Int J Parallel Prog (2015) 43:939–960
DOI 10.1007/s10766-015-0349-6

TuCCompi: A Multi-layer Model for Distributed Heterogeneous Computing with Tuning Capabilities

Hector Ortega-Arranz · Yuri Torres · Arturo Gonzalez-Escribano · Diego R. Llanos

Received: 4 July 2014 / Accepted: 3 February 2015 / Published online: 27 February 2015

Abstract During the last decade, parallel processing 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 processing devices, such as CPU-cores and graphics processing units (GPUs). Maximizing the performance of any GPU parallel implementation of an algorithm requires an in-depth knowledge about the GPU underlying architecture, becoming a tedious manual effort only suited for experienced programmers. In this paper, we present TuCCompi, a multi-layer abstract model that simplifies the programming on heterogeneous systems including hardware accelerators, by hiding the details of synchronization, deployment, and tuning. TuCCompi chooses optimal values for their configuration parameters using a kernel characterization provided by the programmer. This model is very useful to tackle problems characterized by independent, high computational-load independent tasks, such as embarrassingly-parallel problems. We have evaluated TuCCompi in different, real-world, heterogeneous environments using the all-pair shortest-path problem as a case study.

Keywords Abstract parallel model · Auto-tuning · CUDA · GPU · Heterogeneous system · HPC framework · MPI · OpenMP

H. Ortega-Arranz (✉) · Y. Torres · A. Gonzalez-Escribano · D. R. Llanos
Departamento de Informática, Universidad de Valladolid, Valladolid, Spain
e-mail: hector@infor.uva.es

Y. Torres
e-mail: yuri.torres@infor.uva.es

A. Gonzalez-Escribano
e-mail: arturo@infor.uva.es

D. R. Llanos
e-mail: diego@infor.uva.es

1 Introduction

Some computing-intensive problems are divided into many independent tasks that can be executed in parallel without requiring any communication among them. They are called embarrassingly-parallel problems [1]. Many real problems are included in this category, such as index processing in web search [2], bag-of-tasks applications [3], traffic simulations [4] or Bitcoin mining [5].

Although the parallelization of embarrassingly-parallel problems does not require a very complex algorithm to take advantage of parallel computing environments, their high amount of computational work requires high performance computing (HPC). Deployment, load balancing, and tasks synchronization details should be tackled by the programmer in a specific way for different applications, and different execution environments. In order to give support to the massive demand of HPC, the last trends focus on the use of heterogeneous environments including computational units of different nature, such as common CPU-cores, graphics processing units (GPUs) and other hardware accelerators. The exploitation of these environments offers a higher peak performance and a better efficiency compared to the classical homogeneous cluster systems [6]. Due to these advantages, and since the cost of building heterogeneous systems is low, they are being incorporated into many different computational environments, from academic research clusters to supercomputing centers.

Despite the wide use of heterogeneous environments to execute massively-parallel problems, there are two issues that limit the usability of these systems. The first one is the lack of computing frameworks that can easily schedule the workload in such complex environments. Some works have been presented to integrate the use of different programming languages or tools [7,8]. However, the programmer still needs to tackle different design and implementation problems related with each level of parallelism. These problems are specially more complex when integrating GPU programming techniques. The second limitation is the lack of a tuning methodology that efficiently unleashes all the power of GPU devices. Although there are languages, such as CUDA, that aim to reduce the programmer's burden in writing parallel applications, it is a difficult exercise to correctly tune the code in order to efficiently exploit all underlying GPU resources. Several studies [9,10] have shown that, in some cases, the values that are recommended by CUDA do not lead to the optimum performance, leaving to the programmers the responsibility of searching for the best values. This search usually implies to carry out several time-consuming trial-and-error tests. There is not 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.

In this paper, we present TuCCompi (Tuned, Concurrent Cuda, OpenMP and MPI), a multi-layer, skeleton-based abstract model, that transparently exploits heterogeneous systems and squeezes the GPU capabilities by automatically choosing the optimal values for important configuration parameters. Moreover, it easily supports the inclusion of work distribution policies as plug-ins. Each layer represents a level of parallelism. The first layer handles the distributed-memory environment, coordinating different shared-memory systems (nodes). The second layer manages the computational units that are inside the nodes. The third layer automatically deploys the execution in the hardware accelerators, such as the 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 a user to transparently take advantage of all computational capabilities of both, CPU-cores and GPU devices, distributed across 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. The experiments have been run in an academic heterogeneous environment.

The contributions of this work are: (a) a multi-layer abstract parallel model that simplifies programming in heterogeneous systems including hardware accelerators, by hiding the details of synchronization, load balancing, and deployment; (b) a prototype implementation that exploits modern GPU capabilities, such as concurrent kernel execution on a GPU, or parameter tuning for GPU execution; and (c) a technique to allow the programmer to supply abstract kernel characterizations of the GPU codes to help the framework to chose optimal values for important CUDA tuning parameters. These optimal values are valid for any current GPU architecture, and are based on the work of [10]. Experimental work with the prototype framework shows that the new abstraction layers easily allow to obtain performance improvements of up to 12 % in the test case, with minimum extra programming effort, compared with using only the traditional three first ones.

The rest of this paper is organized as follows. Section 2 describes some related work. Section 3 introduces our conceptual approach. Section 4 describes the use of the model through some code snippets. Section 5 shows the internals of the TuC-Compi framework. Section 6 explains the case study used. In Sect. 7 we present the experimental environment and the results obtained. Finally, Sect. 8 summarizes our conclusions and describes the future work.

2 Related Work

There are several works that integrate languages on tools to consider several levels of parallelism. llCoMP [7] 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 he wants to parallelize. The authors are only focused in parallel-loop problems. This compiler does not support the joint 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 [8] propose a framework called OMPICUDA to develop parallel applications on hybrid CPU/GPU clusters by mixing OpenMP, MPI and CUDA models. This framework presents some limitations: it cannot be easily modified to support a new parallel model, and it is not consider any policy to select proper values of CUDA configuration parameters. Another parallel programming approach using hybrid CUDA, MPI and OpenMP programming is presented in [11]. The authors focus on the model to solve iterative problems, and they do not take into account any

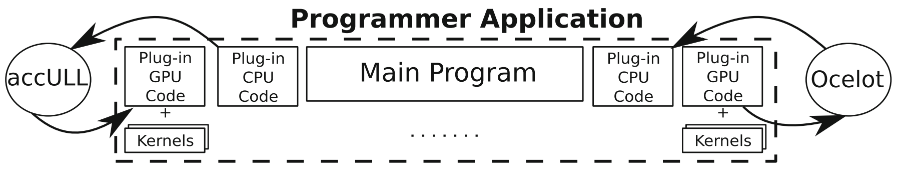


Fig. 1 Usage of TuCCompi with code-transformation modules

generic CUDA optimization technique. It does not support any mechanism to include new load distribution policies.

The authors in [12] 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, nor any automatic mechanism to efficiently exploit heterogeneous environments.

Other programming libraries for hybrid architectures supporting GPUs are SkelCL [13], StarPU [14] and SkePU [15]. The first tries to enhance the OpenCL interface in order to coordinate different GPUs of the same shared-memory machine. However, it does not support load distribution between GPUs of different machines, or even, other computational units of different nature, such as the CPU-cores. These limitations are not present in StarPU and SkePU, but they do not support the exploitation of the concurrent-kernels feature of modern GPUs. StarPU does not even consider the use of tuning techniques for better exploiting GPU capabilities. SkePU tries to find the optimal threadblock size by automatically checking all possibilities using trial-and-error executions, but it does not provide a model for tuning this parameter.

There are other works that aim to transform sequential code to parallel code, and vice-versa. For example, accULL [16] receives a sequential code and automatically transforms it to parallel GPU code. Another example of code transformation is Ocelot [17], that works in the opposite way. Given a GPU implementation, Ocelot transforms it to sequential code. TuCCompi model does not aim to deal with code transformations, but these works can be easily attached as previous functional modules to our multilayer model (see Fig. 1). Another attachable module could be the work of elastic kernels presented in [18]. They do manual source-to-source code transformations in order to obtain GPU kernels that exploit more the multikernel feature of the GPU devices.

3 TuCCompi Architecture

TuCCompi integrates several execution layers with different coordination mechanisms, that are abstracted to provide an unified view of the computing heterogeneous system to the programmer. He has to program his applications in two programming levels: (1) a *coordination level*, that abstracts the work distribution across the computational units inside the distributed shared-memory nodes; and (2) a *deployment level*, that abstracts the management of computational unit of different nature. This

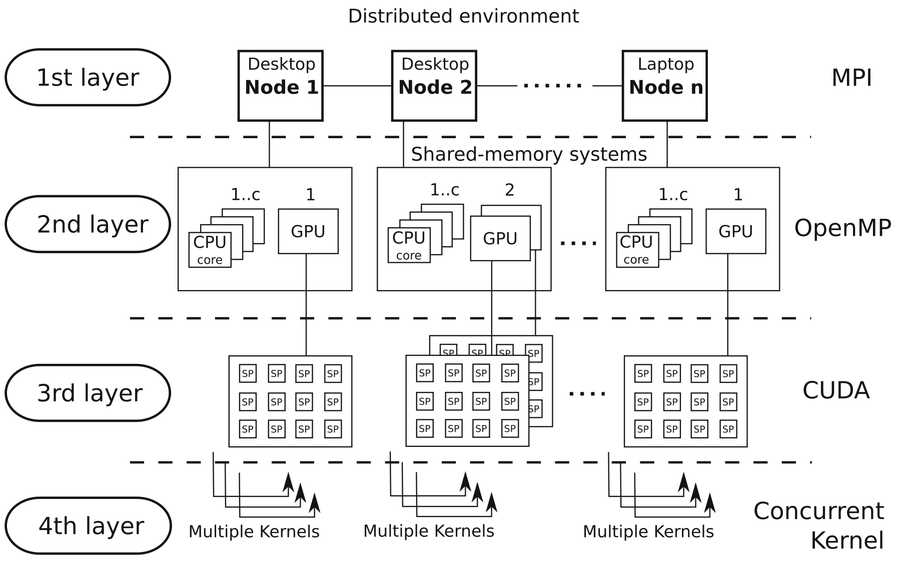


Fig. 2 Layer deployment of TuCCompi model in a heterogeneous cluster

section gives a description of these different layers defined in our model. A graphical representation is depicted in Fig. 2.

3.1 The 1st Layer (Distributed Environment)

Nowadays, one of the most economic ways to assemble a heterogeneous system is to interconnect a set of different individual machines, also called nodes, such as personal computers, laptops, complex virtual host machines, or even other supercomputing systems composed in turn by other machines. It is necessary to apply communication and synchronization mechanisms in order to coordinate these nodes. The first layer of TuCCompi (see Fig. 2) is responsible of managing this node coordination without taking into account the hardware details and features of each machine. This layer is abstracted at the *coordination level*, allowing the programmer to skip thinking in terms of more complex message-passing models.

3.2 The 2nd Layer (Shared-Memory Systems)

Nodes are nowadays composed by several processing units 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. In this layer of TuCCompi we use the concept of "computational unit" for any CPU-core or device hosted in a node. This second layer is responsible of the coordination of all computational units inside the node. For the programmer's point of view, this layer is also encapsulated in the abstraction of the *coordination level*. It also hides the fact that each special device is controlled by a dedicated thread that executes a different code. The programmer sees all devices and CPU-cores in an homogeneous form.

3.3 The 3rd Layer (GPU Devices)

This layer implements the abstraction used at the *deployment level*. It is the responsible of the coordination and deployment actions needed for special devices, such as GPUs, FPGAs, or Xeon Phis, in an homogeneous form. This is done by hiding the details needed to manage different address spaces, offloading codes, etc.

3.4 The 4th Layer (Concurrent GPU Kernel Execution)

The most recent NVDIA GPUs support concurrent-kernel execution [19], where different kernels of the same application context can be executed on a GPU at the same time. This feature is very helpful when kernels that use just few resources are launched, allowing a concurrent execution of other kernels, and thus, exploiting at the same time all resources of the device. Although at first glance this feature seems to be profitable only when low resource-consuming kernels are launched, the concurrent execution of higher resource-consuming kernels also gives performance gains. This occurs because several kernels of the same application context work on the same memory areas taking advantage of the L1 data-cache, originating less number of cache-misses and therefore alleviating the global memory bottlenecks. The programmer provides a parameter to define the number of tasks that will be concurrently deployed in a single GPU for each application. This layer internally take cares of the synchronization of the concurrent kernel launching. It contributes to the functionalities encapsulated in the *deployment level*.

3.5 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 of an appropriate threadBlocks size and shape, the coalescing maximization of the memory accesses, or the occupancy maximization of the Streaming Multiprocessors, 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 it even more difficult to find the optimal configuration for a given GPU. Besides this, the optimal values also depend on the memory access pattern and the characteristics of the code of each executed kernel. This layer allow the programmer to supply to the *deployment level* with an abstract kernel characterization of the CUDA codes in terms of human-understandable features. With these values, the model internally chooses proper values for the execution parameters. This solution opens the possibility to integrate techniques to automatically analyze and characterize the CUDA kernel codes for specific GPU devices.

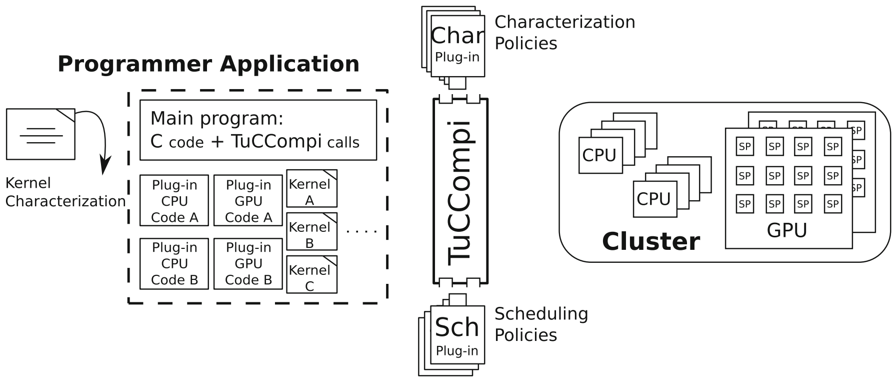


Fig. 3 TuCCompi model usage. Elements in the *dashed box* are provided by the programmer. Note that the user can develop different versions of each plug-in (Code A, Code B, …) but only one at a time will be deployed into TuCCompi framework

4 TuCCompi Model Usage

To build a program using TuCCompi, a programmer should provide the following elements (see Fig. 3): (1) *coordination level*, implemented as a main C language program with the TuCCompi primitives and macros, and (2) *deployment level*, including the sequential-CPU and the parallel-GPU specific codes for each application, named as `PLUG-IN_CPU` and `PLUG-IN_GPU` respectively, and characterizations of the accelerator kernel codes.

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

4.1 *Coordination Level*: TuCCompi Main Program Implementation

Figure 4 shows an example of the code that the user has to implement in order to start and control the execution. The primitive `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 the application. Line M03 shows the primitive needed to set the number of kernels that the GPU devices will execute concurrently (information for the 4th execution layer). Line M04 shows the primitive used to initialize and execute the functions implemented in the corresponding plug-ins. This synchronization expression transparently executes the CPU-plugin code for the CPU-cores, or the specialized GPU-plugin code for the GPU devices, using the same semantics, across a whole heterogeneous cluster. The first parameter of this macro represents the kind of scheduling policy desired by the user (described below). It is used internally by the 1st and 2nd execution layers to balance the workload across the different computational units. Line M05 shows the primitive needed to make the

```
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
```

Fig. 4 User implementation of the TuCCompi main-program. The programmer has to add to his code the *boxed* primitives

process wait until all node computational units have finished. The user is free to insert more code to execute other kernels, before the finalization of the heterogeneous cluster communication, shown in line M07.

4.2 *Coordination Level*: Workload Scheduling

The TuCCompi model includes three different policies to distribute the workload between all available cluster resources through the first parameter of the M04 primitive.

The first one, `EQ1`, is an equitable policy that schedules the same number of tasks to each node of the 1st layer (distributed memory environment), independently of the number of CPU-cores, GPUs, or other accelerators that the nodes have inside. Later, each node equally divides the assigned workload between all its own computational units (CPU-core/Accel.), also in a balanced way.

The second one, `EQ2`, is also an equitable policy, but it divides the workspace straight between the computational units of the whole cluster at the 2nd layer. The workspace division does not consider the computational unit nature.

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 when they become idle, until the master sends a termination signal to them. Thus, the more powerful units will ask for more work, and therefore they will process more tasks than the less powerful units. As the master can be located at any cluster node, these asking-for-tasks requests are issued through distributed-environment communications.

Additionally, TuCCompi also offers the possibility of including a scheduling policy programmed by the user through the *scheduling plug-in* (see Sect. 5.5.1).

4.3 *Deployment Level*: User-Code Plug-Ins

Figure 5 (top) 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
```

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

Fig. 5 Plugin_Cpu (*top*) and Plugin_Gpu (*down*) interfaces. The programmer adds to his code the *boxed* arguments to deploy the Cpu plugin in TuCCompi, and he has to replace the CUDA kernel launch primitives for the *boxed* TuCCompi macros for the GPU plugin

Figure 5 (bottom) shows the code that will be executed in a CPU thread to manage one or more associated GPUs. The control of the GPU often involves active waits. In this case, a CPU-core should be sacrificed to execute this GPU-controller thread. 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 responsible of launching the corresponding kernels. 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 indicated by the programmer in the main control program (MK value) (see line M03 of Fig. 4). Every concurrent kernel launched will need its own workspace to compute its results. The second primitive of line G02 gives to the kernel one memory pointer for each data structure needed. The needed parameters are: The variable name; the native type of the 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 G04). The TuCCompi primitive of line G03 forces the CPU to wait for the finalization of an executing kernel, or kernels concurrently launched, providing a synchronization mechanism.

4.4 *Deployment Level*: Kernel Characterization

The user has to provide a general characterization of his kernels along with 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 will automatically optimize the use of the underlying hardware of any kind of GPU found in the platform, following the guidelines and optimizations proposed in [10] for each possible combination of these parameters.

Table 1 TuCCompi kernel-characterization classification

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 memory data reutilization compared to the number of arithmetic instruction per thread	`high`/ `medium`/ `low`/ `def`

The `def` choice can be used when the user does not know the kernel characterization

```
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: }
```

Fig. 6 Kernel characterizations and implementations. The programmer adds the *boxed* primitive before the kernel implementation to characterize it

Figure 6 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 class chosen from the classification criteria described in Table 1. In the case that the user does not know how to classify his kernels, he 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)`.

5 TuCCompi Internals

In this section we will discuss the internals of the TuCCompi framework. The functions and primitives described here have a correspondence with the model layers described in Sect. 3.

5.1 Cluster Inter-Node Communication (1st Layer): `TuCCompi_COMM`

Once a TuCCompi program is in execution, each process initializes its MPI-identification variables, and enters into a global communication step carried out by

Fig. 7 Implementation of the `comm()` recognition function, called from `TuCCompi_COMM()`

```
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<n_proc; 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<n_proc; 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:  }
```

exchanging a few MPI messages. An arbitrary process is the coordination handler, that we name as parent process. It receives from the remaining processes the number of the computational resources they are able to manage. Then, the parent process sends to each process a global identification number for each resource inside the whole heterogeneous cluster. Additionally, the parent process sends more information about the heterogeneous cluster, such as the total number of computational units and the numeration per node.

Figure 7 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 of the MPI process that coordinates the communication. 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 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.

For problems where not all the input data is needed, and just the required one is wanted be sent from the parent process, the user just has to slightly modify this macro in order to obtain this desired behavior of the framework.

5.2 Cluster In-Node Synchronization (2nd Layer): `TuCCompi_PARALLEL`

Once the TuCCompi model has been initialized and the user variables have been defined, the `TuCCompi_PARALLEL` primitive automatically creates as many

```
00: #define TuCCompi_PARALLEL(MS, pluginCPU, pluginGPU)\
01:   cudaGetDeviceCount(&TuCCompi_gpuCount);\
02:   omp_set_num_threads(omp_get_num_procs());\
03:   #pragma omp parallel\
04:   {\
05:     int task;\
06:     int TuCCompi_local_id  = omp_get_thread_num();\
07:     int TuCCompi_global_id = v_id[id_mpi] + TuCCompi_local_id ;\
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();
```

Fig. 8 `TuCCompi_PARALLEL()` and other macro-definition codes

OpenMP threads as the number of CPU-cores that will perform the parallel execution. Figure 8 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 not shown due to space restrictions). The master–slave implementation just divides the workload between the cluster nodes and the computational units. The slaves execute each task without needing any more communication with the master. Lines 05–07 initialize the intra-node 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 is governed by its corresponding CPU-core. Therefore, lines 10–15 obtain the device properties, entering into the ask-for-tasks working loop and executing the parallel GPU code provided by the pluginGPU (see Sect. 4.3). The normal CPU-cores also enter into the ask-for-tasks working loop but executing the code of the pluginCPU (see Sect. 4.3) (lines 16–18).

5.3 Kernel Launch and Concurrent Kernel Execution (3rd and 4th Layers): `TuCCompi_GPULAUNCH`

Before the task-threads spawn (Line 03 of Fig. 8), the first layer (distributed-memory process) consults how many GPUs are available in the shared-memory node (Line 01 of Fig. 8). Once in the parallel region, an OpenMP thread is assigned to one CPU-core in order to govern each hardware accelerator, also storing some relevant properties of the GPU, such as its architecture (Lines 11–13

```
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()
```

Fig. 9 Declarations for the automatic kernel launch and multikernel support

of Fig. 8). Afterwards, this thread is the responsible of handling the logic control of the algorithm implemented in pluginGPU, actually launching the different kernels invoked through the primitive `TuCCompi_GPULAUNCH(`*`kernel_name, input_size, kernel_vars`*`)`, whose definition is shown in Fig. 9.

The model automatically detects if the concurrent execution of several kernels (the multikernel feature) is supported by the GPU 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. 9). Additionally, in order to make possible that each kernel works in a different workspace, the `PARLLMK(`*`variable_name, variable_type, variable_length`*`)` macro automatically computes the memory offset allocation of the corresponding variables that are task-dependent (Lines 04–05 of Fig. 9).

5.4 Automatic Kernel Tuning (Tuning Layer): `TuCCompi_KERNELCHAR`

The optimization layer automatically configures the kernel parameters depending on: (1) The GPU architecture where it 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 [10]. The authors 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 characterized and classified a wide range of kernel types, also presenting the optimal configurations for them. As previously discussed in Sect. 4.4, we have implemented a classification based on the previously cited work.

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 previous example of Fig. 6), and automatically mapped to a structure that contains the optimal values for all classified architectures (see Fig. 10). As can be seen in lines 02–03 of Fig. 9, these values are already embedded in the primitive of kernel launching as a call to the `t_grid()` function, that returns the optimal number of blocks, and `t_threads()`, that returns the optimal number of threads per block. In this way,

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

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

#define k_defdefdefdef {256, 256, 256, 256}
#define k_scatterlowhighlow {256, 256, 96, 64}
#define k_coalescedlowlowmedium {256, 128, 192, 128}
#define ...
```

Fig. 10 Some declaration examples for the automatic GPU kernel optimizations

TuCCompi automatically selects the optimal configuration of the threadsBlock size–shape.

If the user does not know how to characterize his kernel, the default values can be used. These values are recommended by CUDA [20], to maximize the SM Occupancy. Although these recommended values sometimes work well, we will see that there could be performance differences of more than ten percent compared to the optimal values.

5.5 Advanced TuCCompi Model Features

TuCCompi model has additional functionalities and features, such as the possibility of executing a more complex workload scheduling policy created by the user, or the possibility of changing the optimal values for each kernel and GPU. We will now describe two plugin systems that help with these functions.

5.5.1 Scheduling Plug-In System

The current master–slave policy involved in the prototype gives a simple implementation where only one task is scheduled to each slave independently of its computational power. The master and the slaves execute, respectively, the master-function and slave-function codes provided in the scheduling plug-in. Additionally, if the problem or the user need a different granularity or a particular load distribution that follows a special pattern or policy, the model allows the programmer to use his own scheduling implementation. This is done by injecting new distribution policies through a *scheduling plug-in* system, using an extended primitive `TuCCompi_PARALLEL(MS, pluginCPU, pluginGPU, pluginMASTER, pluginSLAVE)`.

This 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 of a single one. When an OpenMP thread responsible of a GPU device asks for tasks, it is able to retrieve the corresponding device information that could be sent to the master in the requesting message. With this information, the master could give a pack of tasks to the most powerful devices and a single one to the less powerful computational units. Thus, the master can produce a more complex distribution depending on the capabilities of the computational units that

```
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(END_SIGNAL, 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(END_SIGNAL, id_slave);
20:  token++;
21: }
22:}


23:  int slave(id_slave, mpi_master, tag){
24:    SEND(id_slave, mpi_master, tag);
25:    RECV(task, mpi_master, id_slave);
26:    return task;
27:  }
```

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

Fig. 11 Our case-study implementation for the functions, master and slave (*left*), of the distribution plug-in. Case-study user implementation for pluginGPU (*right*)

are asking for work. Figure 11 shows a customized implementation of the *scheduling plug-in* created for the case study.

5.5.2 Characterization Plug-In System

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.

6 Case Study

In order to illustrate the capabilities of the TuCCompi framework prototype, we have chosen the 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 approach for the first three layers. Besides, the GPU solution for this problem involves three kernels of very different nature, and characterization. This variety allows us to check the behavior of the fourth and tuning layers.

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

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

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

6.1 The 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 a 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 [21]. Crauser et al. in [22] introduces an enhancement that tries in each iteration i to augment the threshold Δ_i as much as possible to process more nodes in the next iteration.

6.2 Plug-Ins Used for Our Case Study

Scheduling Plug-In: Each SSSP computation is a single independent task. We have slightly modified the naive master–slave behavior in order to show how easily is to customize the scheduling plug-in, see Fig. 11 (left). 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. This implementation sends MK tasks to each modern GPU, and only one for the Pre-Fermi architectures and the CPU cores.

Figure 11 (left) shows the master (lines 00–22) and slave (lines 23–27) implementations. 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 identifier of the first task of 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), thus, the task counter is

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

simply incremented. When all tasks have been scheduled and carried out, the master sends the termination signal to the rest of active slaves when they request more tasks (lines 17–21).

Regarding the slave implementation, it first notifies the master that it is idle (line 24). Then the slave receives the identifier of the task pack to be executed, 1 task for CPU-cores and Pre-Fermi GPUs, and MK tasks for the modern GPUs in this prototype (line 25).

SSSP Plug-Ins: Both the CPU-core sequential and the parallel GPU codes are implementations of the Crauser algorithm. Their implementation for this problem has been taken from [23]. Algorithm 1 shows the GPU parallel pseudo-code of Crauser's algorithm. Figure 11 (right) 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. 4.4, the kernels are characterized in Table 2.

7 Experimental Evaluation

This section describes the methodology used to test the TuCCompi prototype, the platforms used, and the input set characteristics for the case study (the APSP problem). Finally, the experimental results and a discussion are presented.

7.1 Methodology

In order to evaluate TuCCompi for heterogeneous environments, we have tested the APSP problem as a case study (see Sect. 6) in different scenarios. Each scenario was designed with the aim to check the use of the layers involved in an incremental fashion. Architecture details are shown in Table 3: (1) a single GPU, that uses the 3rd, 4th, and the tuning layer; (2) two GPUs, that involve the 2nd layer in addition to the previous ones; (3) *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; (4) *small HC*: small heterogeneous cluster, that uses all layers of TuCCompi; and (5) *big HC*: big heterogeneous cluster to evaluate the scalability of the model.

We set the parameter of the concurrent kernel execution to four (MK = 4). The workload scheduling used for the scenarios described below is the customized master–slave policy presented in Sect. 6.2. Note that the behavior of the equitable policies, for our heterogeneous scenarios would result in a bottleneck of the slowest node

Table 3 Description of the components that compound the heterogeneous clusters (HCs)

Node	CPUInfo	#CPU-cores	GPU details
Small heterogeneous cluster (small HC)			
Pegaso	IC2 i7 960 3.20 GHz	8	GeForce GTX 480 + GeForce GTX 680
Nodoyuna	IC2 Q8200 2.33 GHz	4	–
Trasgo/Apolo	IC2 Q6600 2.40 GHz	4/4	–
Geopar	IX E7310 1.6 GHz	16	–
Patan	IC2 E6550 2.33 GHz	2	–
Atc01/02	IC2 6300 1.86 GHz	2/2	GeForce 9600GT/–
Atc03	AMD AtX23600+	2	GeForce 8500GT
Atc09	IC Q8299 2.33 GHz	4	–
Big heterogeneous cluster (big HC): small HC plus the following machines			
Titan01/02/05	IX E5-2620 2.00 GHz	4/4/12+12	–
Titan03/04	IX E5645 2.40 GHz	8+8/8+8	–
Atc05/06	IX E5630 2.53 GHz	8+8/4	–
Atc07	IX X-5675 3.07 GHz	12+12	–
Atc08	IX E5-2620 2.00 GHz	12+12	–

whereas the rest are idle. Table 3 describes the heterogeneous platforms used for our experiments. For each node, we indicate the number of CPU-cores and GPUs. The nodes run Ubuntu Desktop 10.04 OS, with CUDA 4.2 and driver 295.41. The Big HC contains a total of 180 CPU-cores and 4 GPUs. However, each GPU device is governed by a single CPU core, thus, the total number of real computational units is 180 (176 CPU-cores plus 4 GPUs). The multi-GPU system includes the 2 GPUs of the *Pegaso* machine. The single GPU scenario uses the fastest of them, the GTX 480.

Finally, with the aim of testing the performance gain offered by the proposed 4th and Tuning layers, we have compared the execution of a single GPU connecting or disconnecting the optimizations introduced by these layers. For the non-automatically optimized versions (without 4th and Tuning layers), we have chosen some of the optimal values recommended by CUDA that maximize the GPU occupancy executing a single kernel at a time.

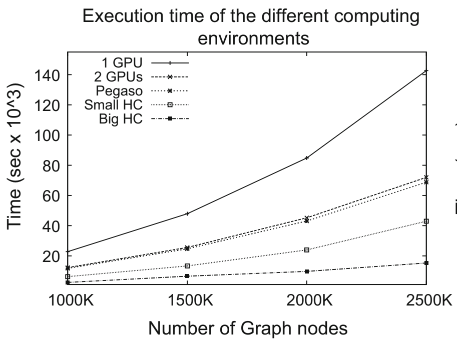
7.2 Input Set Characteristics

The input set is composed of a collection of graphs randomly generated by a graph-creation tool used by [24] in their experiments. The graph generation method leads to irregular loads when applying individual SSSP searches. The graphs are stored in standard CSR format, and the edge weighs are integers that randomly range from 1 . . . 10. 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. The experiments have been carried out just computing enough task sets (1024, 2048, 4096, 8102, 16,204, and 32,408) to produce sufficient computational load to keep scalability in all scenarios.

7.3 Experimental Results

7.3.1 GPUs Versus the Heterogeneous Environments

Figure 12 (left) 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. Moreover, the use of this model has a communication overhead across nodes lower than 1 %. In the Small-HC scenario, this overhead has never surpassed 0.589 % of the total execution time. Figure 13 shows the experimental distribution of tasks per cluster node using the `MS` scheduling policy, compared with the theoretical values that `EQ1` and `EQ2` static policies would obtain.



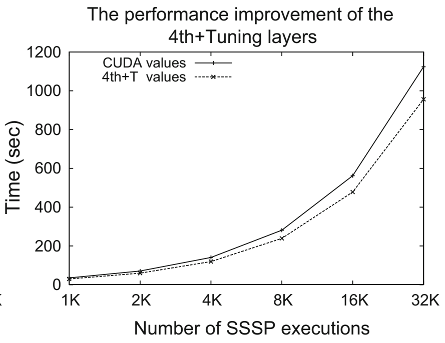


Fig. 12 Execution times of the tested scenarios for different graph-sizes (*left*). Performance improvements obtained by the 4th and Tuning layers with respect to CUDA recommended configuration values (*right*)

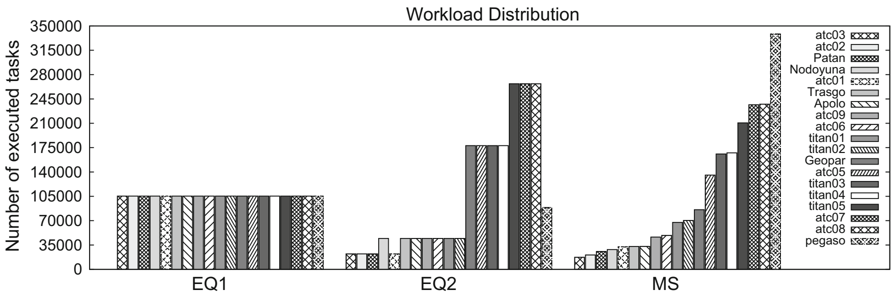


Fig. 13 Number of executed tasks per node of the Big HC with the three scheduling policies

7.3.2 The 4th and Tuning Layers Performance Gain

Figure 12 (right) shows the comparison of the concurrent kernel execution, with MK = 4, combined with the values proposed in [10], with respect to one of the CUDA recommended values for each kind of APSP kernel on the GPU GeForce GTX 480, with only one kernel per time. The use of the concurrent kernel layer and the optimization tuning reduces the execution time for our test case up to 12 %.

8 Conclusions and Future Work

In this paper we propose TuCCompi, a multilayer abstract 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 TuCCompi model.

Compared with previous works, TuCCompi adds a novel parallel layer to the traditional parallel dimensions, with the automatic execution of concurrent kernels in a single GPU. Additionally, it squeezes even more the computational power of the GPUs by applying optimal values for runtime configuration parameters, such as the threadblock size. For our test case, the use of these both new layers leads to performance improvements of up to the 12 %. Thus, these new layers turn out very significant for heterogeneous clusters with GPUs.

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 optimal configurations of different GPU architectures, without making any change in the model. The use of this model exploits even the less powerful devices of a heterogeneous cluster, and it correctly scales if more computational units are added to the environment, with a communication overhead less than one percent of the total execution time.

Our future work includes the implementation and testing of new scheduling plugins for new kinds of applications, also including problems with data-dependencies, and for specific data partition and data distribution schemes, needed in problems with larger input data sets. Regarding the concurrent kernel layer, we plan to incorporate an optional autotuning behavior that allows the framework to find the optimal number of kernels to be deployed during the execution.

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 Economía y Competitividad and ERDF program of the European Union: CAPAP-H5 network (TIN2014-53522-REDT), MOGECOPP project (TIN2011-25639); Junta de Castilla y León (Spain): ATLAS project (VA172A12-2); and the COST Program Action IC1305: NESUS.

References

1. Foster, I.: Designing and building parallel programs: concepts and tools for parallel software engineering. Addison-Wesley Longman Publishing Co., Inc., Boston (1995)
2. Hoelzle, U., Barroso, L.A.: The datacenter as a computer: an introduction to the design of warehouse-scale machines, 1st edn. Morgan and Claypool Publishers, San Rafael (2009)
3. Cirne, W., Paranhos, D., Costa, L., Santos-Neto, E., Brasileiro, F., Sauve, J., Silva, F.A.B., Barros, C., Silveira, C.: Running bag-of-tasks applications on computational grids: the mygrid approach. In: Proceedings of international conference on parallel processing (ICPP 2003), pp. 407–416 (2003)
4. Mangharam, R., Saba, A.A.: Anytime algorithms for GPU architectures. In: Proceedings of the 2011 IEEE 32nd real-time systems symposium, RTSS '11, pp. 47–56. Washington, DC, IEEE Computer Society (2011)
5. Taylor, M.: Bitcoin and the age of bespoke silicon. In: Compilers, architecture and synthesis for embedded systems (CASES), 2013 international conference on, pp. 1–10 (2013)
6. Brodtkorb, A.R., Dyken, C., Hagen, T.R., Hjelmervik, J.M., Storaasli, O.O.: State-of-the-art in heterogeneous computing. Sci. Program. **18**(1), 1–33 (2010)
7. Reyes, R., de Sande, F.: Optimization strategies in different CUDA architectures using llCoMP. Microprocess. Microsyst. **36**(2), 78–87 (2012)
8. Liang, T., Li, H., Chiu, J.: Enabling mixed openMP/MPI programming on hybrid CPU/GPU computing architecture. In: Proceedings of the 2012 IEEE 26th international parallel and distributed processing symposium workshops & PhD forum (IPDPSW), pp. 2369–2377. IEEE, Shanghai (2012)
9. Torres, Y., Gonzalez-Escribano, A., Llanos, D.: 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, pp. 617–624 (2012)
10. Torres, Y., Gonzalez-Escribano, A., Llanos, D.R.: uBench: exposing the impact of CUDA block geometry in terms of performance. J. Supercomput. **65**(3), 1150–1163 (2013)
11. Yang, C., Huang, C., Lin, C.: Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters. Comput. Phys. Commun. **182**, 266–269 (2011)
12. Howison, M., Bethel, E., Childs, H.: Hybrid parallelism for volume rendering on large-, multi-, and many-core systems. Vis. Comput. Graph. IEEE Trans. **18**(1), 17–29 (2012)
13. Steuwer, M., Gorlatch, S.: SkelCL: enhancing OpenCL for high-level programming of multi-GPU systems. In: LNCS, ser, Malyshkin, V. (eds.) Parallel computing technologies, p. 258272. Springer, Berlin (2013)
14. Hugo, A.-E., Guermouche, A., Wacrenier, P.-A., Namyst, R.: Composing multiple starPU applications over heterogeneous machines: a supervised approach. In: Proceedings of IEEE 27th IPDPSW'13, pp. 1050–1059. Washington, USA: IEEE, (2013)
15. Dastgeer, U., Enmyren, J., Kessler, C. W.: Auto-tuning SkePU: a multi-backend skeleton programming framework for multi-GPU systems. In: Proceedings of the 4th IWMSE, pp. 25–32. New York, NY, USA: ACM, (2011)
16. Reyes, R., López-Rodríguez, I., Fumero, J.J., de Sande, F.: accULL: an OpenACC implementation with CUDA and OpenCL support. In: Proceedings of the 18th conference on parallel processing, ser. EuroPar'12, pp. 871–882. Springer, Berlin (2012)
17. Farooqui, N., Kerr, A., Diamos, G.F., Yalamanchili, S., Schwan, K.:A framework for dynamically instrumenting GPU compute applications within GPU Ocelot. In: Proceedings of 4th workshop on GPGPU: CA, USA, 5 Mar 2011. ACM, p. 9 (2011)
18. Pai, S., Thazhuthaveetil, M.J., Govindarajan, R.: Improving GPGPU concurrency with elastic kernels. SIGPLAN Not. **48**(4), 407–418 (2013)
19. NVIDIA.: NVIDIA CUDA programming guide 6.0, (2014)
20. Kirk, D. B., Hwu, W.W.: Programming massively parallel processors: a hands-on approach, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2010)
21. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik **1**, 269–271 (1959)
22. Crauser, A., Mehlhorn, K., Meyer, U., Sanders, P.: A parallelization of Dijkstra's shortest path algorithm. In: LNCS, ser, Brim, L., Gruska, J., Zlatuška, J. (eds.) Mathematical foundations of computer science 1998, pp. 722–731. Springer, Berlin (1998)

23. Ortega-Arranz, H., Torres, Y., Llanos, D.R., Gonzalez-Escribano, A.: A new GPU-based approach to the shortest path problem. In: High performance computing and simulation (HPCS). international conference on 2013, pp. 505–512 (2013)
24. Martín, P., Torres, R., Gavilanes, A.: CUDA solutions for the SSSP problem. In: LNCS, ser, Allen, G., Nabrzyski, J., Seidel, E., van Albada, G., Dongarra, J., Sloot, P. (eds.) Computational Science: ICCS 2009, pp. 904–913. Springer, Berlin (2009)