Automatic Code Generation and Adaptive Grid Scheduling for GPU Cluster Computing

Xuyuan Jin

Supervised by Prof. Henk Corporaal and Cedric Nugteren
Eindhoven University of Technology, The Netherlands
x.jin@student.tue.nl

Abstract

Recent advances in GPUs (graphics processing units) lead to massively parallel hardware that is easily programmable and widely applied in areas which require intensive computation besides graphics acceleration. The appearance of GPU clusters gains popularity in the scientific computing community, and the study on GPU clusters becomes an increasingly hot issue. While extending a single-GPU system to a multi-GPU cluster, the workload is spawned onto a number of GPU devices, and device-level and machine-level parallelism are achieved on top of the native SIMD thread-level parallelism. This requires a programming model extension and a workload scheduling strategy, but currently most of the programmers have to perform the extension manually and schedule the workloads in a naive way. Analytical performance models of GPUs exist, however, prediction of the performance of a GPU cluster is more complicated and no dedicated study has been done on this topic. In this paper, programming model is extended to fit the GPU cluster and propose a tool for automatic code generation and adaptive workload scheduling with which the application gains a 3x speedup from the cluster compared with executed on a single GPU. Besides, results show that with an extended analytical performance model, the performance of a GPU cluster is predictable.

Keywords Code generation, GPU cluster, CUDA programming model, adaptive scheduling, analytical performance model

1. Introduction

GPUs (Graphics processing units) have rapidly evolved to become high performance accelerators for data parallel computing because of their massive thread-level parallelism architecture and high performance/price ratio. The high peak computing power of GPUs make them capable of executing the costly algorithms which are formerly only practicable with FPGAs. On the other hand, the development of programming languages for GPUs, e.g. CUDA and OpenCL, provides great convenience for programmers to write parallel applications. The massive parallel hardware architecture and high performance floating point arithmetic and memory operations of GPUs make them increasingly applied in the general purpose computation area.

Because of its potential to significantly reduce price, space, power, and cooling demands per FLOP, GPUs appear to be a cost-effective HPC (High Performance Computing) accelerator and are widely used in high performance computing clusters [12]. Although the mentioned advantages can be conferred in using GPUs as accelerators in HPC clusters, application development flow, job scheduling and resource management become the new challenges for the HPC community. For example, applications designed to run on a single GPU need to be rewritten in order to map to the GPU cluster, and resource management mechanism needs to be developed to make use of the cluster resource efficiently.

To better understand the performance bottleneck of a parallel program, an analytical performance model is increasingly used. It provides insights into the performance bottleneck of parallel applications on GPU architectures. The programmer can then tune their code targeting the bottlenecks without exhaustive design space exploration. Besides, with an analytical model of a GPU cluster, we can predict the performance of a cluster, which can be helpful to give suggestions in purchasing hardware and creating suitable network connections according to the characteristics of applications. Analytical performance models exist for single GPU devices. However, the model of a cluster is more complicated and no work has been done on that topic.

Concerning these mentioned issues, following contributions are made in this work:

- Based on the CUDA programming model extension, perform source-to-source code translation generating head and worker node code from the regular CUDA code.
- Optimally schedule CUDA kernels to GPU devices by using an adaptively scheduling algorithm.
- A model for the GPU cluster is established on top of the existing analytical GPU performance model.

The remainder of this paper is organized as follows. In Section 2, the research background and related work are introduced. In Section 3, the experimental setup is presented. The work on automatic code generation is illustrated in Section 4, and an adaptive grid scheduling approach is illustrated in Section 5. In Section 6, the analytical performance model extension for the GPU cluster is discussed. Finally the conclusion is drawn in Section 7.

2. Background and Related Work

This section brings the discussion of the background and related work on the GPU architecture, the GPU cluster hardware and tooling, the CUDA programming model and the MPI communication.

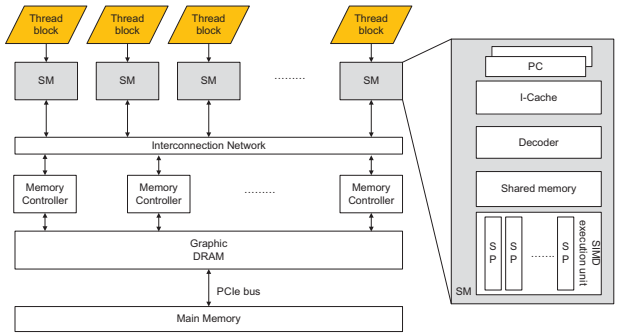


Figure 1. An overview of the GPU architecture

2.1 GPU Architecture

Figure 1 shows an overview of the GPU architecture. The GPU architecture consists a scalable number of streaming multiprocessors (SM), each containing multiple streaming processors (SP) cores, special function units (SFU), a multi-threading instruction fetch and issue unit, and an on-chip shared memory. This multi-threading architecture and the hundreds of SPs enable a GPU to perform massive SIMD (Single Instruction Multiple Data) computation at high performance. For more details about the GPU architecture, we refer to [3],[7] and [6].

2.2 GPU Clusters

Benefiting from device-level parallelism over multiple GPUs, GPU clusters gain increasing popularity in the scientific computation community, and have already been implemented by a number of institutions. For example, the ASTRA group of University of the Antwerp developed a GPU cluster with one host and eight GPUs. It is used to accelerate 3D rendering of medical images with a theoretical peak performance of 5TFlops [2]. At the University of Illinois [14], two GPU clusters are implemented with 32 and 192 nodes respectively for varieties of scientific computations.

2.3 GPU Cluster Tooling

GPU cluster management tools are a kind of software assisting the cluster administrator to manage and monitor the system resource and assigning resources to tasks for the user. Varieties of open source or commercial cluster management tools exist, such as Cordor [1] and PBS (Portable Batch System) [16]. However, like PBS, this kind of management tools only take care of job scheduling on inter-task level, e.g. queueing tasks and setting priorities, and assigning specific devices to a task. This kind of tool is not aware of the intra-task level activities such as grid scheduling in a CUDA application. And those activities still have to be performed by the programmer themselves when developing applications.

2.4 CUDA Programming Model

The CUDA programming model is hierarchical, consisting of thread hierarchy and memory hierarchy. This model ensures that the hardware can benefit from its multi-threading architecture and allows the programmer to parallelize sequential code. The three levels of the hierarchical model are presented as follows, and for more details we refer to [5].

- **Thread level:** A thread is an instance of a kernel. It is executed on an SP.
- **Threadblock level:** A threadblock is a collective of a specified number of threads. It is scheduled to an SM, and within one threadblock, all threads can share data using an on-chip shared memory and can synchronize at a barrier.
- **Grid level:** A grid is the complete execution of a kernel in a GPU device, and all threadblocks within the grid share the off-chip device memory.

To fit the architecture of a GPU cluster, the programming model can be extended to a higher level in both thread hierarchy and memory hierarchy. By Strengert et al., the extension to the CUDA programming language named CUDASA is presented which extends parallelism to a multi-GPU system [19]. On top of the original GPU layer, three abstraction layers are added to scale the model from single device to a multi-node multi-device cluster. In CUDASA, the programmer can design the functionality of the higher levels beyond a GPU with a specific syntax similar to CUDA. The work in this paper is inspired by CUDASA, but in contrast to CUDASA, this work concerns about automatically mapping a single-device application to run on GPU cluster with a programming model extension, which is not possible with CUDASA.

2.5 MPI Communication

MPI (Message Passing Interface) is a standardized and portable message passing system designed for parallel computers based on a distributed memory system. It is widely used as a communication interface in varieties of HPC clusters.

MPI provides a complete set of APIs to perform the thread-to-thread communication, categorized as blocking communication, non-blocking communication and RMA (Remote Memory Access) window communication. The blocking and non-blocking communications need the mutual threads of the communication to take part in communication. When implementing a blocking communication, the thread would not proceed until the communication is finished. Differently, when a non-blocking communication is performed, the thread is allowed to proceed with computation and other communication until it reaches a synchronization request when the communicated material is accessed. Non-blocking communication helps to hide the overhead behind the computation, but it is of lower priority than the blocking communication, and the communication duration is likely to be longer and difficult to be predicted because there might be multiple non-blocking communications are performed in the communication channel simultaneously. With RMA window communication, one thread can directly read or write a certain window area created for the remote memory access of another thread without the participation of the accessed thread, and it provides plenty convenience in program design and development. But synchronization needs to be concerned when communicating in the RMA manner. For more details about MPI, we refer to [10] and [11].

3. Experimental Setup

This study is performed on the existed GPU cluster in Electronic System Group, TU Eindhoven [4]. To give a preliminary understanding of this work, we introduce the two facts in this section: the selection criteria of representative applications and the setup of the existed GPU cluster.

3.1 Application Selection

In this section, representative applications suitable for the cluster are selected by considering the characteristics of the cluster. And Two single-GPU applications are selected to be mapped to GPU cluster:

Sum: Summarize all pixel values of an image. It is a widely used reduction algorithm.

DCT: Perform 8x8 discrete cosine transform of an image. The algorithm is widely used in decoding images and videos.

The GPU cluster works as a distributed memory system, and each node has a complete copy of input data. However, during computation, the memory of each worker node only updates according to the output of the attached device, but not aware of the update from other device. Currently it is not possible to perform inter-block communication between two different devices. So, the application should not contain any inter-block communication. Also it is preferable to choose an application which is computational intensive to cover the costly communication overhead. If the application communicates frequently via the network, it may hardly benefit speedup from the cluster.

The two selected applications perform no inter-block communication, and have a relative higher ratio for computation over the communication. Thus, they are suitable for running on the GPU cluster. The procedure of both applications implemented in CUDA can be written as pseudo code shown in Listing 1. The CPU will first read the input, and perform some CUDA API calls to allocate memory and transfer data to GPU. When the CUDA kernel is launched, the GPU will execute the SIMD instructions in highly parallel. After finishing executing the kernel, the result will be copied back to the CPU side, and the CPU will continue with the sequential part of the application.

```
1  //Start
2  read_input()
3  //start of CUDA session
4  CUDA_allocate_data()
5  CUDA_copy_data(..,HostToDevice)
6  CUDA_start_kernel()
7  CUDA_copy_data(..,DeviceToHost)
8  //end of CUDA session
9  execute_sequential()
10 //End
```

Listing 1. The pseudo code of the selected applications in CUDA implementation.

3.2 Hardware Setup

Here some terms are clarified, which will be frequently referred in following part of this paper.

- **Machine:** A physical desktop which may contain multiple nodes.
- **Head node:** The management machine of the cluster.
- **Worker node:** A CPU-thread GPU pair which is involved in the computation.
- **Global host:** The management CPU thread of the cluster, similar to head node. But when referring as global host, the thread concept other than hardware concept is concerned.
- **Local host:** The CPU thread to which a GPU context is attached.

The GPU cluster consists of GPUs from the vendor of NVIDIA, which are programmable in OpenCL and CUDA. The CUDA 4.0 programming environment is deployed because it is the most widely used and has support for cluster computing. The state-of-the-art NVIDIA GTX570 GPU is applied in the cluster, and each machine is equipped with 4 of these GPUs. The implemented homogeneous cluster consists of 1 head node and 16 worker nodes, which are distributed over 1 head machine and 4 worker machines. The schematic of the cluster is shown in Figure 2. The total theoretical peak performance is 22TFLOPS and the aggregated theoretical GPU memory bandwidth is 2.4TB/s [15][4]. Some specifications of the device and the cluster are listed in Table 1.

	#CUDA cores	Peak perf.	Memory bandwidth
GTX570	480	1.4TFLOPs	152.0GB/s
Cluster	7680	22TFLOPS	2.4TB/s

Table 1. Specification of a GTX570 GPU and the the complete cluster

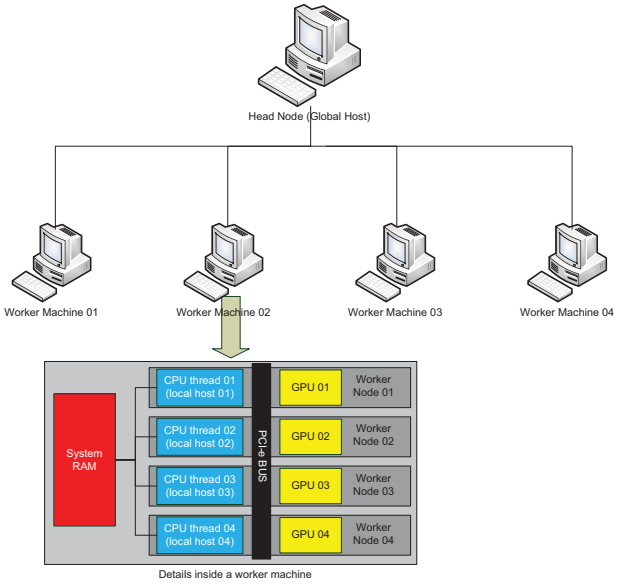


Figure 2. Schematic of the cluster hardware setup

The head node (or global host) is not involved in the computation, but performs as an interface between the cluster user and the worker nodes, taking care of resource management and workload scheduling. The head node is not equipped with a GPU device, but connect to each worker node via Gigabit Ethernet. A worker node is comprised of one local host and one GPU device, and one CPU of a system may behave as multiple local hosts corresponding to the number of GPU contexts.

Considering that the applications do not have massive inter-block communication, one GPU device does not access frequently to the global memory of another one. So, the traffic load is light between worker nodes, and it is tolerable to communicate via the head node. The major traffic of the network is the data transfer between head node and worker nodes. Therefore, the current cluster implementation is in a star topology, where the head node is at the center and all worker nodes connect to it. The connection between worker nodes can be easily established if an application produces heavy traffic load between GPU devices.

4. Automatic Code Generation

In this section, the development process of the automatic code generation tool is presented. First, the concept of a higher level model extended from the CUDA programming model is introduced

to fit the GPU cluster. In the tool implementation, three major phases are performed as shown in Figure 3. They will be discussed in detail in the following sections.

- Separate head node code and worker node code from standard CUDA source code (phase 1)
- Add communication APIs between head node and worker nodes (phase 2)
- Design the management logic of the whole system (phase 3)

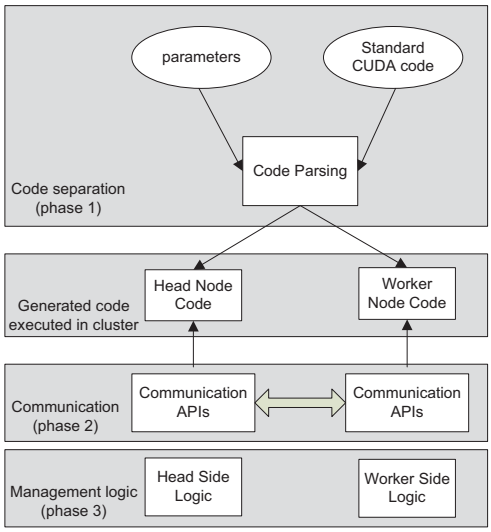


Figure 3. Design flow of tool implementation

4.1 Design Motivation

When a programmer ports a single-GPU application to a GPU cluster, the workload has to be manually distributed over multiple GPUs. And the functionality of the management thread, which is not involved in the actual computation, needs to be designed, e.g. data communication between head node and worker nodes, and the logic of scheduling the grids. This procedure is time consuming and repeatable, and it will provide great convenience if a tool assists with the code generation.

Motivated by this, we develop a tool which inputs a standard CUDA source file, extended it with the maximum number of available GPU devices and the required workload partitioning by the programmer. The outputs of the tool are codes to be executed on the head node and worker nodes respectively. The expected output head node code and worker node code are shown as pseudo code in Listing 2 and 3. Compared with the code shown in Listing 1, the application is split up, that the sequential part is executed by the head node and the parallel part moves to the worker nodes. In addition to the original code, the scheduling logic and communication are performed to enable the cooperation of the head node and worker nodes.

4.2 CUDA Programming Model Extension

As illustrated in Section 2.3, the CUDA programming model consists of three levels: a thread level, a thread-block level and a grid level. In a GPU cluster, the total workload is partitioned by a number of GPUs, in order to achieve parallelism in the device dimension. To support this, higher levels need to be introduced to extend

```
1  //Start of head node
2  read_input();
3  while(scheduling not finished){
4       node_state_query();
5       if(node available){
6            send();
7            wait;
8            recv();
9       }
10 }
11 execute_sequential();
12 //End of head node
```

Listing 2. The pseudo head node code output by the code generation tool

```
1  //Start of worker node
2  if(task assigned){
3       recv();
4       CUDA_alloc_data();
5       CUDA_copy_data(..,HostToDevice);
6       CUDA_start_kernel();
7       CUDA_copy_data(..,DeviceToHost);
8       send();
9  }
10 //End of worker node
```

Listing 3. The pseudo worker node code output by the code generation tool.

the model. This work is inspired by CUDASA[19], in which 3 abstraction layers as application layer, bus layer and network layer are added.

In contrast to CUDASA, we omit the application layer because the tool is dedicated to generating code for single application. Besides, in current implementation, all worker nodes in the same machine directly access the distributed memory at the head node, instead of sharing the system memory of the machine. Unlike CUDASA, all worker nodes are equal from the cluster point of view no matter whether or not they are housed by the same machine. Thus, we combine the bus layer and network layer of the CUDASA model, and introduce a new thread hierarchy named hyper-block:

hyper-block level: A hyper-block is the complete execution of kernels for the whole data set. Within one hyper-block, all grids executed in the worker nodes access to a copy from the system memory in the head node. Data sharing and synchronization are currently not possible on this level.

Figure 4 shows the thread and memory hierarchy of the original CUDA programming model with the extended hyper-block level. The hyper-block is a collective of the grids, and each grid is scheduled to one GPU device to execute a complete kernel for a part of the whole data set.

4.3 Code Parsing

This section presents a detailed description of the phase 1 in Figure 3. In this phase, the original CUDA source code is separated into two parts: one part suitable for executing on the head node and one part for the worker nodes.

To support the extended programming model, the total workload must be partitioned into a number of grids, and each grid is scheduled to a work node and executed by a kernel call. The work node code should contain a complete CUDA process, and cooper-

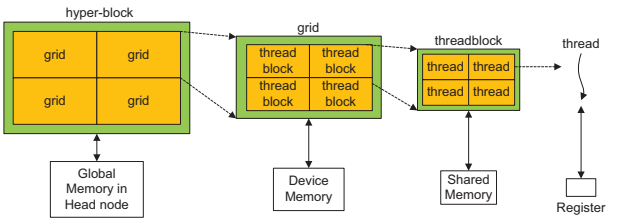
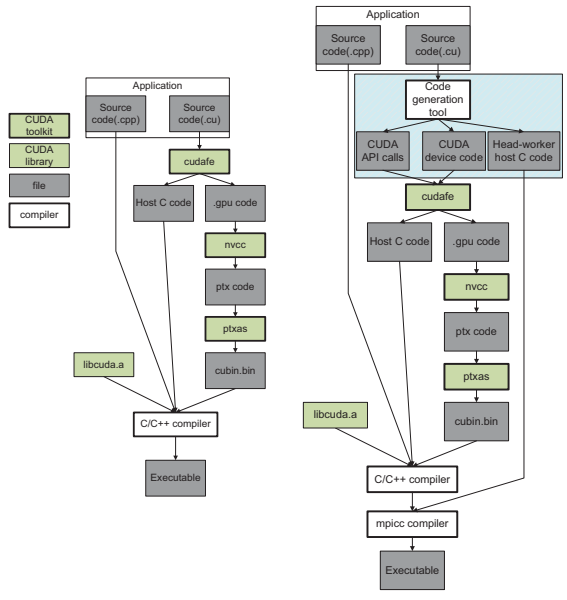


Figure 4. Overview of extended CUDA programming model.

ating with the management of the head node code. Both the head node code and the work node code are derived but from the original CUDA source code.

Standard CUDA source files consist of code executed by the host (CPU) and executed by the device (GPU). Host code contains regular C code (e.g. the input of raw data, processing of returned results from GPU) and CUDA API calls (e.g. memory space allocation on the GPU, memory copy from the system memory to the device memory and the kernel call). Device code is in the CUDA-C extension set, and only executable on a GPU device. It contains the actual computation which is typically highly parallel and multi-threading.

The CUDA compilation flow is shown in Figure 5(a), in which the input program is separated by the CUDA front end (*cudafe*) into host C code and device code in a preprocessing stage [20]. The device code is then compiled by *nvcc* as CUDA intermediate files and finally transformed into the CUDA binary. The CUDA binary along with the host C code separated by *cudafe*, the standard C source code of the application and the CUDA library are compiled and linked by the C/C++ compiler and output as an executable.



(a) Original compilation flow (b) Changed compilation flow

Figure 5. The original CUDA compilation flow and the changed flow after code generation tool is implemented.

It is more complicated to adapt the target device code for the GPU cluster than do this to the original CUDA code. So, a code parser is implemented on the original CUDA code instead of using the output of *cudafe*. The compilation flow changes after adding this code generation tool as shown in Figure 5(b). The result of code parsing is a three-part output as shown in the blue block in the figure.

- Regular C code for the host of the head node and the worker nodes
- C code with CUDA API calls
- CUDA C kernel code to be run on the GPU device

Instructions which will be executed on the GPUs of worker nodes and the CUDA APIs are separated from the original source code by identifying the CUDA syntax, leaving the remaining regular C code as the host code. The main entry of the new application is in the regular C code, and all the CUDA API calls are encapsulated in one function invoked in that C code. The regular C part of the the source code, the CUDA device code and the CUDA API calls are compiled using the standard *nvcc* compilation flow, and the output object file is linked with the compilation output of the host code in the MPI compiler (*mpicc*) to produce the executable for the GPU cluster.

4.4 Communication

As introduced in Section 4.3, the code parser outputs the two code sections for the head node and the worker node respectively. However, they do not interact with each other, because no communication has been performed. In this section, the communication is established as shown in phase 2 in Figure 3.

In a CPU-GPU single-node system, data flows between the CPU and the GPU by means of CUDA APIs such as *cudaMemCpy*. While in a GPU cluster, the head node (global host) is in the position of a CPU in a single-node system, GPUs in worker nodes function the same as they do in single-node system, and data flows between head node to GPUs via the CPUs in worker nodes (local host). Besides, in order to let the head node know the state of all the worker nodes to perform scheduling, a state query to all worker nodes should be done by the head node. In terms of this, communication is necessary between the global host and the local hosts to perform data transfer and state query.

Therefore, two types of parameters are transferred between the global host and the local hosts: application data sets and state parameters. Application data sets are involved in the actual computation, e.g. the input data of GPU transferred from the global host to the local host. The state parameters are communicated between nodes to update the device state in order to manage the scheduling process.

Mentioned as the background in Section 2.5, the communication with MPI can be performed as three different types: blocking communication, non-blocking communication, and RMA window access. By considering the characteristics of these three types of communication, different types of parameters are communicated using different methods:

- The input data is critical for the computation of the worker nodes, the communication time is preferably to be short and both the global host and the specified local host should be aware of the communication, so the input data is communicated in a blocking manner.
- The output result is returned after each grid is processed, but only after the last result package is received will the processing of the result start in the global host. So, it is less strict in timing for the result transfer, and to avoid blocking the thread, the non-blocking communication is performed.

- The state parameters on the global host can be directly written or read through the RMA window by the local hosts.

4.5 Scheduling Logic

To accomplish the scheduling process for all grids of the application, the scheduling logic should be designed as shown in Figure 3 phase 3. This scheduling logic enables the global host to manage all computing resources in the cluster by using the communication between the global host and the local host, by using the state parameters. The flow chart of the scheduling process is shown in figure 6, and it will be illustrated from the perspective both the global host view and the local host view. To run the CUDA-MPI hybrid application on the GPU cluster, N+1 threads should be started by MPI, where N denotes for the number of worker nodes employed by the application, and the one global thread manages the activities of the N local threads.

The arrows between the global host and local hosts in Figure 6 indicate the transferred data in between, including both the state parameters (end, available_head, grid_idx and available_worker) and data sets (input data and output result). The applied state parameters and their functionalities are introduced as follows:

end: indicating the end of the scheduling process when all grids are scheduled, from the global host to the local host, to inform all the local hosts to stop waiting for scheduled grids and to finalize the local host threads.

available_worker: an indication on the worker node of whether this node is available, from the local host to the global host. If the state turns from busy to available, it will update the resource pool in the head node to let the scheduler be able to employ this node again

available_head: an array indicating the available/busy state of all worker nodes in the resource pool, from the global host to the local host. The scheduler always distribute a grid to the first available worker node in this array. The state of a certain worker node becomes busy if the scheduler schedules a grid to this node, and it will update the parameter **available_worker** to be busy in the local host to start the processing. It will not become available until the worker node finishes processing and **available_worker** updates it from the local host.

grid_idx: specifying which grid to be executed by worker node, from global host to local host, to help the worker node to locate the data section to be processed in its copy of the whole data set.

On the global host side, a resource pool is established to indicate the available/busy state of all worker nodes. The scheduler first checks if there is a grid which has not been scheduled (Figure 6(a)), and if there is an available node in the resource pool (Figure 6(b)). The scheduler will always schedule the first grid in the waiting queue to the first available node in the resource pool. The array indicating the state of the worker nodes can be accessed in an RMA manner by the local hosts. Thus, the update of the node state resulted by the scheduling decision is also an indication to inform the specific worker node to be ready for the data from the head node and the following computation (Figure 6(c)). Before the actual computation in the worker node starts, the scheduler will also check whether it is the first time a grid is scheduled to the worker node (Figure 6(d)), and if so, the whole data set will be copied to the local host (Figure 6(e)). Afterwards, the global host will send the index of the grid to be processed (Figure 6(f)) to enable the local host to locate the grid in the copied data set and then copy it to the device memory of the GPU. The global host will also check in the RMA window of the local hosts to be aware of the accomplishment of grid processing in the worker nodes (Figure 6(g)). A non-blocking data receiving will be performed and the resource pool will be updated in the global host (Figure 6(h)) if a worker node finishes processing. Otherwise, the global host will directly skip to query about the remaining number of grids and available nodes, in order to start the next scheduling. When all grids of the application are scheduled, a flag 'end' indicating the

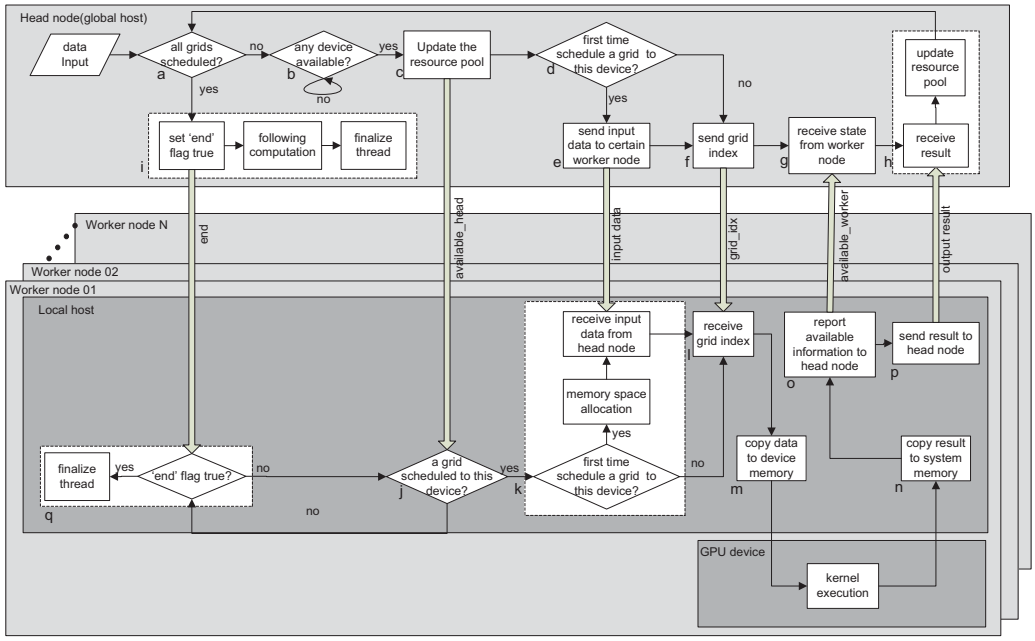


Figure 6. Design flow of tool implementation

end of the grid scheduling process is broadcasted to all local threads and the remaining computation is taken by the global thread alone (Figure 6(i)).

On the other side, the local host keeps waiting for a message from the global thread to start the processing (Figure 6(j)). If a grid is first time scheduled to the device, memory space used to receive data from global host must be allocated (Figure 6(k)). After receiving the index of the grid being scheduled (Figure 6(l)), a certain grid in the whole input data set can be located and moved to the GPU device memory (Figure 6(m)). The local host and GPU device of one worker node are a CPU-GPU single node system, so the communication is performed by CUDA APIs. After a GPU finishes the processing and outputs the result to the local host (Figure 6(n)), the local host will update the node state which is accessed by the the global (Figure 6(o)), send the result back to the global host (Figure 6(p)), and wait again for the next scheduled grid. Finally if the 'end' message is received, the local thread will jump out of waiting and get terminated (Figure 6(q)).

4.6 Performance and Evaluation

After performing the code parsing, the communication and the scheduling logic as introduced in the previous sections, the original single-GPU application can be automatically mapped to multiple GPUs in the cluster. In this section, we show the experimental results that the two benchmarks are ported to different number of nodes automatically. In Figure 7 and 8, the total execution time and the percentage of hardware utilization by employing scalable number of worker nodes of the two benchmarks are presented.

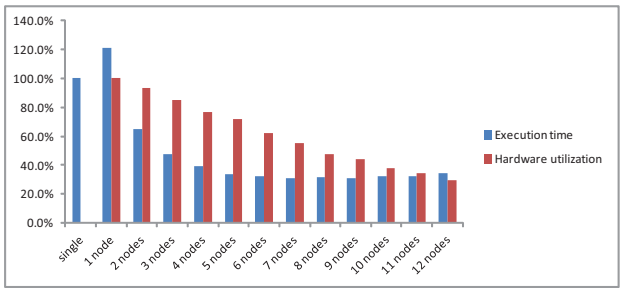


Figure 7. Performance of the application *Sum*

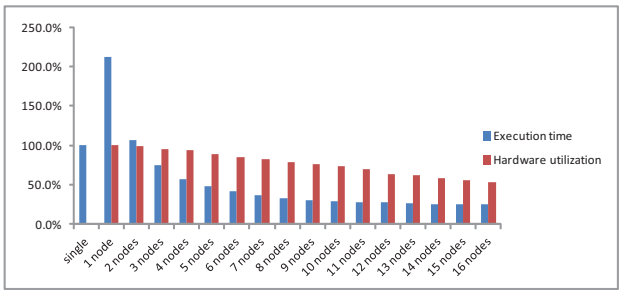


Figure 8. Performance of the application *DCT*

As shown in Figure 7 and 8, if only a few number of worker nodes are used in the cluster (e.g. 1 or 2), the performance of the cluster is worse than a single GPU, because of extra communication has to be performed in the network. Scaling with the number of worker nodes, the applications gain speedup with the multi-node cluster. For both benchmarks, we observe a 3x speedup as maximal. On the other hand, along with the increment of worker nodes, the utilization efficiency of the hardware becomes lower. As a result, it should be noted that it is not always better to employ more worker nodes. If the number of worker nodes is beyond the critical point (9 nodes for the *Sum* application), the application cannot gain any more speedup by employing more worker nodes, and the performance may even get worse.

It can be observed from the result that the more computational intensive an application is, the higher the critical point will be. For example, compared with *Sum*, the *DCT* application requires more computation per data element, thus the performance is improving when the number increases within the range of 16 nodes. The hardware efficiency of the application *DCT* stays relatively higher.

4.7 Limitation and Future Work

In this section, two limitations of the work in this paper are identified: extracting the name of the input/output array from the original code and perform inter-block communication between worker nodes.

In the implemented code generation tool, the name of the input/output array must be known for the tool to add necessary communication and invoke a function to call the CUDA APIs in the main entry. At present, the programmer should manually add a directive indicating the variable names, in order to let the code parser be aware of the arrays or parameters to be transferred and invoked. Also, the programmer should indicate the transfer direction of the arrays in the directive (i.e. from global host to local host, within local host or from local host to global host). This makes the code generation procedure not completely automatic, but we believe it is possible to develop a source-to-source compiler with some more engineering work to automatically extract variables, analyze their communication direction and separate the code based on the parser.

Secondly, there is no direct connection possible between worker nodes, and inter-block communication of different grids is not supported by the tool. Although applications with no inter-block communication are more suitable for the cluster (low communication overhead), we still hope to study on realizing inter-block communication within the cluster in the future work.

5. Adaptive Grid Scheduling

As a multi-device system, the performance of a GPU cluster is dominated by two main factors: the performance of hardware (processing power of CPU and GPU, bandwidth of interconnection) and the efficiency of the scheduling algorithm. In this section, first we will present the risk of introducing unnecessary communication overhead by the naive greedy scheduling. And a solution is given to adaptively schedule the workload of an application to an optimal number of worker nodes to improve the utilization of the worker nodes and the performance of the cluster. Based on the timing information provided by an actual runtime measurement before the start of the application, a tool is presented in Section 5.2 to realize the optimization. This tool can be either on top or not on top of the code generation tool introduced in Section 4. In Section 5.3, another feasible approach is presented to adaptively adjust the workload of each grid according to the time elapsed for data transfer, in order to hide more communication overhead behind the computation.

5.1 A Scheduling Example

As introduced in Section 4, the scheduling logic added by the code generation tool follows a FCFS (First Come First Serve) mechanism, the grids are scheduled sequentially to the worker node which

first becomes available. Naturally, this process is performed using a greedy strategy: the scheduler tries to distribute workload to as many as possible nodes bounded by the maximum amount of nodes whenever it sees an available node. As introduced in Section 4, when a grid is first time scheduled to a worker node, it consumes extra time compared with the following grids. The overhead of first time loading the CUDA driver is relatively larger than the following times. To employ more devices in the computation, more device level parallelism can be achieved, but meanwhile more setup time will be added. Thus, the greedy algorithm is not necessarily the most efficient solution, and a trade-off has to be made to determine the optimal number of the employed nodes. Figure 9 shows an example indicating that a greedy schedule is not an optimal approach. In this synthetic example, the total workload is 12 grids, the data transfer time is $1.6t$, the processing time of the first schedule is $1.2t$ and for the following schedules it is t.

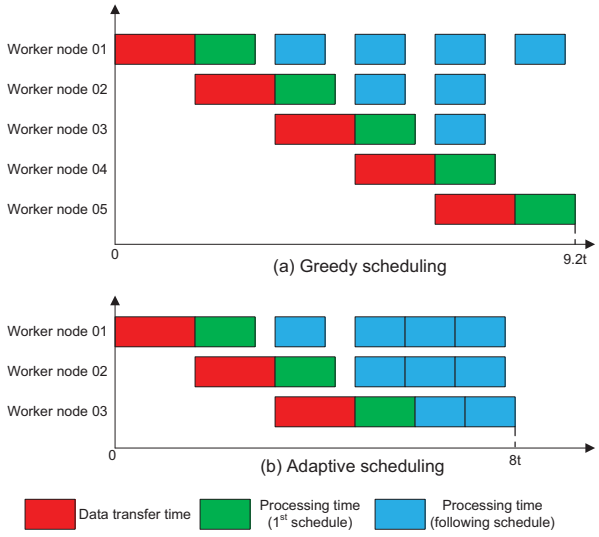


Figure 9. A comparison example between a greedy schedule and an adaptive schedule

In Figure 9, the processing time for the first scheduling (green blocks) is larger than the processing time for the following scheduling (blue blocks), because of the device start-up overhead the first time a device is started. When the data transfer from the head node to one worker node is performed, this blocking communication will stop the global host to proceed with the following operations until the communication is completed. So, the scheduler can schedule a new grid to a worker node only when there is no data transfer (red blocks) in the network. With the current MPI communication APIs and the network infrastructures, the broadcast process is not able to send the data set to all the worker nodes simultaneously. The data transfer can only be performed sequentially by a series of blocking sends. Consequently, idle slots of a worker node could appear if the worker node finishes processing but a data transfer is undertaken in the network.

As shown in Figure 9(a), the greedy scheduler tries to start as many as possible worker nodes until the limit of the number of grids is met. By the greedy scheduling, the more worker nodes are employed lead to more extra data transfer will be performed, which results in a non-optimal performance of the cluster. In contrast to that, by adaptive scheduling, only an optimal number of worker nodes are employed for processing, guaranteeing the least total processing time of the application. In Figure 9(b), the adaptive scheduling is performed to the same synthetic example as used for the greedy scheduling. The scheduler searches for the optimal number of worker nodes, which leads to the total execution time as a minimum. As a result, that the three worker nodes proves to be an optimal solution instead of five nodes started by the greedy scheduler, and the performance is improved by 13%.

5.2 Adaptive Scheduler Implementation

The target of the adaptive scheduler is to distribute the workload over an optimal number of worker nodes bounded by the maximal number of nodes in the cluster, in order to achieve the smallest execution time. To function this, the adaptive scheduler uses a cluster performance simulator to calculate the total execution time in conditions employing different number of worker nodes as shown in Listing 4. The optimal solution is searched among the simulated results, and the adaptive scheduler will start grid scheduling and only distribute workload over an optimal number of worker nodes.

```
1  //Start
2  read_parameter(#max_node);
3  for(i=1; i<#max_node; i++)
4       start_simulator(i);
5  compare_result();
6  start_scheduling();
7  //End
```

Listing 4. The pseudo code of the adaptive scheduler.

Listing 5 shows how the performance simulator works in a pseudo code. This simulator is input with the timing information derived from the runtime measurement of a test run by scheduling only two grids to one worker node, and the required timing information are data transfer time (t_trans), processing time for the first and following scheduling (t_1st and t_2nd). It tries out all possible number of nodes equipped in the cluster, and considers three scheduling cases which occur in actual runtime:

- A device is started at the first time (Line 7 to Line 17). Data transfer needs to be performed and the processing takes longer time (t_1st). The next communication is only available at t_trans after this scheduling.
- A device is started not at the first time, but all devices are started (Line 18 to Line 26). No data transfer is needed, and the processing takes shorter time (t_2nd). A new grid can only be scheduled after finishing data transfer to a new started node.
- All devices have been started (Line 18 to Line 34). Scheduling a new grid is always possible, and a new grid will be scheduled when a node first finish processing among all nodes.

With these information, the simulator is able to simulate the scheduling process in the cases with arbitrary number of worker nodes, and obtain the total execution time for processing all the grids.

Currently, we only obtain the timing information by a test run of the application. But in the future, using a complete analytical performance model of the GPU cluster, all the timing information can be calculated mathematically. Also, instead of simulating the process of scheduling, the performance of the cluster equipped with a different number of nodes can be predicted in a mathematical model, and this work will be introduced in Section 6.

```
start_simulator(){
    read_parameter(#node,#grid);
    read_parameter(t_trans,t_1st,t_2nd);
//try out all possible number of nodes
    for (n=0; n<#node; n++){
        while (#remaining_grid>0){
            for (m=0; m<=n; m++){
                if (not all devices are started
                    && device[m].first==1){
                    n_grid--;
                    device[m].first=0;
//the next communication can be performed
//after t_trans
                    t_comm=t_comm+t_trans;
//node available after t_trans+t_1st
                    device[m].time=t_comm+t_1st;
                    break;
                    }
//in the case that the node is not the
//first time scheduled to
                else if (t_comm>=device[m].time){
                    n_grid--;
//node available after t_2nd
                    device[m].time=t_comm+t_2nd;
                    break;
                    }
            }
            if (all devices are started){
//in the case all devices are started, the
//next communication is performed when a
//node first finish its processing
                for (m=0; m<=n; m++)
                    t_comm=min(device[m].time);
            }
        }
//total execution time obtained when the
//last node finished its processing
        for(m=0; m<=n; m++)
        t_total[n]=max(device[m].time);
    }
}
```

Listing 5. The pseudo code of the performance simulator.

5.3 Simulation Tool Test

The simulation tool is used to try out all scheduling possibilities and search for an optimal solution. The accuracy is quite important for this tool which consists of two factors: the numerical accuracy (to precisely predict the cluster performance) and the trend accuracy (to be able to find out the optimal point correctly).

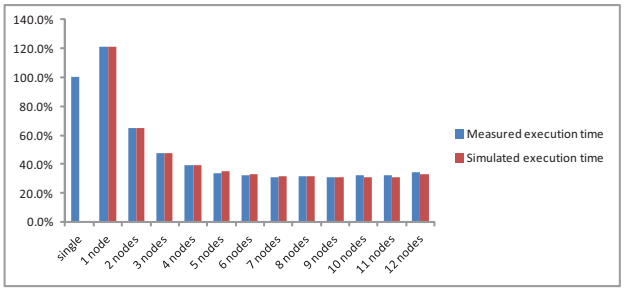


Figure 10. A comparison between measured performance and simulated performance of the application *Sum*

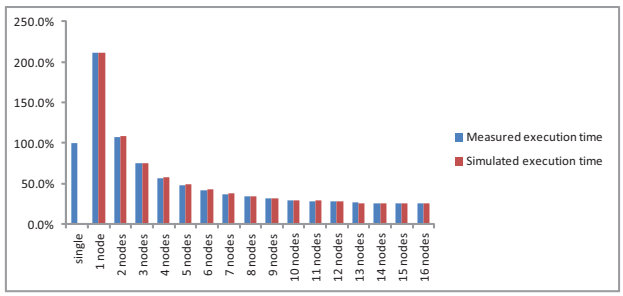


Figure 11. A comparison between measure performance and simulated performance of the application *DCT*

Figure 10 and 11 show the comparison between the measured performance and the simulated performance for the two benchmarks. As shown in the figures, the simulated result quite closely predicts the measured performance, with an error smaller than 5%. Also, the simulator predicts the trend correctly. For the application *Sum* the simulator finds the optimal number of nodes is 9, and for the application *DCT* it finds 16. Both results are both consistent with the real cases. So, we are convinced the this simulation tool is reliable to predict the performance of the cluster and search for an optimal number of worker nodes.

5.4 Optimal Workload Partitioning

From another point of view, to improve the efficiency of the cluster, an alternative is to adaptively partition the workload into an optimal number of grids, instead of letting the programmer set a fixed number. Another synthetic example is given in Figure 12 to illustrate the performance improvement by workload partitioning. In this example, the original total workload is still 12 grids and the processing time is t per grid. However, if we resize the grids to become $\frac{1}{3}$ larger, the number of grids is reduced to 9, and the processing time of the first schedule is $1.6t$ (equals to data transfer time). The processing time for a following grid is now $1\frac{1}{3}t$. As shown in Figure 12(b), more computation time of each grid can be hidden behind the communication to perform data transfer, and less idle slots occurs compared with the condition processing a fixed number of grids (Figure 12(a)). An ideal case is that the processing time of each grid is exactly equal to the communication cost, in order to reduce the idle slots of the worker nodes to a minimum.

This optimal workload partitioning can be implemented by using an analytical performance model of the GPU cluster. The data transfer time can be calculated according to the network specification, and with a GPU performance model, we can partition the whole data set into suitable size of grids, whose processing time can perfectly be hidden behind the communication overheads. The implementation of this optimization is left for the future work.

6. Analytical Performance Model

There is a considerable amount of work on the analytical GPU performance model [9][13][17][18]. However, these performance models only concern about the performance on kernel execution, but don't consider the host levels (e.g. data transfer). To extend the performance model to a GPU cluster, we first need to model all the individual factors on the host level and the network level. Then by combining the scheduling logic with these factors, the performance of the cluster can be modeled.

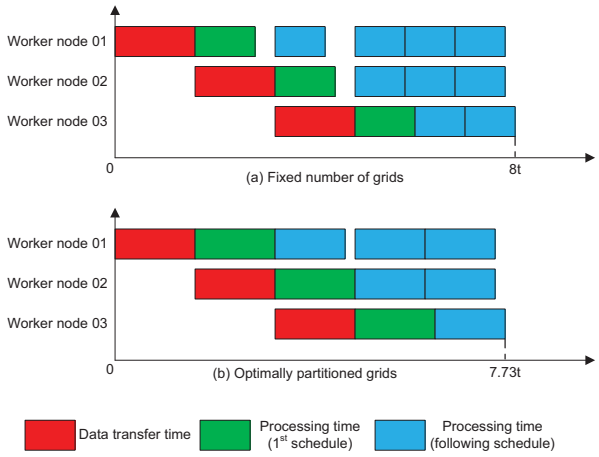


Figure 12. A comparison example between a fixed number of grids and an adaptive number of grids

6.1 Existing GPU Performance Model

There are various GPU performance models existing, and in this section, 2 models will be introduced. Hong and Kim [13] propose an analytical model for GPU architecture, where they introduce two terminologies: MWP (Memory Warp Parallelism) and CWP (Computation Warp Parallelism). The degree of MWP is tightly related to the DRAM system of the GPU and the memory access pattern (coalesced or non-coalesced memory access), and the degree of CWP depends on the number of active warps on an SM and number of warps that SM can execute during one memory waiting period. The total execution time of a launched CUDA kernel can be analytically calculated according to the MWP, the CWP, a profile of the application and the specification of the GPU.

By A.Bakhoda et al., a detailed cycle-level GPU simulator is developed to analyze CUDA workloads [9]. In this simulator, a timing model is constructed for GPU micro-architecture (SM, caches, interconnection, graphics DRAM, etc.) [8]. The *ptx* assembly is input to the simulator and the whole life cycle of each instruction is simulated according to the timing model. Consequently the total execution time of the kernel can be predicted and is proved to be 89% match with the actual case.

6.2 GPU Performance Model Extension

As presented in Section 6.1, kernel execution time can be precisely predicted by existing analytical performance models. From the point of system efficiency, applications that run on a GPU cluster are preferably to take millions of cycles to compensate for the communication bottlenecks with device-level parallelism. So, in the model extension, only the major factors such as data transfer and memory allocation are concerned, and the cycle-level operations in the scheduling logic on global and local hosts are omitted. To extend the existing models to fit a GPU cluster, the following performances still have to be estimated to obtain the total execution time of an application:

- Memory allocation time in graphics DRAM.
- Data transfer rate between graphics DRAM and system memory by PCIe.
- Memory allocation time in system memory.
- Data transfer rate between head node and worker node by Ethernet connection.
- CUDA start-up overhead when loading CUDA driver.

Among these five mentioned parameters, memory allocation time is negligible. And the data transfer rate can be obtained from the specification of the cluster. In our case, the theoretical peak bandwidth for PCIe bus is 6GB/s and for Gigabit Ethernet is 125MB/s.

In order to analytically model the performance of the GPU cluster, the parameters needed are defined in Table 2.

parameter name	Description
t_{Kernel}	kernel execution time
N_{grid}	size of data set transferred on PCIe bus when each grid is processed
N_{total}	size of the whole data set
η_{PCIe}	utilization rate of PCIe bus
B_{PCIe}	bandwidth of PCIe bus
η_{Eth}	utilization rate of Ethernet
B_{Eth}	bandwidth of Ethernet
$t_{CUDAoverhead}$	start-up overhead of CUDA API call

Table 2. Descriptions of performance parameters

By using the performance parameters, the data transfer time (t_{Eth}) from global host to local host, and between local host and GPU devices (t_{PCIe}) can be calculated as:

$$t_{Eth} = \frac{N_{total}}{B_{Eth} * \eta_{Eth}} \tag{1}$$

$$t_{PCIe} = \frac{N_{grid}}{B_{PCIe} * \eta_{PCIe}} \tag{2}$$

The processing time of each grid can be obtained with respect to the first scheduling (t_{1st}) or following scheduling (t_{2nd}), by considering the data transfer time (t_{PCIe}), the CUDA overhead ($t_{CUDAoverhead}$) and the kernel execution time (t_{Kernel}):

$$t_{1st} = t_{PCIe} + t_{CUDAoverhead} + t_{Kernel} \tag{3}$$

$$t_{2nd} = t_{PCIe} + t_{Kernel} \tag{4}$$

6.3 Model for Greedy Scheduling

In the greedy scheduling approach, the scheduler will automatically start as many as possible worker nodes until all the grids are processed. In this case (assuming the cluster is equipped with infinite number of worker nodes), the node which is last started must only process one grid, and the number of processed grids of other nodes depends on the relation between the data transfer time and the grid processing time. In this section, a mathematical method is given to describe the maximal employable number of nodes in the greedy scheduling and the total execution time by employing this number of worker nodes.

To simplify the expressions, two coefficients K_1 and K_2 are introduced to indicate the ratio between the processing time of each grid and the data transfer time from global host to local host as following:

$$K_1 = ceiling(\frac{t_{1st}}{t_{Eth}}), K_2 = ceiling(\frac{t_{2nd}}{t_{Eth}}) \tag{5}$$

It should be noted that these two ratios are integer numbers, and a ceiling divide is performed in the calculation.

Figure 13 shows examples as an explanation of the mathematical formulating. In Figure 13(a) and (c) $K_1 = K_2 = 1$, while in figure 13 (b) and (d) $K_1 = K_2 = 2$. Because of the rounding, the blue block (t_{2nd}) does not necessarily equals to the green block (t_{1st}). Besides, in (a) and (b), the scheduling achieves the best case, in which all nodes are used when the last node is started. And in (c) and (d), the scheduling works in the worst case, where only the last node processes data and the other nodes have already finished processing.

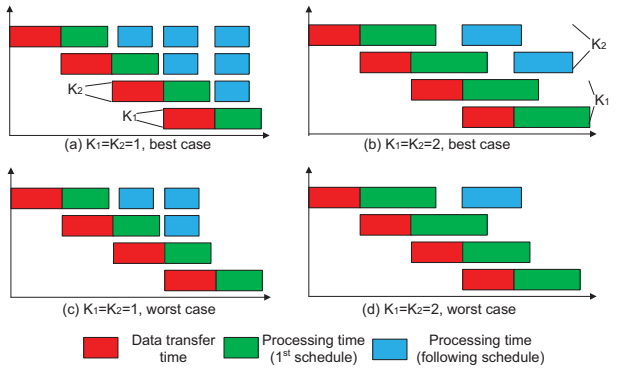


Figure 13. Simple examples in different conditions explaining the modeling

As shown in Figure 13 (a) and (b), when the grids are scheduled in a greedy manner in the best case, the last K_1 devices only process 1 grid. The coefficient K_2 indicates the number of GPU devices with the same number of processed grids. So given a certain number of nodes, we can calculate the maximal and minimal number of grids that can be processed by this number of nodes with a greedy scheduler, as respect to the best case and the worst case. On the other hand, a given number of grids can always be located somewhere in between the best case and the worst case of a certain number of nodes, and thus we can calculate the maximal required number of devices of an application.

Equation (7) and (8) show the relation between the number of grids G and the maximal number of required devices D_{max} in the best case and worst case respectively. The P and Q in the equations indicate the quotient and remainder of the equation $\frac{D_{max}-K_1}{K_2}$:

$$D_{max} = K_1 + PK_2 + Q, \quad (where\ Q < K_2) \tag{6}$$

In the best case:

$$\begin{aligned} &G_{best}|_{Dmax} \\ &= K_1 + 2K_2 + 3K_2 + ... + (P+1)K_2 + (P+2)Q \\ &= K_1 + K_2 \textstyle\sum_{n=2}^{n=P+1} n + \sum_{n=1}^{n=Q}(P+2) \end{aligned} \tag{7}$$

And in the worst case:

$$\begin{aligned} &G_{worst}|_{Dmax} = G_{best}|_{Dmax-1} + 1 \\ &= K_1 + 2K_2 + 3K_2 + ... + (P+1)K_2 + (P+2)(Q-1) + 1 \\ &= K_1 + K_2 \textstyle\sum_{n=2}^{n=P+1} n + \sum_{n=1}^{n=Q-1}(P+2) + 1 \end{aligned} \tag{8}$$

Consequently, if the given number of grids G satisfies that $G_{worst}|_{Dmax} \leq G \leq G_{best}|_{Dmax}$, the maximal required number of nodes for G grids is D_{max}. For example, in the case where $K_1 = K_2 = 1$ and $G = 9$, the best case and the worst case around G are $G_{best} = 1+2+3+4 = 10$ and $G_{worst} = 1+2+2+1 = 7$. P and Q can be obtained as $P = 3$ and $Q = 0$, so the D_{max} for this case $G = 9$ is 4. This example covers a case in between Figure 13 (a) and (c).

Considering the greedy scheduling, the last started node will always only processes one grid. Naturally the processing time t_{1st} is larger than t_{2nd}, and the whole execution process always finalizes when the last started node accomplishes its processing. Thus, we can formulate the performance of the cluster as the following equation:

$$t_{total} = t_{Eth} D_{max} + t_{1st} \tag{9}$$

6.4 Model for Scheduling to Arbitrary Nodes

Sometimes an actual cluster may not contain so many nodes as the greedy scheduling requires, and also the greedy scheduling may lead to worse performance than using less number of nodes, which makes the model for scheduling to arbitrary number of nodes quite critical. With this model, we can predict the performance of a cluster equipped with a limited amount of nodes, and can also find out an optimal number of nodes for an application in a mathematical way.

This model is based on the model for the greedy scheduling introduced in Section 6.3. Instead of assuming the cluster to have an infinite number of nodes, in this model, we consider only an insufficient number of nodes which is smaller than D_{max} in the cluster. Assuming the number of nodes in the cluster is D_1 ($D_1 < D_{max}$), and a greedy scheduling is first performed to distribute part of the total workloads over all the D_1 nodes. After that, the remaining grids can be scheduled freely to a node whenever the node become available, without considering the constraints resulted by the data transfer as in the greedy scheduling phase. The number of greedily scheduled grids (G_{greedy}) and the remaining grids (G_{remain}) can be described as Equation (11) and (12), where G denotes the total number of grids set by the programmer.

$$D_1 = K_1 + P_1K_2 + Q_1, \quad (where\ Q_1 < K_2) \tag{10}$$

$$\begin{aligned} &G_{greedy} = G_{best}|_{D1} \\ &= K_1 + 2K_2 + 3K_2 + ... + (P+1)K_2 + (P+2)Q \\ &= K_1 + K_2 \textstyle\sum_{n=2}^{n=P+1} n + \sum_{n=1}^{n=Q}(P+2) \end{aligned} \tag{11}$$

$$G_{remain} = G - G_{greedy} \tag{12}$$

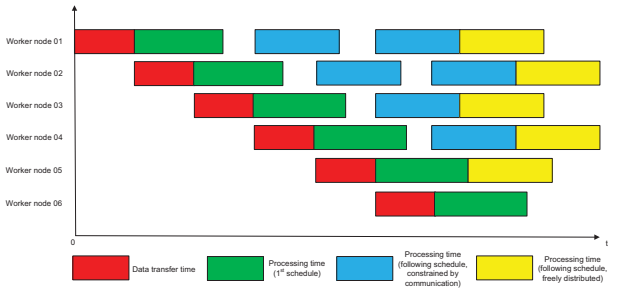


Figure 14. An example to explain the scheduling process

In Figure 14, an example is given, where $K_1 = K_2 = 2$. In the example, the red, green and blue blocks are confined by the communication undertaken in the network, but the yellow blocks are free to be scheduled. For simplicity and it is valid in most of cases, we assume K_1 always equals to K_2 because the CUDA start-up overhead is negligible compared to the data transfer time. Then, easily all the D_1 nodes can be sorted into K_2 groups, among which the nodes share their timing similarities. Accordingly in the example, the nodes 1, 3, 5 and nodes 2, 4, 6 are separated into two groups. The groups are named from S_{D1} to $S_{D1-K2+1}$, because

these $K_2 - 1$ representative nodes must be separated into $K_2 - 1$ different groups. The term $\#S_x$ denotes the size of the group containing the *node x*. The number of nodes of each group can be obtained by:

$$\begin{aligned}
&\#S_{D1} = ceiling(\tfrac{D_1}{K_2}) \\
&\#S_{D1-1} = ceiling(\tfrac{D_1 - \#S_{D1}}{K_2 - 1}) \\
&.... \\
&\#S_{D1-K2+1} = ceiling(\tfrac{D_1 - \#S_{D1} - \#S_{D_2} - ... - \#S_{D1-K2+2}}{1})
\end{aligned} \tag{13}$$

After greedy scheduling, the remaining grids are scheduled in the following pattern: the grids are scheduled to each device once sequentially from the group $S_{D1-K2+1}$ to S_{D1}, and repeat this process again until all grids are scheduled. By dividing the remaining number of grids with the number of nodes as Equation (14), we obtain 2 numbers: the quotient X and remainder Y. While scheduling the remaining grids (G_{remain}), each node can receive X grids in this process. And the last Y grids are scheduled depending on the number and the size of the groups.

$$G_{remain} = XD_1 + Y, (Y < D_1) \tag{14}$$

To determine the node which the last grid is scheduled to, the following cases are considered:

- $Y = 0$, the last grid is scheduled to the node D_1
- $1 \leq Y < \#S_{D1-K2+1}$, the last grid is scheduled to one of nodes in the the group $S_{D1-K2+1}$, except the node $(D_1 - K_2 + 1)$
- $Y = \#S_{D1-K2+1}$, the last grid is scheduled to the node $(D_1 - K_2 + 1)$
- $\sum_{n=0}^{m-1}(\#S_{D1-K2+n}) < Y < \sum_{n=0}^{m}(\#S_{D1-K2+n})$, the last grid is scheduled to one of the nodes in the group $S_{D1-K2+m}$ $(where\ m \leq K_2)$, except the node $S_{D1-K2+m}$
- $Y = \sum_{n=0}^{m}(\#S_{D1-K2+n})$, the last grid is scheduled to the node $S_{D1-K2+m}$ $(where\ m \leq K_2)$

Correspondingly to these cases, the total execution time of the application with an arbitrary number of nodes can be expressed as:

$$t_{total} = \begin{cases} t_{Eth}D_{max} + t_{1st},\ if\ D_1 \geq D_{max} \\ t_{Eth}(D_1 - K_2 + m) + t_{1st} + (X+1)t_{2nd}, \\ \qquad if\ Y = \sum_{n=0}^{m}(\#S_{D1-K2+n}) \\ t_{Eth}(D_1 - K_2 + m) + (X+2)t_{2nd}, \\ \qquad if\ Y > \sum_{n=0}^{m-1}(\#S_{D1-K2+n}) \\ \qquad and\ Y < \sum_{n=0}^{m}(\#S_{D1-K2+n}) \end{cases}, \quad (where\ m \leq K_2) \tag{15}$$

In the example shown in Figure 14, the total number of grids is 17, $K_1 = K_2 = 2$, and the cluster employs 6 worker nodes. According to Equation (11) and (12), $G_{greedy} = 12$ and thus $G_{remain} = 5$. Divide G_{remain} by the number of nodes and we can obtain $X = 0$ and $Y = 5$. The 6 worker nodes are separated into 2 groups because $K_2 = 2$, and with Equation (13) the size of the groups is $\#S_6 = \#S_5 = 3$. Because $\#S_5 < Y < (\#S_5 + \#S_6)$, the last grid is scheduled to one node of the group S_6 except *node 6*. The total execution time can be calculated as $t_{total} = 6t_{Eth} + 2t_{2nd}$ according to Equation (15), which is consistent with the result shown in the figure.

7. Conclusion

In this paper, we propose a code generation tool to map a single-GPU application to a GPU cluster automatically. This tool performs source-to-source code translation, MPI communication between nodes and scheduling logic. Although it has some limitation, it is the first tool to automatically generate code for a GPU cluster. And with this tool a 3x speedup is achieved for both benchmarks benefiting from the device-level parallelism of the cluster.

To predict the performance of the GPU cluster and to adaptively search for an optimal number of worker nodes, a simulation tool is implemented and proved to be 95% accurate with the two benchmarks. By using this simulation tool, we show how a greedy scheduler can be improved by means of an adaptive scheduling algorithm and optimal workload partitioning.

Also, on top of the existing GPU analytical performance models, an extension is performed to the GPU cluster. By modeling the kernel execution of the worker nodes, data transfer in the interconnect network, and the scheduling logic in the head node, the performance model of the GPU cluster can be established mathematically. Based on this GPU cluster performance model, the adaptive scheduler can be implemented in a completely mathematical method. The performance of a GPU cluster becomes predictable by using this model, which provides useful information for code tuning and cluster purchasing.

References

[1] Condor Manual. http://www.cs.wisc.edu/condor/manual/v7.7/.

[2] FASTRA GPU SuperPC. http://fastra.ua.ac.be/en/index.html.

[3] Geforce GTX 570 Architecture. http://www.geforce.com/Hardware/GPUs/geforce-gtx-570/architecture.

[4] GPU Cluster Computing. http://parse.ele.tue.nl/st/cluster.

[5] *NVIDIA CUDA C Programming Guide*, 4.0 edition.

[6] NVIDIA GeForce 8800 GPU Architecture Overview. Technical report, NVidia Corporation, 11 2006.

[7] NVIDIA GeForce GTX 200 GPU Architecture Overview. Technical report, Nvidia Corporation, 5 2008.

[8] T. M. Aamodt, A. Bakhoda, and W. W. L. Fung. *GPGPU-Sim: A Performance Simulator for Massively Multithreaded Processor Research*.

[9] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing CUDA Workloads Using a Detailed GPU Simulator. 2006.

[10] B. Barney. Message Passing Interface (MPI). https://computing.llnl.gov/tutorials/mpi/.

[11] Z. Du. *Parallel Programming for High Performance Computation–MPI Parallel Program Design*.

[12] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover. GPU Cluster for High Performance Computing. 2004.

[13] S. Hong and H. Kim. An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness. 2009.

[14] V. V. Kindratenko, J. J. Enos, G. Shi, M. T. Showerman, G. W. Arnold, J. E. Stone, J. C. Phillips, and W.-m. Hwu. GPU Clusters for High-Performance Computing. 2009.

[15] C. Nugteren. *20 TFLOP GPU Cluster–Proposal for a 4-system, 16-GPU Massively Parallel Compute Cluster*. Technology University of Eindhoven, 4 2011.

[16] PBS Works. *Scheduling Jobs onto NVIDIA Tesla GPU Computing Processors using PBS Professional*, 10 2010.

[17] A. Resios. GPU Performance Prediction Using Parametrized Models. Master's thesis, Utrecht University, 2011.

[18] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu. An Adaptive Performance Modeling Tool for GPU Architectures. 1 2010.

[19] M. Strengert, C. Mller, C. Dachsbacher, and T. Ertl. CUDASA: Compute Unified Device and Systems Architecture. 2008.

[20] J. VanderSpek. *The CUDA Compiler Driver NVCC*. NVIDIA Corporation, 1 2008.