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-Tunig, CUDA, GPU, Heterogeneous system, HPC framework, MPI, OpenMP, Parallel model

1. Introduction

Some 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 [21], some molecular physics computations [2], biomedical-domain data processing [11], computational geometry problems [12] or Bitcoin mining [22].

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 schedule the workload in such complex environments. Some works have tried to ease the jointly use of parallel programing 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 [19]. However, in that work CUDA can not be jointly used with the other parallel models. Another example is OMPICUDA [13], 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.

The second limitation is the lack of a tuning methodology that efficiently unleashes the power of GPU devices. There is not known a parallel model that automatically selects the optimal values for CUDA configuration parameters, such as the threadBlock size-shape, or the state of L1 cache memory, for each kernel. These optimization techniques significantly enhance the GPU performance. 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. Several studies [23, 24] 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 important configuration parameters. Each layer represents a level of parallelism (see Fig. 1). The first layer handles the distributed-memory environment, coordinating the nodes. The second layer manages the computational units which belong to the shared-memory environment inside a node. The third layer automatically deploys

CONF 'yy, Month d–d, 20yy, City, ST, Country.
Copyright © 20yy ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

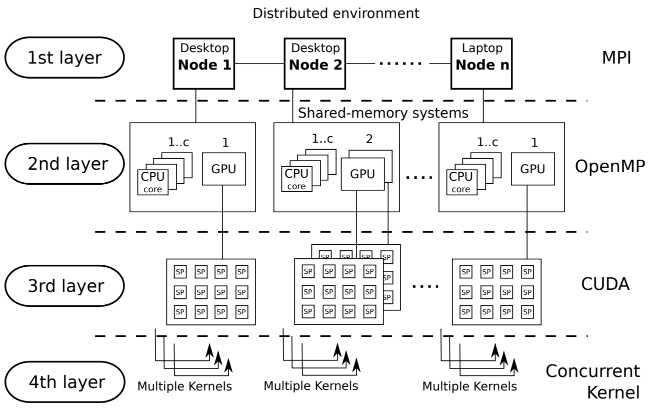

Figure 1. Layer deployment of TuCCompi model in a heterogeneous cluster.

the execution in hardware accelerators such as GPUs. The fourth layer automatically handles concurrent works inside these GPUs. Finally, an internal Tuning mechanism automatically selects the optimal values for GPU configuration parameters for each kernel and each GPU architecture. We have developed a prototype framework to test this model, allowing any user to transparently take advantage of all computational capabilities of both, CPU-cores and GPU devices, distributed in different shared-memory systems, without having a deep knowledge of parallel programming methods. The case study used to evaluate the model is the All-Pair Shortest-Path (APSP) problem. Experiments have been run in an academic heterogeneous environment.

The contributions of this work are: (a) Mechanisms to automatically choose the optimal values for important CUDA configuration parameters, based on provided programmer kernel characterization, for any current kind of GPU architecture; (b) The automatic exploitation of a modern GPU feature such as the concurrent-kernel execution as a new dimension of parallelism; (c) The creation of a specific prototype framework that combines the use of these two novel layers and the traditional ones, whose use leads to performance improvements up to 12% in a test case.

The rest of this paper is organized as follows. Section 2 introduces our conceptual approach. Section 3 describes the use of the model through some code snippets. Section 4 explains the case study used. In Sect. 5 we present the experimental environment and the results obtained. Section 6 describes some related work. Finally, Sect. 7 summarizes our conclusions.

2. TuCCompi Architecture

This section gives a description of the different layers defined in our model. The use of a multi-language framework provides the user with mechanisms to obtain a good performance, tuning the devices in an optimal way. A graphical representation is depicted in Fig. 1.

The 1st layer (distributed environment) Nowadays, one of the most economic ways to assemble a heterogeneous system is to interconnect a set of individual machines, also called nodes, such as personal computer, laptops, complex virtual host machines or even other supercomputing systems composed in turn by other machines. The nodes found in these heterogeneous environments usually consist in shared-memory systems, with very different computational capabilities. It is necessary to apply 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 accelerator devices, such as GPUs, FPGAs and Xeon Phi, that are usually controlled by a host system (CPU) and are capable of executing kernels independently. 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 such accelerator 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 management of those devices, 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, FPGAs and Xeon Phi. 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, in addition to the sequential execution of the remaining node CPU-cores. In our prototype implementation we have included CUDA to manage GPU devices.

The 4th layer (concurrent GPU kernel execution) The most recently NVDIA GPUs support concurrent-kernel execution [15, 16], 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 of bigger kernels also gives performance improvements. This occurs because several kernels of the same application context that, work on the same memory areas take advantage of the L1 data-cache, originating less number of cache-misses and therefore alleviating the global memory bottlenecks. Additionally, the threadBlock-warp dispatcher schedule kernels faster if they have been previously launched [17].

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 in traditional CPU cores. Usually, it requires an extensive CUDA programming experience. Some examples of code tuning strategies are the choice

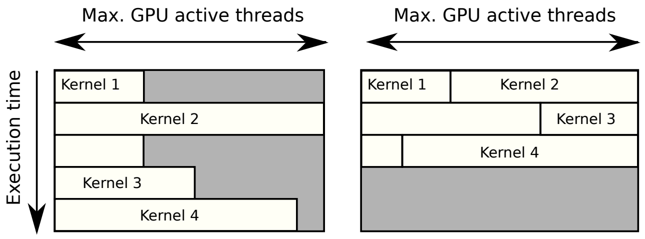

Figure 2. Sequential (left) vs concurrent execution (right) of several kernels. When the total needed threads of a kernel surpass the maximum GPU active threads (number of concurrent threads supported by each Streaming Multiprocessor (SM) $\times$ number of SMs) CUDA driver is the responsible of splitting it into two sequential steps.

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

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

of an appropriate threadBlocks size and shape, the coalescing maximization of the memory accesses, or the occupancy maximization of the Streaming Multiprocessor, among others. Moreover, the resource differences between each GPU architecture and release, such as the number of computational units, cache-sizes, and other features, make even more difficult to find the optimal configuration for each GPU. Besides this, the optimal values also depend of the access-memory pattern and the characteristics of the code of each executed kernel. The Tuning layer automatically selects some important parameter values for each kernel and GPU, based on a kernel characterization provided by the application programmer.

3. TuCCompi Model Usage

TuCCompi users should provide to the model the following (see Fig. 3): (a) The sequential-CPU and the parallel-GPU code applications, that are named as `PLUG-IN_CPU` and `PLUG-IN_GPU` respectively, (b) the kernel characterizations used in the GPU code, and (c) the main C language program with TuCCompi primitives and macros.

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

3.1 Kernel characterization

The user has to provide a general characterization of her kernels along with its definition. This information is easily expressed in our

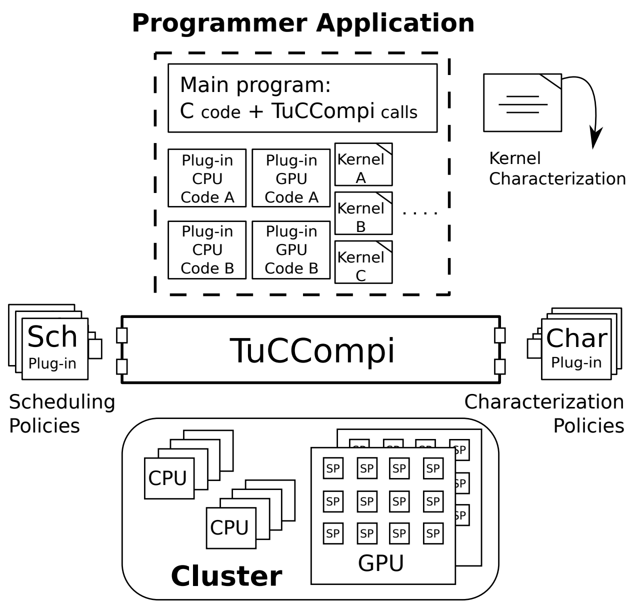

Figure 3. TuCCompi model usage. Elements in the dashed box are provided by the programmer. Note that the user can develop different versions of each plug-in (Code A, Code B, . . .) but only one at a time will be deployed into TuCCompi framework.

```
K00: TuCCompi_KERNELCHAR(k1, 2, scatter, none, high, low);
K01: __global__ void k1 (...){
K02:   (kernel implementation)
K03: }
K04: TuCCompi_KERNELCHAR(k2, 1, coalesced, low, low, high);
K05: __global__ void k2 (...){
K06:   (kernel implementation)
K07: }
```

Figure 4. Kernel characterizations and implementations. The programmer adds the boxed primitive before the kernel implementation to characterize it.

prototype implementation through the `TuCCompi_KERNELCHAR( kernel_name, num_dims, A, B, C, D)` primitive. The values for parameters *A*, *B*, *C* and *D* have to be chosen from the kernel-characterization classification shown in Table 1. TuCCompi model automatically optimizes the use of all underlying hardware resources of GPU devices, following the guidelines and optimizations proposed in [24] for each possible combination of these parameters.

Figure 4 shows some examples of the code used to characterize the kernels. Lines K00 and K04 describes the characterization of kernels *k1* and *k2* respectively, indicating the kernel name, the number of dimensions of the threadBlock, and the classification criteria described in Tab. 1. In the case that the user does not know how to classify her kernels, she can use the default (`def`) values provided by the model. The primitive used for this default case is `TuCCompi_KERNELCHAR(kernel_name, num_dim, def, def, def, def)`.

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 inserting the code to implement the algorithm that solves a single task (line C01, `Cpu user code`).

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

Figure 5. Plugin_Cpu interface. The programmer adds to her code the boxed arguments to deploy the Cpu plugin in TuCCompi.

```
G00:  plugin_Gpu(user_vars ...) {
G01:    (Gpu user code)
G02:   TuCCompi_GPULAUNCH(k1, input_size,
G03:           TuCCompi_PARLLMK(vector1, type, lng), ...);
G04:    TuCCompi_GPUSYN( );
G05:   TuCCompi_GPULAUNCH(k2, input_size2,
G06:           TuCCompi_PARLLMK(vector2, type, lng), ...);
G07:    TuCCompi_GPUSYN( );
G08:  }//pluginGPU
```

Figure 6. Plugin_Gpu interface and internal structure. The programmer has to replace the typical CUDA kernel launch primitives for the boxed TuCCompi macros.

Figure 6 shows the code that will be executed in a CPU to manage the associated GPUs. 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 TuCCompi macro that carries out a kernel launch, with the name of the kernel as first parameter, and followed by other user variables that have been previously allocated in the GPU. Transparently for the user, the model executes as many kernel instances as the MK value defined in the main program control by the user (see line M03 of Fig. 7). Every concurrent kernel launched will need its own workspace to compute its results. The primitive 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 elements that it contains; and the number of elements that compounds it. As we said before, the algorithm implementation can require the execution of different kernels that should be sequentially launched for a single task computation (line G05). The TuCCompi primitive of Line G04 forces the CPU to wait for the finalization of an executing kernel, or for the termination of all kernels that many have been launched, in order to provide a synchronization mechanism if needed.

3.3 TuCCompi main program implementation

Figure 7 shows an example of the code that the user has to implement in order to start an control the execution in our prototype. The primitive `TuCCompi_COMM` in Line M01 initializes the system. Afterwards, the user can introduce her code, including variable declarations, initializations and the sequential code needed for the application. Line M03 shows the primitive needed to set the number of tasks that the GPU devices have to execute concurrently if they have such possibility. Line M04 shows the primitive 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 (described below). In line M05 the programmer specifies that the process has to wait until all computational units of the cluster node

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

Figure 7. User implementation of the TuCCompi main-program. The programmer has to add to her code the boxed primitives.

have finished. The user is free to insert more code to execute other parallelization instances, if needed, before the finalization of the heterogeneous environment communication, shown in line M07.

3.4 Workload scheduling

TuCCompi 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 from the master until it sends a termination signal to them. As the master can be located at any cluster node, these asking-for-tasks requests are issued through distributed-environment communications.

4. Case study

In order to check TuCCompi framework prototype, we have chosen the All-Pair Shortest-Path (APSP) problem for sparse graphs as our case study because it is a representative example with 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 kind of 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 in more detail and we describe the corresponding plug-ins developed for the TuCCompi model.

4.1 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

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

```
1: <<<initialize>>> $(U, F, \delta)$;            //Initialization
2: while $(\Delta \neq \infty)$ do
3:     <<<relax>>> $(U, F, \delta)$;             //Edge relaxation
4:     $\Delta$ =<<<minimum>>> $(U, \delta)$;    //Settlement step_1
5:     <<<update>>> $(U, F, \delta, \Delta)$;    //Settlement step_2
6: end while
```

Table 2. Summary of kernels characterization.

Kernel	A	B	C	D
Relax	scatter	low	high	low
Minimum	coalesced	low	low	medium
Update	coalesced	low	low	low

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 Δ_i as more as possible to process more nodes in the next iteration.

4.2 Plug-ins for our example

Both sequential and parallel GPU codes are implementations of the Crauser algorithm. Their implementation for this problem has been taken from [18]. Algorithm 1 shows the GPU parallel pseudo-code of Crauser's algorithm. Figure 9 shows the TuCCompi implementation for the pluginGPU. This implementation repeatedly launches three kernels (relax, minimum and update) with different features. Following the classification criteria described in Sect. 3.1, the kernels are characterized in Table 2.

Regarding to the scheduling issue, due to the parallel nature of the problem we have defined each SSSP computation as a single independent task. We have implemented our own master-slave scheduling plug-in (see Fig. 8). The master differentiates the nature of the slave that is requesting a task. Depending on the slaves computational power, the master will send more or less tasks. The TuCCompi model is better exploited if the master gives more tasks to the modern GPUs (Fermi, Kepler and so on) due to their multi-kernel execution feature. For our particular master, we decided to dispatch four tasks for each modern GPU, and only one for the Pre-Fermi architectures and the CPU cores.

Figure 8 (top) shows the master implementation. The master will manage the task distribution while there are task to be executed (lines 01-16). To do so, the master waits for a task request from any slave (line 3). If the slave is a modern GPU (Fermi or Kepler) (line 04), the master checks if there are MK available tasks to be sent. In this case, it sends the pack to the corresponding slave using its identifier, and updates the task counter (lines 05-07). However, if there are not enough tasks for this type of slave, the master sends to it the termination signal and updates the counter of slaves that have already finished (lines 08-11). If the requesting slave is an old GPU (pre-Fermi) or a CPU-core, the master only sends a single task to the slave (lines 12-15). Afterwards, the task counter is updated. When all tasks have been scheduled and carried out, the master sends to the finishing slaves the termination signal and updates the corresponding counter (lines 17-21).

Figure 8 (bottom) shows the slave implementation. First, the slave notifies the master that it is idle (line 1). Then the slave receives the task(s) to be executed (line 2). Finally, the slave returns the task identification (line 3).

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

```
00:  int slave(id_slave, mpi_master, tag){
01:    SEND(id_slave, mpi_master, tag);
02:    RECV(task, mpi_master, id_slave);
03:    return task;
04:  }
```

Figure 8. Our case-study implementation for the functions, master (top) and slave (bottom), of the distribution plug-in.

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

Figure 9. Case-study user implementation for pluginGPU.

5. 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 a discussion is presented.

5.1 Methodology

In order to evaluate TuCCompi for heterogeneous environments, we have tested the APSP problem as a case study (see Sect. 4) 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. Architecture details are shown in Table 3:

- A single GPU, that uses the 3rd, 4th and the tuning layer.

Table 3. Summary of heterogeneous clusters.

Small HC

Node	CPUInfo	#CPUcores	GPU details
Pegaso	IC2 i7 960 3.20GHz	8	GeF GTX 480 GeF GTX 680
Nodoyuna	IC2 Q8200 2.33GHz	4	-
Trasgo	IC2 Q6600 2.40GHz	4	-
Apolo	IC2 Q6600 2.40GHz	4	-
Geopar	IX E7310 1.6GHz	16	-
Patan	IC2 E6550 2.33GHz	2	-
Atc01	IC2 6300 1.86GHz	2	GeF 9600GT
Atc02	IC2 6300 1.86GHz	2	-
Atc03	AMD AtX2 3600+	2	GeF 8500GT
Atc09	IC Q8299 2.33GHz	4	-

Big HC: Small HC plus the following machines

Node	CPUInfo	#CPUcores	GPU details
Titan01	IX E5-2620 2.00GHz	4	-
Titan02	IX E5-2620 2.00GHz	4	-
Titan03	IX E5645 2.40GHz	8+8	-
Titan04	IX E5645 2.40GHz	8+8	-
Titan05	IX E5-2620 2.00GHz	12+12	-
Atc05	IX E5630 2.53GHz	8+8	-
Atc06	IX E5630 2.53GHz	4	-
Atc07	IX X-5675 3.07GHz	12+12	-
Atc08	IX E5-2620 2.00GHz	12+12	-

- Two GPUs, that involve the 2nd layer in addition to the previous ones.
- *Pegaso*: 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 by mixing two different kinds 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 particular master-slave policy described in Fig. 8.

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 few task sets (1 024, 2 048, 4 096, 8 102, 16 204, and 32 408).

5.2 Target Architectures

Table 3 describes the heterogeneous platforms used for our experiments. For each node we indicate the number of CPUcores 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. The multi-GPU systems include the two devices present in the *Pegaso* machine, and the single GPU scenario uses the most powerful of them, the GeForce GTX 480.

The different nodes shown in Table 3 run the Ubuntu Desktop 10.04 (64 bits) operating system. The CUDA toolkit release used is 4.2 with the 295.41 64-bit driver.

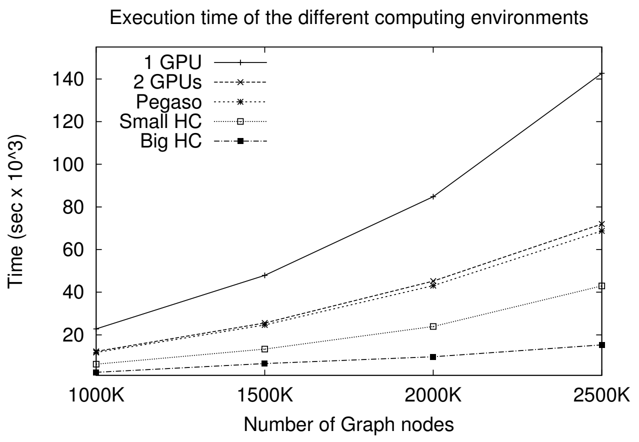

Figure 10. Execution times of the tested scenarios for different graph-sizes.

5.3 Input Set Characteristics

The input set is composed by a collection of graphs randomly generated by a graph-creation tool used by [14] in their experiments. The graphs have been created adding seven adjacent predecessors to each node of the graph. Afterward, graphs are inverted in order to store the node successors sequentially. These graphs are represented through adjacency lists, with the nodes numbered from $0 \dots |V| - 1$, and integers that randomly range from $1 \dots 10$ for edge weights.

We have used four different graph-sizes, whose number of vertices are 1 049 088, 1 509 888, 2 001 408 and 2 539 008. These sizes have been chosen because they are multiple of the threadBlock sizes considered. In this way the GPU algorithm is easier to implement because we do not have to use padding techniques to avoid buffer overrun errors.

5.4 Experimental results

GPUs vs the heterogeneous environments Figure 10 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 11 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 of 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 to the concurrent kernel execution combined with the values proposed in [24] is shown in Fig. 12. The use of the concurrent kernel layer and the optimization tuning reduces the execution time for our test case up to 12%.

6. Related work

llCoMP [19] 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 she wants to parallelize. The

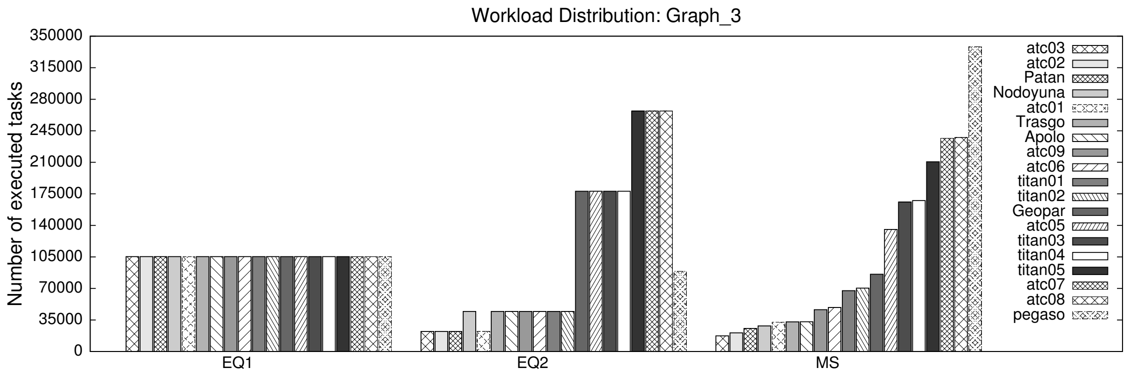

Figure 11. Number of executed tasks per cluster node with different distribution policies in the big heterogeneous cluster.

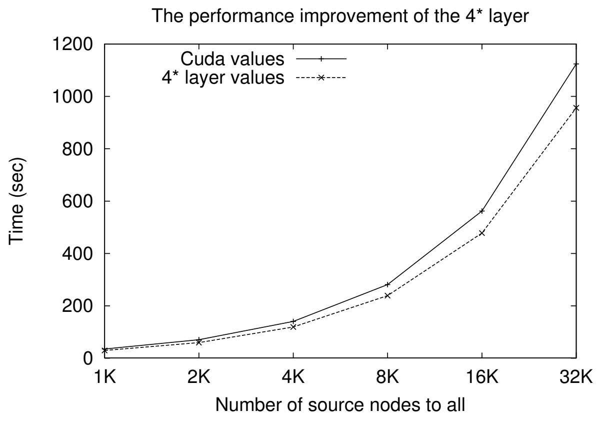

Figure 12. Performance improvements obtained by the 4th and Tuning layers with respect to CUDA recommended configuration values.

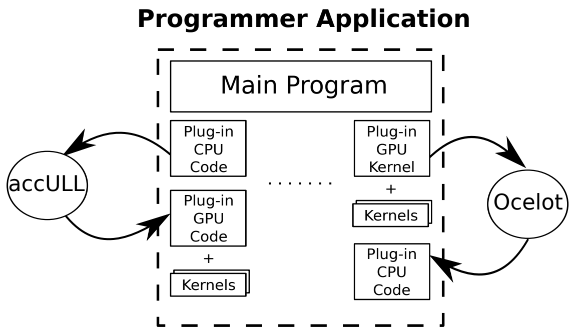

Figure 13. Usage of TuCCompi with code-transformation modules.

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 llCoMP compiler does not easily support a new GPU architecture or other kind of accelerators.

The authors in [13] propose a framework called OMPICUDA to develop parallel applications on the hybrid CPU/GPU clusters by mixing OpenMP, MPI and CUDA models. Besides this, they include a 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 pointers are used. Third, the critical OpenMP sections are not fully translated to CUDA. Fourth, this framework can not be easily modified to support a new parallel model. Finally, they do not develop any policy to select the proper values of CUDA configuration parameters. A parallel programming approach using hybrid CUDA, MPI and OpenMP programming is presented in [25]. The authors only focus on the model to solve iterative problems and they do not take into account any CUDA optimization technique. The proposed model does not support any mechanism to include new work distribution policies.

The authors in [9] have created an hybrid tool that includes the same parallel models used by the previous mentioned works to solve raycasting volume rendering algorithm. They test the system scalability when the input data size is increased. This tool is only focused in a single parallel application and does not include any CUDA optimization technique. Moreover, this work does not include any automatic mechanism to efficiently exploit all available hardware devices in heterogeneous environments. Additionally, the proposed models can not be used for other kinds of parallel problems.

Another work in this field is StarPU [10], a task programming library for hybrid architectures supporting GPUs. However, to the best of our knowledge, StarPU does not include the concurrent-kernel feature of modern GPUs, nor our tuning techniques for better exploiting GPU computational capabilities.

With respect to sequential-to-parallel code transformation, proposals in this field include accULL [20], that receives a sequential code of an algorithm and automatically transforms it to parallel code that can be deployed into GPU devices. Another example of code transformation is Ocelot [6], that works in the opposite 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. 13). Additionally, in order to solve the automatic GPU kernel characterization, it is also easy to attach a module that analyzes the GPU implementation and connects its output to the Tuning layer.

7. 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 the kernels. Any parallel application that can be devised as a collection of non-dependent tasks working on shared data-structures can be exploited with the current model of TuCCompi.

The use of new layers to exploit a concurrent kernel execution in GPUs and optimal parameters adds a novel parallel dimension and a new automatic optimization compared with previous works, representing in our test case a performance gain up to the 12% for the GPUs usage. Therefore, these new layers turn out to be very important for heterogeneous environments that include these GPU devices.

The model is designed to provide a mechanism of plug-ins, in order to easily change: (1) The algorithms to be deployed; (2) the scheduling policies of the tasks; and (3) the parameter values for GPUs optimal configurations without making any change in the model. Furthermore, it is easy to exploit tools generated by other research works related with parallel code transformation in order to give a complete single programming environment 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 one percent of the total execution time.

As future work, we plan to devise other uses for TuCCompi, including massively parallel problems such as Bitcoins currency mining or molecular computations, as well as other kind of parallelizable problems.

Acknowledgments

The authors would like to thank Javier Ramos López for his support with technical issues. This research has been partially supported by Ministerio de Economa y Competitividad (Spain) and ERDF program of the European Union: CAPAP-H4 network (TIN2011-15734-E), MOGECOPP project (TIN2011-25639); and Junta de Castilla y Len (Spain): ATLAS project (VA172A12-2).

References

[1] A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. O. Storaasli. State-of-the-art in heterogeneous computing. *Sci. Program.*, 18(1):1–33, Jan. 2010. ISSN 1058-9244.

[2] Z. Chen, X. Chen, Z. Shao, Z. Yao, and L. T. Biegler. Parallel calculation methods for molecular weight distribution of batch free radical polymerization. *Computers & Chemical Engineering*, 48(0): 175–186, 2013. ISSN 0098-1354.

[3] W. Cirne, D. Paranhos, L. Costa, E. Santos-Neto, F. Brasileiro, J. Sauve, F. A. B. Silva, C. Barros, and C. Silveira. Running bag-of-tasks applications on computational grids: The Mygrid approach. In *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, pages 407–416, 2003.

[4] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders. A parallelization of Dijkstra's shortest path algorithm. In L. Brim, J. Gruska, and J. Zlatuška, editors, *Mathematical Foundations of Computer Science 1998*, volume 1450 of *LNCS*, pages 722–731. Springer Berlin / Heidelberg, 1998. ISBN 978-3-540-64827-7. 10.1007/BFb0055823.

[5] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. ISSN 0029-599X.

[6] N. Farooqui, A. Kerr, G. F. Diamos, S. Yalamanchili, and K. Schwan. A framework for dynamically instrumenting GPU compute applications within GPU Ocelot. In *Proceedings of 4th Workshop on General Purpose Processing on Graphics Processing Units, GPGPU 2011, Newport Beach, CA, USA, March 5, 2011*, page 9. ACM, 2011. ISBN 978-1-4503-0569-3.

[7] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0201575949.

[8] U. Hoelzle and L. A. Barroso. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 1st edition, 2009. ISBN 159829556X, 9781598295566.

[9] M. Howison, E. Bethel, and H. Childs. Hybrid parallelism for volume rendering on large-, multi-, and many-core systems. *Visualization and Computer Graphics, IEEE Transactions on*, 18(1):17–29, 2012. ISSN 1077-2626.

[10] A.-E. Hugo, A. Guermouche, P.-A. Wacrenier, and R. Namyst. Composing Multiple StarPU Applications over Heterogeneous Machines: A Supervised Approach. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '13, pages 1050–1059, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-4979-8.

[11] S. Kazmi, M. Kane, and M. Krauthammer. Benchmarking technology infrastructures for embarrassingly and non-embarrassingly parallel problems in biomedical domain. In *Biomedical Sciences and Engineering Conference (BSEC), 2013*, pages 1–4, 2013. .

[12] A. Khlopotine, V. Jandhyala, and D. Kirkpatrick. A variant of parallel plane sweep algorithm for multicore systems. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 32(6):966–970, 2013. ISSN 0278-0070. .

[13] T.-Y. Liang, H.-F. Li, and J.-Y. Chiu. Enabling Mixed OpenMP/MPI Programming on Hybrid CPU/GPU Computing Architecture. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 2369–2377, 2012.

[14] P. Martín, R. Torres, and A. Gavilanes. CUDA solutions for the SSSP problem. In G. Allen, J. Nabrzyski, E. Seidel, G. van Albada, J. Dongarra, and P. Sloot, editors, *Computational Science – ICCS 2009*, volume 5544 of *LNCS*, pages 904–913. Springer Berlin / Heidelberg, 2009. ISBN 978-3-642-01969-2. 10.1007/978-3-642-01970-8_91.

[15] NVIDIA. Whitepaper: NVIDIA's next generation CUDA compute architecture: Fermi, 2010.

[16] NVIDIA. NVIDIA GeForce GTX 680, 2012.

[17] H. Ortega-Arranz, A. Gonzalez-Escribano, and D. Llanos. A Tuned, Concurrent-Kernel Approach to the APSP problem. In *The 13th International Conference Computational and Mathematical Methods in Science and Engineering, CMMSE 2013*, 2013. ISBN 978-84-616-2723-3.

[18] H. Ortega-Arranz, Y. Torres, D. R. Llanos, and A. Gonzalez-Escribano. A New GPU-based Approach to the Shortest Path Problem. In *High Performance Computing and Simulation (HPCS), 2013 International Conference on*, pages 505–512, 2013.

[19] R. Reyes and F. de Sande. Optimization strategies in different CUDA architectures using llCoMP. *Microprocess. Microsyst.*, 36(2):78–87, Mar. 2012. ISSN 0141-9331.

[20] R. Reyes, I. López-Rodríguez, J. J. Fumero, and F. de Sande. accULL: an OpenACC implementation with CUDA and OpenCL support. In *Proceedings of the 18th international conference on Parallel Processing*, Euro-Par'12, pages 871–882, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-32819-0.

[21] A. A. Saba, S. Mohan, and R. Mangharam. Anytime algorithms for multi-core architectures. *Proceedings Work-in-Progress Session*, 2010.

[22] M. Taylor. Bitcoin and the age of bespoke silicon. In *Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2013 International Conference on*, pages 1–10, 2013. .

[23] Y. Torres, A. Gonzalez-Escribano, and D. Llanos. Using Fermi architecture knowledge to speed up CUDA and OpenCL programs. In *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, pages 617–624, 2012. .

[24] Y. Torres, A. Gonzalez-Escribano, and D. R. Llanos. uBench: Exposing the impact of CUDA block geometry in terms of performance. *The Journal of Supercomputing*, pages 1–14, 2013. ISSN 0920-8542.

[25] C.-T. Yang, C.-L. Huang, and C.-F. Lin. Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters. *Computer Physics Communications*, 182:266–269, Jan. 2011.