Automatic CUDA Code Synthesis Framework for Multicore CPU and GPU Architectures

Hanwoong Jung[1], Youngmin Yi[2], and Soonhoi Ha[1]

[1] School of EECS, Seoul National University,
Seoul, Korea
{jhw7884,sha}@iris.snu.ac.kr
[2] School of ECE, University of Seoul,
Seoul, Korea
ymyi@uos.ac.kr

Abstract. Recently, general purpose GPU (GPGPU) programming has spread rapidly after CUDA was first introduced to write parallel programs in high-level languages for NVIDIA GPUs. While a GPU exploits data parallelism very effectively, task-level parallelism is exploited as a multi-threaded program on a multicore CPU. For such a heterogeneous platform that consists of a multicore CPU and GPU, we propose an automatic code synthesis framework that takes a process network model specification as input and generates a multithreaded CUDA code. With the model based specification, one can explicitly specify both function-level and loop-level parallelism in an application and explore the wide design space in mapping of function blocks and selecting the communication methods between CPU and GPU. The proposed technique is complementary to other high-level methods of CUDA programming.

Keywords: GPGPU, CUDA, model-based design, automatic code synthesis.

1 Introduction

Relentless demand for high computing power is leading us to a many core era where tens or hundreds of processors are integrated in a single chip. Graphics Processor Units (GPUs) are the most prevailing many-core architecture today. Compute Unified Device Architecture (CUDA) is a programming framework recently introduced by NVIDIA, which enables general purpose computing on Graphics Processing Units (GPGPU). With massive parallelism of GPGPU, CUDA has been very successful for acceleration of a wide range of applications in various domains. CUDA is essentially a C/C++ programming language with several extensions for GPU thread execution and synchronization as well as GPU-specific memory access and control. Its popularity has grown rapidly, as it allows programmers to write parallel program in high-level languages and achieve huge performance gain by utilizing the massively parallel processors in a GPU effectively.

Although GPU computing can increase the performance of the task by exploiting data parallelism, it is common that not all tasks can be executed in GPUs as they expose insufficient data parallelism or they require more memory space than the GPU can provide, and so on. While a GPU exploits data parallelism very effectively, task-level parallelism is more easily exploited as a multi-threaded program on a multi-core CPU. Thus, to maximize the overall application performance, one should exploit the underlying heterogeneous parallel platforms that consist of both multi-core CPU and GPU.

It is very challenging to write parallel programs on heterogeneous parallel platforms. It is a well-known fact that high performance is not easily achieved by parallel programming. It is reported that about 100-fold performance improvement is achieved by optimizing the parallel program in a heterogeneous system that consists of a host and a GPU [1]. But optimizing a parallel program is very laborious and time-consuming. To help CUDA programming, several programming models and tools have been proposed, which will be reviewed in the next section. In this paper, we propose a novel model-based parallel programming methodology.

We specify a program with a task graph where a node represents a code segment and an arc represents the dependency and interaction between two adjacent nodes. The task graph has the same execution semantics as dataflow graphs: a node becomes executable when it receives data from the predecessor nodes [2] [3] and an arc represents a FIFO queue that stores the data samples transferred from the source node to the destination node. Such dataflow models express the task-level parallelism of an application naturally. A node can be mapped to a CPU processor core or to a GPU. We identify a data-parallel node that may be mapped to the GPU, to explicitly express the data parallelism. A data-parallel node is associated with a *kernel* to be executed in the GPU.

From this specification, one can explore a wide design space that is constructed according to how design choices are combined: which node should be mapped to the CPU and implemented in C, or mapped to the GPU and implemented in CUDA kernel? And we can select the types of communication APIs between CPU and GPU and kernel invocation methods. Therefore, in this paper, we propose an automatic code synthesis framework that takes as input a dataflow graph (similar to KPN graph) and generates CUDA code for heterogeneous GPGPUs and multi-core CPU platforms.

2 Related Work

Since CUDA places on the programmer the burden of packaging GPU code in separate functions, of explicitly managing data transfer, and of manually optimizing the utilization of the GPU memory, there is keen interest in developing a high-level method of CUDA programming without worrying about the complexity of the underlying GPU architecture.

*hi*CUDA [4] is an example that has been proposed as a directive-based programming language for CUDA programming. It uses compiler directives to specify parallel execution region of the application similarly to OpenMP. A *hi*CUDA compiler translates it to an equivalent CUDA program without any modification to the original sequential C code.

In our framework, it is assumed that the kernel function is given. Therefore, if we assume the kernel function code can be obtained using *hi*CUDA, our framework is complementary to the works.

GPUSs [5] is a directive-based programming model for exploiting task parallelism as well as data parallelism. So it focuses on heterogeneous systems with general purpose processors and multiple GPUs. Also it supports other targets such as Cell BE, SMP with almost the same user experience. Although its runtime system keeps track of the input and output of each kernel in order to reduce the communication, they do not support asynchronous communication between the CPU and the GPU for overlapping the data transfer and the kernel execution yet to our best knowledge.

StreamIt [6] is a programming model especially targeting for streaming applications. It is based on a dataflow model with a block-level abstraction: a basic unit for a block is called a *filter*. The *filter*s can be connected in a commonly occurred predefined fashion into composite blocks such as pipelines, split-joins, and feedback loops. Like our framework, *StreamIt* supports other heterogeneous platforms such as Cell BE. It is not known to us, however, how to explore the design space of CUDA programming.

Jacket [7] is a runtime platform for executing *MATLAB* code on GPGPUs. It supports simple casting functions to create GPU data structure allowing programmers to describe the application with the native *MATLAB* language. Through simply casting the input data, native *MATLAB* functions wrap those input into GPU functions.

3 Motivation: CUDA Programming

Typical GPUs consist of hundreds of processing cores that are organized in a hierarchical fashion. A computational unit of GPU that is executed by a number of CUDA threads is called the kernel. A kernel is defined as a function with the special annotation, and the kernel is launched in the GPU when the kernel function is called in the host code. The number of threads to be launched can be configured when the kernel is called.

CUDA assumes heterogeneous architectures that consist of different type of processors (i.e., CPUs and GPUs) and separate memory spaces. GPU has its own global memory separated from the host memory; therefore, to execute a kernel in GPU, input data needs to be copied to the GPU memory from the host memory. Likewise, the result needs to be copied from the GPU memory to the host memory after the kernel is completed.

A CUDA program usually uses a fork-join model of parallel execution: when it meets a parallelizable section of the code, it defines the section as a kernel function that will be executed in the GPU by launching a massive number of threads (fork). And the host CPU waits until it receives the result back from the GPU (join) and continues. The maximal performance gain can be formulated as follows:

$$G(x) = \frac{S+P}{S+C+\dfrac{P}{N}} = \frac{1}{x+\dfrac{1-x}{N}+\dfrac{C}{S+P}} \tag{1}$$

where S and P mean the execution time of the sequential section and the parallelizable section to be accelerated by GPU and C means the communication and synchronization overhead between CPU and GPU. N is the effective number of processors in the GPU, which is usually very large, and x presents the ratio of the sequential section in the application. It is observed that C is not negligible and often becomes the limiting factor of the performance gain. In equation (1), if P increases and C remains small, synchronous communication is commonly used between the host CPU and the GPU.

In this paper, however, we also consider the case where P is not dominant so that it is important to use task-parallelism that is included in S, and reduce the communication overhead C in equation (1). So we utilize the multi-core CPU maximally by making a multi-threaded program, in which a thread is mapped to the GPU if the thread has a massive data parallelism inside. And to overlap kernel execution and data communication between the CPU and the GPU, asynchronous CUDA APIs are also considered.

Therefore the design space of CUDA programming is defined by the following design axes: mapping of tasks to the processing elements, number of GPUs, communication protocols, and the communication optimization methods that will be discussed later in section 6. In this paper, we explore the design space of CUDA programming for a given task graph specification graph.

4 The Proposed CUDA Code Synthesis Framework

Fig. 1 illustrates the overall flow of the proposed CUDA code synthesis framework. An application is specified in a task graph based on the CIC model that has been proposed as a parallel programming model [8]. A CIC task is a concurrent process that communicates with other tasks through FIFO channels. We assume that the body of the task is defined in a separate file suffixed by .cic. If a task has data-parallelism inside, we assume that two definitions of the task are given, one for a CPU thread implementation and the other for a GPU kernel implementation. Architecture information is separately specified that defines the number of cores in the CPU and the number of GPUs available.

In the mapping stage, the programmer decides which task node to execute on which component in the target architecture. While multiple tasks can be mapped onto a single GPU, a single task cannot be mapped onto multiple GPUs. After mapping is determined, programmers should decide further design configurations such as communication methods between CPU and GPU, and the number of CUDA threads, and the grid configuration for each GPU task (i.e., CUDA kernel).

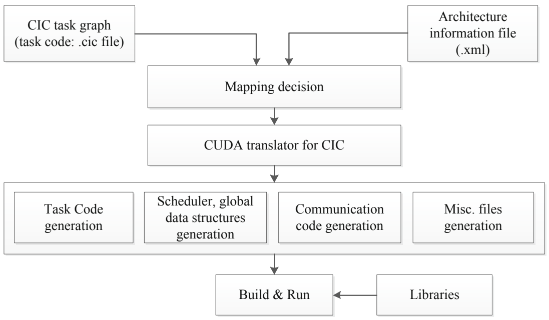


Fig. 1. Overall flow of the proposed CUDA code synthesis framework

Once these decisions have been made, we generate intermediate files for subsequent code synthesis steps. The code synthesizer generates target executable code based on the mapping and configuration information. In task code generation, GPU task is synthesized into a CUDA kernel code. Since the kernel code itself is already given, the synthesizer only adds variable declarations for parameters and includes the header files that declare the generic API prototypes. The code synthesizer also generates a main scheduler function that creates the threads and manages global data structures for tasks, channels, and ports. In the current implementation, we support both POSIX threads and Windows threads. In communication code generation, the synthesizer generates communication code between CPU host and GPU device by redefining the macros such as $MQ\_SEND(), MQ\_RECEIVE()$, used as generic communication APIs in the CIC model. As will be explained in the next section, we support various types of communication methods. This information is kept in an intermediate file (*gpusetup.xml*). There are additional files necessary for building process: for example $Makefile$ help the developer build programs easier.

```
__global__ void Vector_Add(int* C, int* A, int* B){
    const int ix = blockDim.x * blockIdx.x + threadIdx.x;
    C[ix] = A[ix] + B[ix];
}
TASK_GO {
    MQ_RECEIVE(port_input1, DEVICE_BUFFER(input1), sizeof(int) * SIZE);
    MQ_RECEIVE(port_input2, DEVICE_BUFFER(input2), sizeof(int) * SIZE);
    KERNEL_CALL(Vector_Add, DEVICE_BUFFER(output), DEVICE_BUFFER
                    (input1), DEVICE_BUFFER(input2));
    MQ_SEND(port_output, DEVICE_BUFFER(output), sizeof(int) * SIZE);
}
```

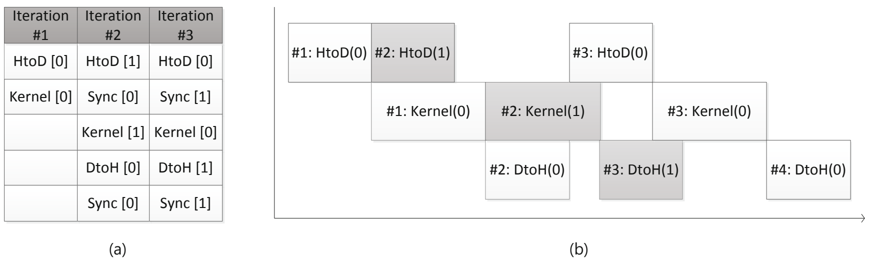
Fig. 2. A CUDA CIC task file (VecAdd.cic)

Fig. 2 shows a simple CIC task code (VecAdd.cic) that defines a CUDA kernel function. The main body of the task is described in the $TASK\_GO$ section that uses various macros whose prototypes are given as templates in the user interface of the proposed framework. The $DEVICE\_BUFFER(port\_name)$ API indicates GPU memory space for the channel and GPU memory allocation code is automatically generated by the code synthesizer relieving the programmers burden. GPU memory de-allocation code is also automatically generated at the wrap-up stage of the task execution. The kernel launch is define by the $KERNEL\_CALL(kernelfunctionname, parameter1, parameter2, ...)$ macro.

5 Code Generation for CPU and GPU Communication

5.1 Synchronous/Asynchronous Communication

CUDA programming platform supports both of synchronous and asynchronous communications. With the synchronous communication method, the CPU task that calls the methods can only proceed after the communication completes. Asynchronous communication is usually better for higher throughput but synchronous communication methods require less memory space than the asynchronous ones where additional memory allocations for *stream* buffers are required. Kernel launches are by default asynchronous executions so that the function call returns immediately and the CPU thread can proceed during the launched kernel is executed. Communications (i.e., memory copy) performed by CUDA APIs that are suffixed with *async* also behave in this manner: instead of blocking the CPU until the memory copy is finished, it returns immediately. Moreover, using *stream*s the communication and kernel execution can be overlapped hiding the communication time as shown in Fig. 3. Kernel executions and memory copy with different *stream*s do not have any dependency, and therefore can be executed asynchronously overlapping their operations. On the contrary, operations with the same *stream* should be serialized. The same *stream* is used



Iteration #1	Iteration #2	Iteration #3
HtoD [0]	HtoD [1]	HtoD [0]
Kernel [0]	Sync [0]	Sync [1]
	Kernel [1]	Kernel [0]
	DtoH [0]	DtoH [1]
	Sync [0]	Sync [1]

Fig. 3. (a) Asynchronous calls with the *stream* ID and (b) the execution timeline of each *stream*

to specify the dependency between CUDA operations and a synchronization function (i.e., *streamSynchronize* ()) denoted as "Sync" in Fig. 3 should be called later to guarantee the completion of the actual copying.

5.2 Bypass/Clustering Communication

To reduce the communication overhead, we define two optimization methods in the proposed code synthesizer, *bypass* and *clustering* as depicted in Fig. 4. By default, the data in a channel is copied into the local buffer in a task. If we want to accelerate the task utilizing GPUs, we should copy the data in the local buffer into the GPU device memory. Hence, it copies the data twice. To reduce such a copy overhead, we implement *bypass* communication. In *bypass* method, the data in the channel is copied to the device memory directly, not through the local buffer.

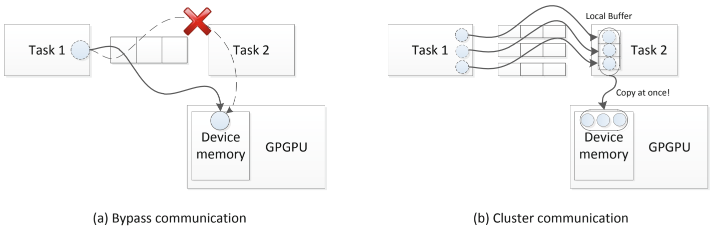


Fig. 4. *Bypass/Clustering* communication

In case there are more than one input/output channels and the size of data is not large, the setup overhead of Direct Memory Access (DMA) becomes significant, sometimes even larger than the actual memory copy overhead. To reduce this overhead, we support *clustering* communication: After the data in all of the input channels are copied into the local buffer inside the task, we send all the data into the device memory at once. This local cluster buffer is declared and allocated by the code synthesizer automatically freeing the programmers from detailed implementation.

$$\sum_{i=1}^{N}(Di * DMAcost + DMAsettingtime) \tag{2}$$

$$\sum_{i=1}^{N}(Di * Memcpycost) + \sum_{i=1}^{N}(Di) * DMAcost + DMAsettingtime \tag{3}$$

(Di: Sample data size of channel ith)

The total execution time of the *bypass* method and the *clustering* method is formalized in equation (2) and (3) respectively. Comparing these two values, we choose the better technique for communication optimization.

6 Experiments

For experiments, we used Intel Core i7 CPU (2.8GHz) with Debian 2.6.32-29 Linux distribution and two Tesla M2050 GPUs. For CUDA programming, we used NVIDIA GPU Computing SDK 3.1 and CUDA toolkit v3.2 RC2.

6.1 Matrix Multiplication

We performed experiments with a simple matrix multiplication example to compare communication overhead between the *bypass* method and the *clustering* method. Two tasks send matrices to a task which multiply two matrices. So there are two input channels in the task.

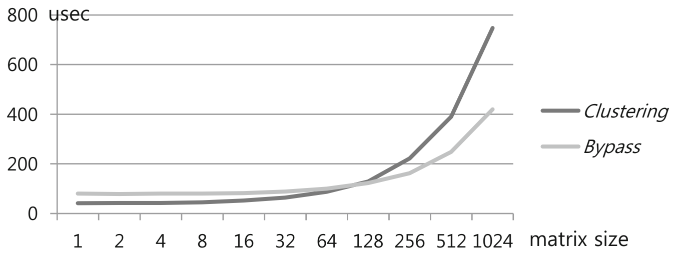


Fig. 5. Communication cost for two communication methods

Fig. 5 shows the communication time (in usec units) of two methods varying the data size (in KBs). When the data size is smaller than 128 KB, it takes less time with the *clustering* method. Otherwise, the *bypass* method is better.

6.2 Lane Detection Algorithm

With a real-life example of lane-detection algorithm, we performed the design space exploration of CUDA programming in the proposed framework. As shown in the Fig. 6, the algorithm consists of two filter chains; one is for detecting the lane in the frame and the other is for providing clearer image display to the driver. Tasks with gray color indicate that they can be mapped to GPU. We used the *Highway.yuv* video clip which consists of 300 frames of HD size (1280x720).

Table 1 shows the execution time of each task on a CPU core and a GPU, obtained by profiling. As can be seen in the table, filter tasks have enough data parallelism to be run on a GPU. Since our target platform contains two GPUs, we can use two GPUs in the mapping stage. As of now we perform manual mapping based on the profiling information of Table 1.

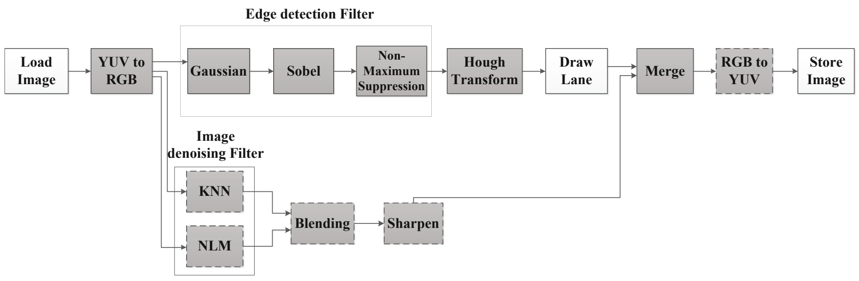


Fig. 6. Task graph of lane detection application

Table 1. Profiling information of tasks (unit: usec)

Task	CPU	GPU	Task	CPU	GPU	Task	CPU	GPU
LoadImage	479	-	KNN	4963268	1615	YUVtoRGB	70226	265
NLM	6911048	13740	Gaussian	389758	1110	Blending	62069	294
Sobel	36616	181	Sharpen	336404	714	Non-max	473716	1752
Merge	45500	245	Hough	369178	2820	RGBtoYUV	76848	300
Draw Lane	3740	-	StoreImage	1068	-	-	-	-

In this experiment, we compared the following three cases: 1) All tasks are mapped on the CPU 2) All GPU-mappable tasks are mapped on a single GPU 3) All GPU-mappable tasks of each filter chain are mapped on each GPU (tasks with solid line in Fig. 6: GPU_0, tasks with dashed line in Fig. 6: GPU_1). Since all tasks have only one or two input ports, we used the *bypass* method. The result is shown in Table 2. Note that we also varied the number of *streams* for asynchronous communication to change the depth of pipelining.

By using one GPU, we could get about 140X speed-up compared using only one CPU core. When we used two GPUs, we could further increase the performance by more than 20% from the gain with one GPU. With asynchronous communications when using one GPU, we could increase the performance gain by 20%. The gain was reduced to 13% when we used two GPUs because the data transfer itself between the device memories in GPU took little time compared to the transfer between the CPU memory and the GPU memory.

Table 2. Results of design space exploration (unit: sec)

CPU	2109.500			
	Sync	Async 2	Async 3	Async 4
1 GPU	12.485	10.654	10.645	10.653
2 GPUs	9.845	9.254	9.168	8.992

* "Async N" denotes asynchronous communication with N *streams*.

7 Conclusions

In this paper, we propose a novel CUDA programming framework that is based on a dataflow model for application specification. The proposed code synthesizer supports various communication methods, so that a user can select suitable communication methods by simply changing the configuration parameters through the GUI. Also we can change the mapping of tasks easily, which increases the design productivity drastically. The proposed methodology could be applied for other platforms such as Cell BE and multi-processor systems. We verified the viability of the proposed technique with the real-life example.

Acknowledgements. This work was supported by Seoul R&BD Program (JP090955), the 2010 Research Fund of the University of Seoul, the MKE (The Ministry of Knowledge Economy), Korea, under the ITRC(Information Technology Research Center), National Research Foundation of Korea Grant funded by the Korean Government (NRF-2011-0013479) and research Grant of Education and Research Foundation College of Engineering, Seoul National University.

References

1. Kirk, D., Hwu, W.: Programming Massively Parallel Processors: A Hands-on Approach, pp. 78–79. Morgan Kaufmann Publisher (2010)
2. Kahn, G.: The semantics of a simple language for parallel programming. In: Proceedings of IFIP Congress, vol. 74, pp. 471–475 (1974)
3. Lee, E.A., Messerschmitt, D.G.: Synchronous Data Flow. Proceedings of the IEEE 75(9), 1235–1245 (1987)
4. Han, T.D., Abdelrahman, T.S.: hiCUDA: A High-level Language for GPU programming. IEEE Transactions on Parallel and Distributed Systems 22(1), 78–90 (2011)
5. Ayguadé, E., Badia, R.M., Igual, F.D., Labarta, J., Mayo, R., Quintana-Ortí, E.S.: An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 851–862. Springer, Heidelberg (2009)
6. Udupa, A., Govindarajan, R., Thazhuthaveetil, M.J.: Software Pipelined Execution of Stream Programs on GPUs. In: Symposium on Code Generation and Optimization, pp. 200–209 (2009)
7. Accelereyes, `http://wiki.accelereyes.com/wiki/index.php/Jacket_Documentation`
8. Kwon, S., et al.: A Retargetable Parallel-Programming Framework for MPSoC. In: TODAES, vol. 13, pp. 1–18 (July 2008)