cluster resources through the M04 primitive. The first parameter allows to choose between the following three different policies.

The first one, EQ1, is an equitable policy that schedules the same number of tasks to each node, of the 1st layer, of the heterogeneous environment. Later, each process equally divides its assigned workload between all its own computational units also in a balanced way.

The second one, EQ2, is also an equitable policy but in this case it schedules the same number of task to each computational unit of the 2nd layer. The workspace division does not take into account the hardware nature of the computational unit.

The third one, MS, follows a master-slave model. One computational unit is sacrificed to act as the master, and the rest of the computational units work as slaves. The slaves enter into a working loop, requesting tasks to the master until it sends a termination signal to them. As the master can be located in any cluster node, these asking-for-tasks requests are issued through distributed-environment communications.

4. TuCComp internals

In this section we will discuss the internals of the TuCComp framework.

4.1 Cluster Node Communication: TuCComp_COMM

Once a heterogeneous cluster is defined and the TuCComp program is in execution, each process initializes its MPI-identification variables, and enters in a global communication step carried out by exchanging a few MPI messages. The parent process adopts a listening posture, receiving from the remaining processes the number of the computational resources they are able to manage. Afterwards, in order to have a global identifier for each computational unit, the parent process sends each process an identification number for each resource, avoiding conflicts with other computational units. Additionally, the parent process sends more information about the whole heterogeneous environment, such as the number of resources of each node and the particular numeration of each computational unit, among others.

Fig. 8 shows the implementation of this first phase. We will now review the data structures involved. The \( v_{\text{cu}} \) vector stores the number of computational units from each process. The \( v_{\text{ld}} \) vector stores the number from which the numeration of computational units should start for the process \( i \). The \( t_{\text{total cu}} \) variable stores the total number of computational units. The \( t_{\text{mpi}} \) variable stores the identifier of the MPI process. The \( n_{\text{proc}} \) variable stores the total number of MPI processes. Finally, the \( \text{PARENT} \) constant is the identifier of the MPI process that coordinates the communication, whose value is zero. In this first phase, lines 02-04 initialize some values and ask to the second layer how many computational units has the machine. Lines 05-09 receive information from the rest of
4.2 Parallel Computation: TuCCompi\_PARALLEL

Once the TuCCompi model has been initialized and the user variables have been defined, this primitive automatically creates as many OpenMP threads as the number of CPU-cores that will perform the parallel execution. Figure 9 shows the code that is executed when the programmer uses the TuCCompi\_PARALLEL primitive for the master-slave scheduling policy. (EQ1 and EQ2 policies are shown due to space restrictions.) The master-slave implementation just divides the workload between the cluster nodes and the computational units, and execute the task without any model computation just divides the workload between the cluster nodes and the master, executing the default master function. If there are GPUs, each one will be governed by its corresponding CPU-core. Therefore, lines 10-15 first obtain the device properties, entering into the ask-for-tasks working loop, executing the parallel GPU code provided in the pluginGPU. The normal CPU-cores also enter into the ask-for-tasks working loop but executing the code of pluginCPU (lines 16-18). Finally, line 19 indicates the end of the parallel OpenMP region.

4.3 Kernel Launch and MultiKernel: TuCCompi\_GUALUNAHC

Before the parallel divergence, the layer-1, distributed-memory process consults how many GPUs are available in the shared-memory node (Line 01 of Fig. 9). Once in the parallel region, a OpenMP thread is assigned to each CPU-core in order to govern the hardware accelerator, also storing some relevant properties of the GPU, such as its architecture. (Lines 11-13 of Fig. 9). Afterwards, this thread is the responsible of handling the logic control of the algorithm implemented in pluginGPU, launching different kernels through the primitive TuCCompi\_GUALUNAHC(kernel\_name, input\_size, kernel\_vars) whose definition is shown in Fig. 10.

The model automatically detects if the concurrently execution of several kernels (the multikernel feature) is supported by the GPUs using the properties previously retrieved. Otherwise, the model always launches only one kernel at the same time. The multikernel feature is also embedded in the GPU launching primitive (Line 01 of Fig. 10). Additionally, in order to make possible that each kernels computes in its corresponding workspace, the PARALLMK(variable\_name, variable\_type, variable\_length) macro automatically makes the memory offset allocation of the corresponding variables that are task-dependent (Lines 04-05 of Fig. 10).

4.4 Automatic Kernel Tuning: TuCCompi\_KERNELCHAR

The optimization layer automatically configures the kernel parameters depending on: (1) the GPU architecture where is going to be launched, and (2) the kernel characteristics provided by the user.

In order to obtain the optimal values in terms of kernel features, we have followed the guidelines proposed in [21]. These authors have designed and implemented a suite of micro-benchmarks, called uBench, in order to evaluate how different threadBlock sizes and shapes affect the performance for each GPU architecture (Fermi and Kepler). They have characterized and classified a wide range of kernel types, also presenting the optimal configurations for them.

The kernel classification that we have implemented was previously described in Sect. 3.1. As long as the model recognizes the architecture of the GPUs that are present in each cluster node, it only needs to know the characterization of each user-defined kernel. This characterization is indicated by the programmer before the kernel definition (See Fig. 4), and automatically mapped.
to an structure that contains the optimal values for all architectures (See Fig. 11). As can be seen in lines 02-03 of Fig. 10, these values are already embedded in the primitive of kernel launching as t.grid(), that returns the optimal number of blocks, and t.threads(), that returns the optimal number of threads per block. Then, our model automatically selects the optimal configuration of the threadBlock size-shape.

If the user does not know how to characterize his/her kernel, the default values can be used. These values are those recommended by CUDA [11], maximizing the SM Occupancy. Although These recommended values sometimes work well, we will see that there could be performance differences of more than ten percent depending on the use of different, recommended CUDA configurations for a particular kernel.

4.5 Advanced TuCCompi Model Features
TuCCompi model has additional functionalities and features, such as the possibility of executing a complex workload scheduling created by the user, or the possibility of changing the optimal values for each kernel and GPU. We will now describe two plugins that help with these tasks.

4.5.1 Scheduling plugin
The master and the slaves execute, respectively, the master-function and slave-function code provided in the distribution plug-in. The model gives a simple implementation for both, where only one task is scheduled to each slave independently of its computational power. Additionally, if the problem or the user needs a particular load distribution that follows a special pattern or policy, the model allows to the programmer to use his/her own master-slave implementation injecting it through the scheduling plug-in, using an extended primitive TuCCompi_PARALLEL(MS, pluginGPU, pluginMASTER, pluginSLAVE).

That is very useful if the user has in the heterogeneous environment some devices that works very fast compared with the rest. In this case, it may be a good choice that the master gives them a pack of tasks instead a single one. When a OpenMP thread responsible of a GPU device asks for tasks, it is able to retrieve the corresponding device information that could be send to the master in the requesting message. With this information, the master can produce a more complex distribution depending on the capabilities of the computational units that are asking for work. In this way, the master could give a pack of tasks to the most powerful devices and a single one to the less powerful computational units. Figure 12 shows a customized implementation of the scheduling plugin created for the case study.

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

5. Case study
In order to check the developed TuCCompi framework, we have chosen the APSP problem for sparse graphs as our case study because it gathers good characteristics to evaluate the model features. Being an embarrassingly parallel problem, it suits perfectly with TuCCompi philosophy for the first three layers. Additionally, the GPU solution for this problems involves three kernels of very different nature, size and characterization. This variety allows us to check the behaviour of the fourth layer and the tuning layer.

In this section we explain this problem more in detail and we describe the corresponding plug-ins developed for the TuCCompi model.

5.1 All-Pair Shortest-Path (APSP) problem
The APSP problem is a well-known problem in graph theory whose objective is to find the shortest paths between any pair of nodes. Given a graph \( G = (V, E) \) and a function \( w(e) : e \in E \) that associates a weight to the edges of the graph, it consists in computing the shortest paths for all pair of nodes \( (u, v) : u, v \in V \).

The APSP problem is a generalization of the classical problem of optimization, the Single-Source Shortest-Path (SSSP), that consists in computing the shortest paths from just one source node \( s \) to every node \( v \in V \).

An efficient solution for the APSP problem in sparse graphs is to execute a SSSP algorithm \( |V| \) times selecting a different node as source in each iteration. The classical algorithm that solves the SSSP problem is due to Dijkstra [5]. Crauser et al. in [4] introduces an enhancement that tries in each iteration \( i \) to augment the threshold \( \Delta_i \) as more as possible to process more nodes in the next iteration.

5.2 Plug-ins
Both sequential and parallel GPU codes are implementations of the Crauser algorithm. Their implementation for this problem has been taken from [17]. Algorithm 1 shows the GPU parallel pseudo-code of Crauser’s algorithm. Figure 13 shows the TuCCompi implementation for the pluginGPU. This implementation repeatedly launches three kernels (relax, minimum and update) with different features. Following the classification described in Sect. 3.1, the kernels are characterized in Table 2.

<table>
<thead>
<tr>
<th>Kernel</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relax</td>
<td>scatter</td>
<td>low</td>
<td>high</td>
<td>low</td>
</tr>
<tr>
<td>Minimum</td>
<td>coalesced</td>
<td>low</td>
<td>low</td>
<td>medium</td>
</tr>
<tr>
<td>Update</td>
<td>coalesced</td>
<td>low</td>
<td>low</td>
<td>low</td>
</tr>
</tbody>
</table>

Figure 11. Some declaration examples for the automatic GPU kernel optimizations.

Algorithm 1 GPU implementation of Crauser’s algorithm. Kernels are delimited by <<< ... >>>.

1: \(<\text{initialize}>(U, F, \delta)\); //Initialization
2: \(\text{while}(\Delta \neq \infty)\) do
3: \(<\text{relax}>(U, F, \delta)\); //Edge relaxation
4: \(\Delta = <\text{minimum}>(U, \delta)\); //Settlement step,1
5: \(<\text{update}>(U, F, \delta, \Delta)\); //Settlement step,2
6: end while
6. Experimental evaluation

This section describes the methodology used to test the TuCCompi model, the platforms used, and the input set characteristics for the case study (the APSP problem). Finally, the experimental results and conclusions are shown.

6.1 Methodology

In order to evaluate TuCCompi for heterogeneous environments, we have tested the APSP problem as a case study (see Sect. 5) in different scenarios. Each scenario was designed with the aim to check the use of the layers involved in each scenario in an incremental fashion.

- A single GPU, that uses the 3rd, 4th and the tuning layer.
- Two GPUs, that involves the 2nd layer in addition to the previous ones.
- Pegasus: A shared-memory system with two GPUs and eight CPU-cores (two for handling the GPUs and six for computing), in order to test the 2nd layer mixing two different kind of computational units.
- Small HC: Small heterogeneous cluster, that uses all layers of TuCCompi.
- Big HC: Big heterogeneous cluster, with the aim to evaluate the scalability of the model.

The workload scheduling used for these environments was the master-slave policy.

Finally, with the aim of testing the performance gain offered by the innovative 4th and Tuning layers, we have compared the execution of a single GPU without these layers with respect to the use of them. For the former execution, we have chosen some of the optimal values recommended by CUDA that maximizes the GPU occupancy and only one kernel at a time. These experiments have been carried out just computing a small quantity of tasks (1024, 2048, 4096, 8102, 16204, and 32408).

6.2 Target Architectures

Table 3 describes the heterogeneous platforms used for our experiments. For each node we indicate the number of CPU-cores and the GPUs used. This heterogeneous cluster contains a total of 180 CPU-cores and 3 GPUs. However, each GPU device is governed by a single CPU core, thus, the total number of real computational units is 180. Some cluster nodes are virtual machines (VM), that have been running on processors that had additional, low workloads of other desktop virtual machines. The multi-GPU systems includes the two devices present in the pegasus machine, and the single GPU scenario uses the most powerful of them, the GeForce GTX 480.

The first heterogeneous cluster evaluated, named Small HC, is a shared-memory system with two GPUs and eight CPU-cores (two for handling the GPUs and six for computing), in order to test the 2nd layer mixing two different kind of computational units.

Finally, with the aim of testing the performance gain offered by the innovative 4th and Tuning layers, we have compared the execution of a single GPU without these layers with respect to the use of them. For the former execution, we have chosen some of the optimal values recommended by CUDA that maximizes the GPU occupancy and only one kernel at a time. These experiments have been carried out just computing a small quantity of tasks (1024, 2048, 4096, 8102, 16204, and 32408).
Table 3. Summary of heterogeneous clusters.

<table>
<thead>
<tr>
<th>Node</th>
<th>CPU Info</th>
<th>#CPU Cores</th>
<th>GPU Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pegaso</td>
<td>IC2 E7310 1.6GHz</td>
<td>8</td>
<td>GeForce 9600GT</td>
</tr>
<tr>
<td>Nodoyuna</td>
<td>IC2 Q6600 2.40GHz</td>
<td>8</td>
<td>GeForce 9600GT</td>
</tr>
<tr>
<td>Trago</td>
<td>IC2 Q6600 2.40GHz</td>
<td>8</td>
<td>GeForce 9600GT</td>
</tr>
<tr>
<td>Apolo</td>
<td>IC2 Q6600 2.40GHz</td>
<td>8</td>
<td>GeForce 9600GT</td>
</tr>
<tr>
<td>Geopar</td>
<td>IX E7310 1.6GHz</td>
<td>2</td>
<td>GeForce 9600GT</td>
</tr>
<tr>
<td>Patan</td>
<td>IX E5650 2.33GHz</td>
<td>2</td>
<td>GeForce 9600GT</td>
</tr>
<tr>
<td>Ato01</td>
<td>IC2 6300 1.86GHz</td>
<td>2</td>
<td>GeForce 9600GT</td>
</tr>
<tr>
<td>Ato02</td>
<td>IC2 6300 1.86GHz</td>
<td>2</td>
<td>GeForce 9600GT</td>
</tr>
<tr>
<td>Ato03</td>
<td>AMD AIX2 6000+</td>
<td>2</td>
<td>GeForce 8500GT</td>
</tr>
<tr>
<td>Ato09</td>
<td>IC Q8299 2.33GHz</td>
<td>4</td>
<td>GeForce 8500GT</td>
</tr>
</tbody>
</table>

Small HC: Small HC plus the following machines

<table>
<thead>
<tr>
<th>Node</th>
<th>CPU Info</th>
<th>#CPU Cores</th>
<th>GPU Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Titan01 (VM)</td>
<td>IX E5-2620 2.00GHz</td>
<td>4</td>
<td>GeForce GTX 680</td>
</tr>
<tr>
<td>Titan02 (VM)</td>
<td>IX E5-2620 2.00GHz</td>
<td>4</td>
<td>GeForce GTX 680</td>
</tr>
<tr>
<td>Titan03 (VM)</td>
<td>IX E5-2620 2.00GHz</td>
<td>8+8</td>
<td>GeForce GTX 680</td>
</tr>
<tr>
<td>Titan04 (VM)</td>
<td>IX E5-2620 2.00GHz</td>
<td>8+8</td>
<td>GeForce GTX 680</td>
</tr>
<tr>
<td>Titan05 (VM)</td>
<td>IX E5-2620 2.00GHz</td>
<td>12+12</td>
<td>GeForce GTX 680</td>
</tr>
<tr>
<td>Ato05 (VM)</td>
<td>IX E5-2620 2.00GHz</td>
<td>8+8</td>
<td>GeForce GTX 680</td>
</tr>
<tr>
<td>Ato06 (VM)</td>
<td>IX E5-2620 2.00GHz</td>
<td>8+8</td>
<td>GeForce GTX 680</td>
</tr>
<tr>
<td>Ato07 (VM)</td>
<td>IX E5-2620 2.00GHz</td>
<td>12+12</td>
<td>GeForce GTX 680</td>
</tr>
<tr>
<td>Ato08 (VM)</td>
<td>IX E5-2620 2.00GHz</td>
<td>12+12</td>
<td>GeForce GTX 680</td>
</tr>
</tbody>
</table>

6.3 Input Set Characteristics

The input set is composed by a collection of graphs randomly generated by a graph-creation tool used by [13] in their experiments. The graphs have been created adding seven adjacent predecessors to each node of the graph. Afterwards, they have inverted the graphs in order to store the node successors sequentially. These graphs are represented through adjacency lists, with the nodes numbered from 0 to |V| - 1, and integers that randomly range from 1 to 10 for edge weights.

We have used four different graph-size in order whose number of vertices are 1 049 088, 1 509 888, 2 001 408 and 2 539 008. These sizes have been chosen because they are multiple of the thread-Block sizes considered. In this way the GPU algorithm is easier to implement because we do not have to add padding techniques that avoids out-of-memory errors.

6.4 Experimental results

GPUs vs the heterogeneous environments

Figure 14 shows the execution times for the single GPU, the multi-GPU system and the two heterogeneous cluster scenarios. Although the GPUs are the most powerful devices, and their combined use significantly decreases the execution times, the addition of many less-powerful computational units enhances even more the total performance gain. We can observe that the execution times have been reduced as more computational resources are used. Moreover, the use of this model has a communication overhead lower than 1 percent.

The overhead of the Small-HC have never surpassed 0.589% of the total execution time. Figure 15 represents the task distribution between the Big-HC nodes for the executed master-slave policy. Furthermore, the figure shows the theoretical distribution for each cluster node if the equitable policies, EQ1 and EQ2, were used.

The 4th and Tuning layers performance gain

The comparison between the worst execution on the GPU GeForce GTX 480, with only one kernel per time, together with the threadBlock values recommended by CUDA, with respect the concurrent kernel execution combined with the values proposed in [21] is shown in Fig. 16. The use of these layers reduces for our test case up to 12% the execution time.

7. Related work

IlCoMP [18] is a source-to-source compiler that translates C annotated code to MPI + OpenMP or CUDA code. The user needs to specify the sequential code that he/she wants to parallelize. The authors are only focused in parallel-loop problems. This compiler does not support the jointly use of CUDA with any other parallel model, therefore, it is not appropriate to be used in heterogeneous environments. Besides this, the IlCoMP compiler does not easily support a new GPU architecture or other kind of hardware accelerators. The authors in [12] propose a framework called OM-PICUDA to develop parallel applications on the hybrid CPU/GPU clusters by mixing OpenMP, MPI and CUDA models. Besides this, they include compiler that translates automatically OpenMP source code to CUDA. This framework presents serious programming limitations. First, it does not support any recursive function. Second, the 1.X CUDA architectures can not be used because of the way...
Given a GPU implementation, Ocelot transforms it to sequential code. TuCCompi model does not aim to deal with sequential-to-parallel code transformation. However, both proposals described above and many others can be easily attached to our multilayer model (see Fig. 17). Additionally, in order to solve the automatic GPU kernel characterization is also easy to attach a module that analyzes the GPU implementation and connects its output to the Tuning layer.

8. Conclusions and future work

We propose TuCCompi, a multilayer deployment model that helps the programmer to easily obtain flexible and portable programs that automatically detect at run-time the available computational resources and exploits hybrid clusters with heterogeneous devices. This model offers to the programmer a transparent and easy mechanism to select the optimal values of GPU configuration parameters just characterizing the nature of his kernels. Any parallel application that can be devised as a collection of non-dependent tasks just characterizing the nature of his kernels. Any parallel application that can be devised as a collection of non-dependent tasks can be attached to the environment, with a communication overhead less than the one percent of the total execution time.

The model is designed to provide a mechanism of plug-ins, in order to easily change: (1) The corresponding algorithms that are wanted to be deployed; (2) The scheduling policies of the task division; and (3) The parameter values for GPUs optimal configurations; without making any change in the model. Furthermore, it is easily to attach some research works related with parallel code transformations; without making any change in the model. Furthermore, it is easily to attach some research works related with parallel code transformation in order to give a complete tool to the user. The use of this model exploits even the less powerful devices of a heterogeneous cluster and correctly scales if more computational units are added to the environment, with a communication overhead less than the one percent of the total execution time.

As future work, we plan to include support for TBBs and OpenCL languages into the framework, in order to have a suite of parallel models for each layer. In this way, the user will be able to choose the model that better fits the specific problem and heterogeneous environment. Other uses for TuCCompi include massively parallel problems such as Bitcoons currency mining, or molecular computations, as well as other kind of parallelizable problems.
Acknowledgments
The authors would like to thank Javier Ramos López for the creation and management of different virtual machines added to the heterogeneous environment. This research is partly supported by the Spanish Government (TIN2007-62302, TIN2011-25639, CENIT OCEANLIDER, CAPAP-H networks TIN2010-12011-E and TIN2011-15734-E), Junta de Castilla y León, Spain (VA094A08, VA172A12-2), and the HPC-EUROPA2 project (project number: 228398) with the support of the European Commission - Capacities Area - Research Infrastructures Initiative and the ComplexHPC COST Action.

References