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

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 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 [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

CONF 'yy, Month d–d, 20yy, City, ST, Country.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

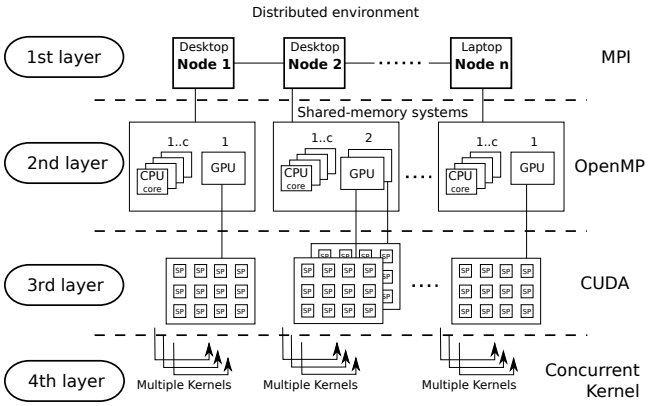

Figure 1. Layer deployment of TuCCompi model in a heterogeneous cluster.

it. The second layer manages the computational units inside these shared-memory systems. The third layer automatically deploys the execution in hardware accelerators such as GPUs. The fourth layer automatically handles concurrent works inside these GPUs. Finally, a Tuning layer 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 the CUDA configuration parameters, using a given kernel characterization, for any current kind of GPU architecture; (b) The use of a modern GPU feature 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 to the traditional ones, whose use leads to performance improvements in our test case up to 12%.

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 discusses TuCCompi internals. Section 5 explains the used case study. In Sect. 6 we present the experimental environment and the results obtained. Section 7 describes some related work. Finally, Sect. 8 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 us with more mechanisms to obtain a better 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-power 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 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 NVDIA 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

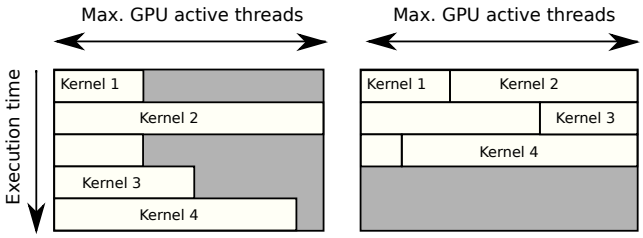

Figure 2. Sequential (left) vs concurrent execution (right) of several kernels. When the total needed threads of a kernel surpasses the maximum GPU active threads (like kernel 2), CUDA driver is the responsible of splitting it into two sequential steps.

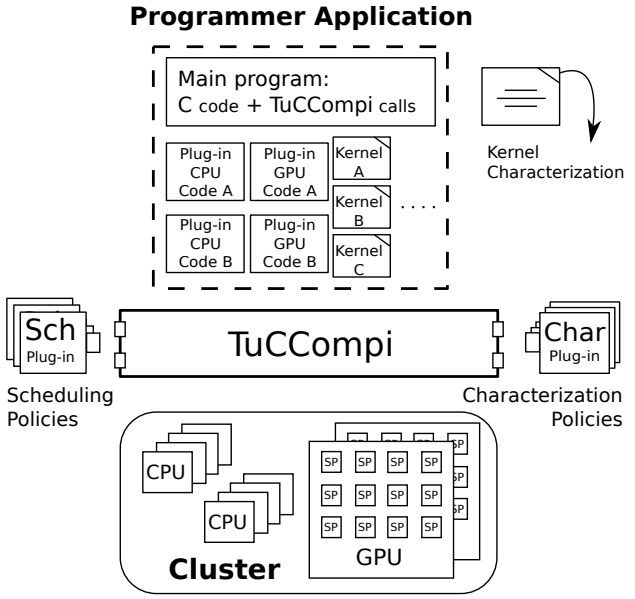

Figure 3. TuCCompi model usage. Elements in the dashed box are provided by the programmer. Note that the user can develop different versions of each plug-in (Code A, Code B, ...) but only one will be deployed into TuCCompi framework.

CPUs cores. Usually, it requires an extensive CUDA programming experience. Some examples of code tuning strategies are the choice of an appropriate threadBlocks size and shape, the coalescing maximization of the memory accesses, or the occupancy maximization of the Streaming Multiprocessor, among others. However, 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 the parameter values for an optimal configuration for each kernel and GPU, using the kernel characterization provided by the user.

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.

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

```
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 programer adds the boxed primitive before the kernel implementation to characterize it.

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

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

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_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 [21] for each of the 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 previously described classification criteria. In case that the user does not know how to classify his kernels, he can use the default (`def`) values provided by the model. The macro 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

```
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 programer has to change the typical CUDA kernel launch primitives for the boxed TuCCompi macros.

```
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 programer has to add to his code the boxed primitives.

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 it 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 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 elements that 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 macro of Line G04 forces the CPU to wait for the finalization of an executing kernel, or 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 the execution of our model. The macro `TuCCompi_COMM` in Line M01 initializes the system. Afterwards, the user can introduce his code, including variable declarations, initializations and the sequential code needed for 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 programmer 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

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 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. TuCCompi internals

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

4.1 Cluster Node Communication: `TuCCompi_COMM`

Once a heterogeneous cluster is defined and the TuCCompi 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 send each process an identification number for each resource, avoiding conflicts with other computational units. Additionally, the parent process send 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_cu` vector stores the number of computational units from each process. The `v_ id` vector stores the number from which the numeration of computational units should start for the process `i`. The `total_cu` variable stores the total number of computational units. The `id_mpi` variable stores the identifier of the MPI process. The `n_proc` variable stores the total number of MPI processes. Finally, the `PARENT` constant is the identifier 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

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

Figure 8. Implementation of the `comm()` recognition function, called from `TuCCompi_COMM()`.

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

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

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

Figure 9. `TuCCompi_PARALLEL()` and other macro-definition codes.

processes. Lines 10-14 perform the heterogeneous-environment information shipping. Lines 15-21 correspond to the behavior of the rest of process, that looks up for the available resources, sends this value to parent process and receives the cluster information.

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 com-

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

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

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

Figure 10. Declarations for the automatic kernel launch and multikernel support.

munication interruption through a `for` loop. Lines 05-07 initialize the computational units identifiers. Lines 08-09 check whether any of the current OpenMP thread should act as 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_GPULAUNCH`

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_GPULAUNCH(`*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 `PARLLMK(`*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

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

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

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

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

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 plug-in

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, pluginCPU, 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 plug-in* 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.

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

```
1: <<<initialize>>> (U, F, δ);        //Initialization
2: while (Δ ≠ ∞) do
3:     <<<relax>>> (U, F, δ);         //Edge relaxation
4:     Δ =<<<minimum>>> (U, δ);       //Settlement step_1
5:     <<<update>>> (U, F, δ, Δ);     //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

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 Δ_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.

Regarding to the scheduling issue, due to the parallel nature of the problem we have define each SSSP computation as a single independent task. We have implemented our own master-slave scheduling plug-in (See Fig. 12). The master differentiates the nature of the slave that is requesting a task. Depending on its compu-

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

tational 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 12 (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 send. In this case, it sends the pack to the corresponding slave through its identifier and updates the task counter (lines 05-07). However, if there are no 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 9 (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).

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

```
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 13. Case-study user implementation for pluginGPU.

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.
- *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 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 (1 024, 2 048, 4 096, 8 102, 16 204, and 32 408).

6.2 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. 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 *pegaso* 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 composed by the non-virtual machines. The second heterogeneous cluster, named Big HC, contains all machines described in Table 3. The VM nodes run Ubuntu Server 12.04 and Debian desktop (64

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 (VM)	IX E5-2620 2.00GHz	4	-
Titan02 (VM)	IX E5-2620 2.00GHz	4	-
Titan03 (VM)	IX E5645 2.40GHz	8+8	-
Titan04 (VM)	IX E5645 2.40GHz	8+8	-
Titan05 (VM)	IX E5-2620 2.00GHz	12+12	-
Atc05 (VM)	IX E5630 2.53GHz	8+8	-
Atc06 (VM)	IX E5630 2.53GHz	4	-
Atc07 (VM)	IX X-5675 3.07GHz	12+12	-
Atc08 (VM)	IX E5-2620 2.00GHz	12+12	-

bits) operating systems, and the remaining ones 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.

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 $0 \ldots |V| - 1$, and integers that randomly range from $1 \ldots 10$ 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 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 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.

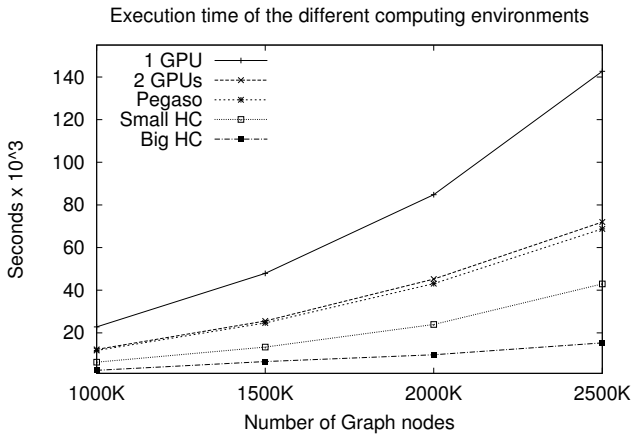

Figure 14. Execution times of the tested scenarios for different graph-sizes.

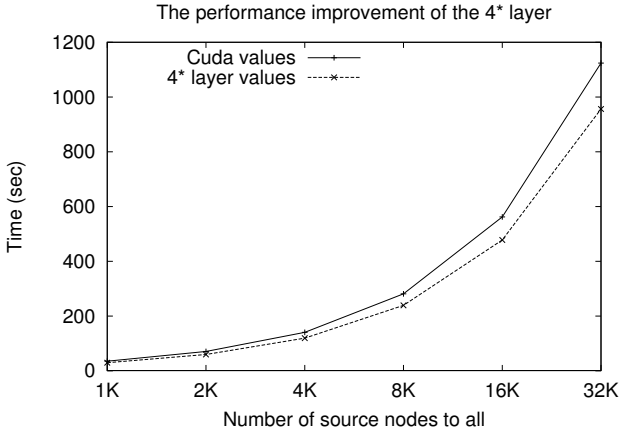

Figure 16. Performance improvements obtained by the 4th and Tuning layers with respect to CUDA recommended configuration values.

7. Related work

llCoMP [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 llCoMP compiler does not easily support a new GPU architecture or other kind of hardware accelerators. The authors in [12] 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 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

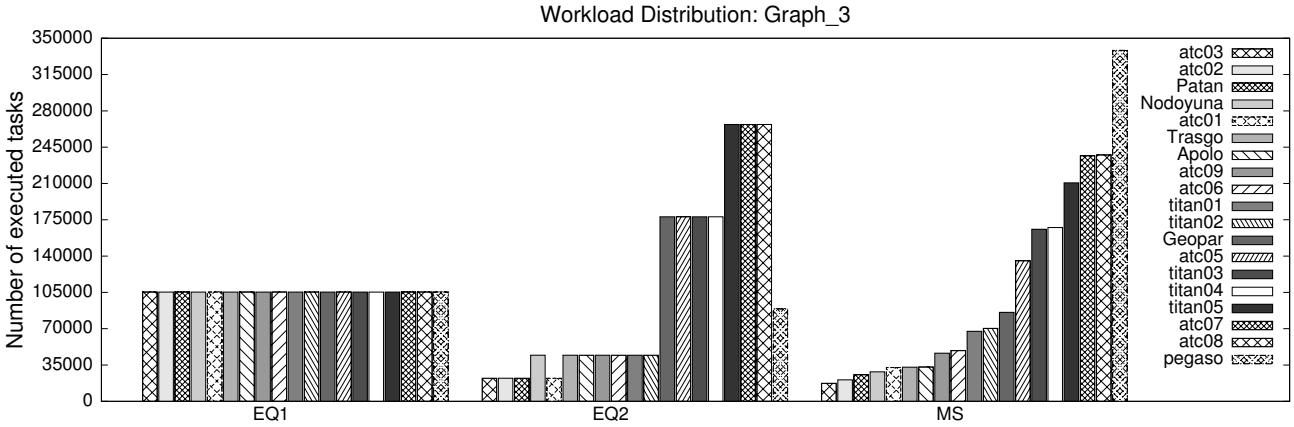

Figure 15. Number of executed tasks per cluster node with different distribution policies in the big heterogeneous cluster.

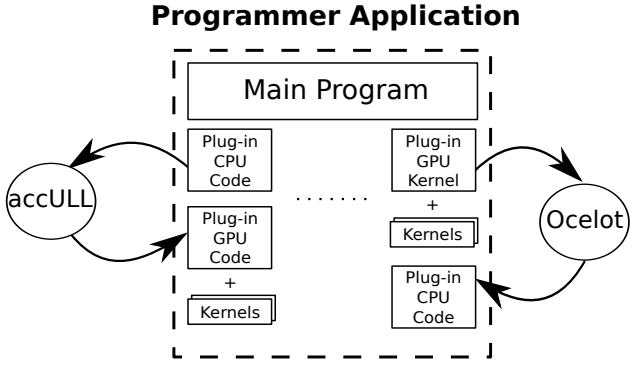

Figure 17. Usage of TuCCompi with code-transformation modules.

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 [22]. The authors only focus 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 do 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], that is 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 layer for better exploiting GPU computational capabilities.

With respect to sequential-to-parallel code transformation, proposals in this field include accULL [19], 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. 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 working on shared data-structures can be exploited with the current model of TuCCompi.

The use of the 4th and Tuning layers 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 turns out to be very important for heterogeneous environments with a high presence of these GPU devices.

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 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 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 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

[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] D. B. Kirk and W. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, Feb. 2010. ISBN ISBN: 978-0-12-381472-2, 1.

[12] 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.

[13] 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.

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

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.