Cooperative Heterogeneous Computing for Parallel Processing on CPU/GPU Hybrids

Changmin Lee and Won W. Ro
School of Electrical and Electronic Engineering
Yonsei University
Seoul 120-749, Republic of Korea
{exahz, wro}@yonsei.ac.kr

Jean-Luc Gaudiot
Department of Electrical Engineering and
Computer Science, University of California
Irvine, CA 92697-2625
gaudiot@uci.edu

Abstract

This paper presents a cooperative heterogeneous computing framework which enables the efficient utilization of available computing resources of host CPU cores for CUDA kernels, which are designed to run only on GPU. The proposed system exploits at runtime the coarse-grain thread-level parallelism across CPU and GPU, without any source recompilation. To this end, three features including a work distribution module, a transparent memory space, and a global scheduling queue are described in this paper. With a completely automatic runtime workload distribution, the proposed framework achieves speedups as high as 3.08 compared to the baseline GPU-only processing.

1. Introduction

General-Purpose computing on Graphics Processing Units (GPGPU) has recently emerged as a powerful computing paradigm because of the massive parallelism provided by several hundreds of processing cores [4, 15]. Under the GPGPU concept, NVIDIA® has developed a C-based programming model, Compute Unified Device Architecture (CUDA), which provides greater programmability for high-performance graphics devices. As a matter of fact, general-purpose computing on graphics devices with CUDA helps improve the performance of many applications under the concept of a Single Instruction Multiple Thread model (SIMT).

Although the GPGPU paradigm successfully provides significant computation throughput, its performance could still be improved if we could utilize the idle CPU resource. Indeed, in general, the host CPU is being held while the CUDA kernel executes on the GPU devices; the CPU is not allowed to resume execution until the GPU has completed the kernel code and has provided the computation results. The main motivation of our research is to exploit parallelism across the host CPU cores in addition to the GPU cores. This will eventually provide additional computing power for the kernel execution while utilizing the idle CPU cores. Our ultimate goal is thus to provide a technique which eventually exploits sufficient parallelism across heterogeneous processors.

The paper proposes *Cooperative Heterogeneous Computing* (CHC), a new computing paradigm for explicitly processing CUDA applications in parallel on sets of heterogeneous processors including x86 based general-purpose multi-core processors and graphics processing units. There have been several previous research projects which have aimed at exploiting parallelism on CPU and GPU. However, those previous approaches require either additional programming language support or API development. As opposed to those previous efforts, our CHC is a software framework that provides a virtual layer for transparent execution over host CPU cores. This enables the direct execution of CUDA code, while simultaneously providing sufficient portability and backward compatibility.

To achieve an efficient cooperative execution model, we have developed three important techniques:

- A workload distribution module (WDM) for CUDA kernel to map each kernel onto CPU and GPU
- A memory model that supports a transparent memory space (TMS) to manage the main memory with GPU memory
- A global scheduling queue (GSQ) that supports balanced thread scheduling and distribution on each of the CPU cores

We present a theoretical analysis of the expected performance to demonstrate the maximum feasible improvement of our proposed system. In addition, the performance evaluation on a real system has been performed and the results

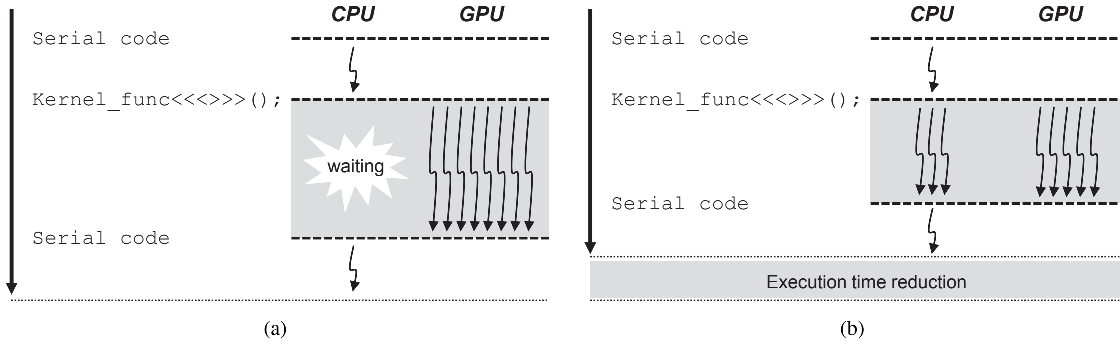

Figure 1. Execution flow of CUDA program: limitation where CPU stalls (a) and cooperative execution in parallel (b).

show that speedups as high as 3.08 have been achieved. On average, the complete CHC system shows a performance improvement of 1.42 over GPU-only computation with 14 CUDA applications.

The rest of the paper is organized as follows. Section 2 reviews related work and Section 3 introduce the existing CUDA programming model and describe motivation of this work. In Section 4, we presents the design and limitations of the CHC framework. Section 5 gives preliminary results. Finally, we conclude this work in Section 6.

2. Related work

There have been several prior research projects which aim at mapping an explicitly parallel program for graphics devices onto multi-core CPUs or heterogeneous architectures. MCUDA [18] automatically translates CUDA codes for general purpose multi-core processors, applying source-to-source translation. This implies that the MCUDA technique translates the kernel source code into a code written in a general purpose high-level language, which requires one additional step of source recompilation.

Twin Peaks [8] maps an OpenCL-compatible program targeted for GPUs onto multi-core CPUs by using the LLVM (Low Level Virtual Machine) intermediate representation for various instruction sets. Ocelot [6], which inspired our runtime system, uses a dynamic translation technique to map a CUDA program onto multi-core CPUs. Ocelot converts at runtime PTX code into an LLVM code without recompilation and optimizes PTX and LLVM code for execution by the CPU. The proposed framework in this paper is largely different from these translation techniques (MCUDA, Twin Peaks, and Ocelot) in that we support cooperative execution for parallel processing over both CPU cores and GPU cores.

In addition, EXOCHI provides a programming environment that enhances computing performance for media kernels on multicore CPUs with Intel® Graphics Media Accelerator (GMA) [20]. However, this programming model uses the CPU cores only for serial execution. The Merge framework has extended EXOCHI for the parallel execution on CPU and GMA; however, it still requires APIs and the additional porting time [13]. Lee et al. have presented a framework which aims at porting an OpenCL program on the Cell BE processor [12]. They have implemented a runtime system that manages software-managed caches and coherence protocols.

Ravi et al. [17] have proposed a compiler and a runtime framework that generate a hybrid code running on both CPU and GPU. It dynamically distributes the workload, but the framework targets only for generalized reduction applications, while our system targets to map general CUDA applications. Qilin [14], in the most relevant study to our proposed framework, has shown an adaptive kernel mapping using a dynamic work distribution. The Qilin system trains a program to maintain databases for the adaptive mapping scheme. In fact, Qilin requires and strongly relies on its own programming interface. This implies that the system cannot directly port the existing CUDA codes, but rather programmers should modify the source code to fit their interfaces. As an alternative, CHC is designed for seamless porting of the existing CUDA code on CPU cores and GPU cores. In other words, we focus on providing backward compatibility of CUDA runtime APIs.

3. Motivation

One of the major roles of the host CPU for the CUDA kernel is limited to controlling and accessing the graphics devices, while the GPU device provides a massive amount of data parallelism. Fig. 1(a) shows an example where the host controls the execution flow of the program only, while

the device is responsible for executing the kernel. Once a CUDA program is started, the host processor executes the program sequentially until the kernel code is encountered. As soon as the host calls the kernel function, the device starts to execute the kernel with a large number of hardware threads on the GPU device. In fact, the host processor is held in the idle state until the device reaches the end of the kernel execution.

As a result, the idle time causes an inefficient utilization of the CPU hardware resource of the host machine. Our CHC system is to use the idle computing resource with concurrent execution of the CUDA kernel on both CPU and GPU (as described in Fig. 1(b)). Considering that the future computer systems are expected to incorporate more cores in both general purpose processors and graphics devices, parallel processing on CPU and GPU would become a great computing paradigm for high-performance applications. This would be quite helpful to program a single chip heterogeneous multi-core processor including CPU and GPU as well. Note that Intel® and AMD® have already shipped commercial heterogeneous multi-core processors.

In fact, CUDA is capable of enabling *asynchronous concurrent execution* between host and device. The concurrent execution returns a control to the host before the device has completed a requested task (i.e., non-blocking). However, the CPU that has the control can only perform a function such as memory copy, setting other input data, or kernel launches using *streams*. The key difference on which we focus is in the use of idle computing resources with concurrent execution of the same CUDA kernel on both CPU and GPU, thereby easing the GPU burden.

4. Design

An overview of our proposed CHC system is shown in Fig. 2. It contains two runtime procedures for each kernel launched. Each kernel execution undergoes those procedures. The first includes the *Workload Distribution Module* (WDM), designed to apply the distribution ratio to the kernel configuration information. Then, the modified configuration information is delivered to both the *CPU loader* and the *GPU loader*. Two sub-kernels ($Kernel_{CPU}$ and $Kernel_{GPU}$) are loaded and executed, based on the modified kernel configurations produced by the WDM.

The second procedure is designed to translate the PTX code into the LLVM intermediate representation (LLVM IR). As seen in Fig. 2, this procedure extracts the PTX code from the CUDA binary to prepare the LLVM code for cooperative computing. On the GPU device, our runtime system passes the PTX code through the CUDA device driver, which means that the GPU executes the kernel in the original manner using the PTX-JIT compilation. On the CPU

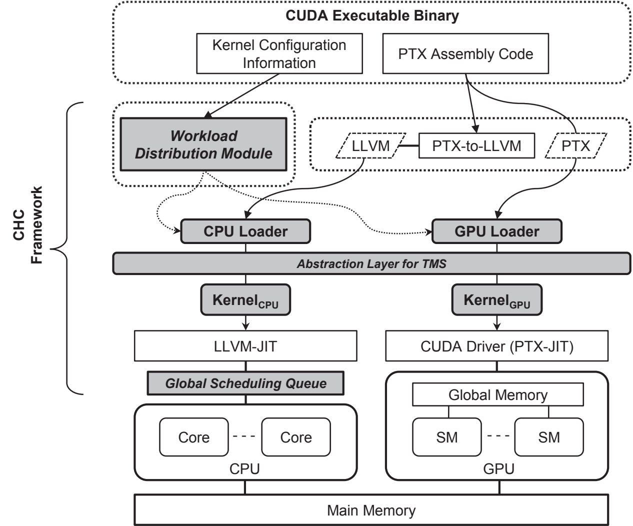

Figure 2. An overview of the CHC runtime system.

core side, CHC uses the PTX translator provided in Ocelot in order to convert PTX instructions into LLVM IR [6]. This LLVM IR is used for a kernel context of all thread blocks running on CPU cores, and LLVM-JIT is utilized to execute the kernel context [11].

The CUDA kernel execution typically needs some start-up time to initialize the GPU device. In the CHC framework, the GPU start-up process and the PTX-to-LLVM translation are simultaneously performed to hide the PTX-to-LLVM translation overhead.

4.1. Workload distribution module and method

The input of WDM is the kernel configuration information and the output specifies two different portions of the kernel, each for CPU cores and the GPU device. The kernel configuration information contains the execution configuration which provides the dimension of a grid and that of a block. The dimension of a grid can be efficiently used for our workload distribution module.

In order to divide the CUDA kernel, the workload distribution module determines the amount of the thread blocks to be detached from the grid considering the dimension of the grid and the workload distribution ratio as depicted in Fig. 3. As a result, WDM generates two additional execution configurations, one for CPU and the other for GPU. WDM then delivers the generated execution configurations (i.e., the output of the WDM) to the CPU and GPU loaders. With these execution configurations, each loader now can make a sub-kernel by using the kernel context such as

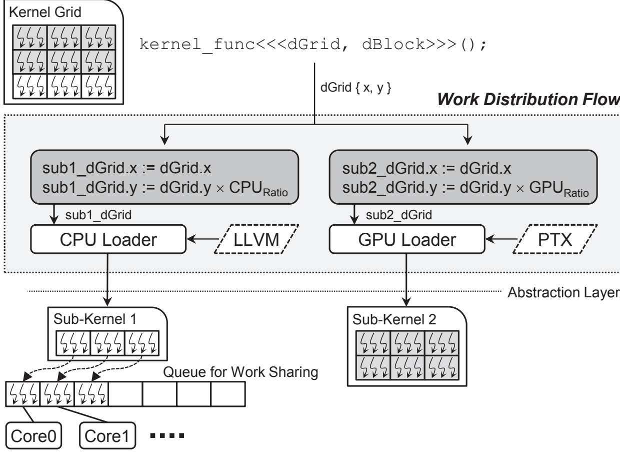

Figure 3. Work distribution flow and kernel mapping to CPU and GPU.

LLVM and PTX.

Typically, WDM assigns the front portion of thread blocks to the GPU-side, while the rest is assigned to the CPU-side. Therefore, the first identifier of the CPU's sub-kernel will be $(dGrid.y \times GPU_{Ratio}) + 1$. Then, each thread block can identify the assigned data with the identifier since both sides have an identical memory space.

In order to find the optimal workload distribution ratio, we can probably predict the runtime behavior such as the execution delay on CPU cores. However, it is quite hard to predict characteristics of a CUDA program since the runtime behavior strongly relies on dynamic characteristics of the kernel [1, 10]. For this reason, Qilin used an empirical approach to achieve their proposed adaptive mapping [14]. In fact, our proposed CHC also adopts a heuristic approach to determine the workload distribution ratio. Then, the CHC framework performs the dynamic work distribution at runtime based on this ratio. The proposed work distribution can split the kernel according to the granularity of thread block.

4.2. Memory consolidation for transparent memory space

A programmer writing CUDA applications should assign memory spaces in the device memory of the graphics hardware. These memory locations (or, addresses) are used for the input and output data. In the CUDA model, data can be copied between the host memory and the dedicated memory on the device. For this purpose, the host system should preserve *pointer variables* pointing to the location in the device memory.

As opposed to the original CUDA model, *two* different memory addresses exist for *one* pointer variable in our proposed CHC framework. The key design problem is caused by the fact that the computation results of the CPU side are stored into the main memory that is different from the device memory. To address this problem, we propose and design an abstraction layer, *Transparent Memory Space* (TMS), to preserve two different memory addresses in a pointer variable at a time.

Accessing memory addresses. The abstraction layer uses *double pointers* data structures (similar to [19]) for pointer variables to map *one* pointer variable onto *two* memory addresses: for the main memory and the device memory. As seen in Fig. 4, we have declared the abstraction layer that manages a list of the TMS data structures. Whenever a pointer variable is referenced, the abstraction layer translates the pointer to the memory addresses, for both CPU and GPU. For example, when a pointer variable (e.g., *d_out*) is used to allocate device memory using `cudaMalloc()`, the framework assigns memory spaces both on the device memory and the host memory. The addresses of these memory spaces are stored in a TMS data structure (e.g., *TMS1*), and the framework maps the pointer variable on the TMS data structure. Thus, the runtime framework can perform the address translation for a pointer variable.

Launching a kernel. For launching a kernel, pointer variables defined in advance may be used as arguments of the kernel function. At that time, the CPU and GPU loaders obtain each translated address from the mapping table so that each sub-kernel could retain actual addresses on its memory domain.

Merging separated data. After finishing the kernel computation, the computation results are copied to the host memory (`cudaMemcpy()`) to perform further operations. Therefore, merging the data of two separate memory domains is required. To reduce memory copy overhead, the framework traces memory addresses which are modified by the CPU-side computation.

4.3. Global scheduling queue for thread scheduling

GPU is a throughput-oriented architecture which shows outstanding performance with applications having a large amount of data parallelism [7]. However, to achieve meaningful performance from the CPU side, scheduling thread blocks with an efficient policy is important.

Ocelot uses a locality-aware *static* partitioning scheme in their proposed thread scheduler, which assigns each thread block considering load balancing between neighboring worker thread [6]. However, this static partitioning method probably causes some cores to finish their execution early. In our scheduling scheme, we allow a thread block to be assigned dynamically to any available core. For this pur-

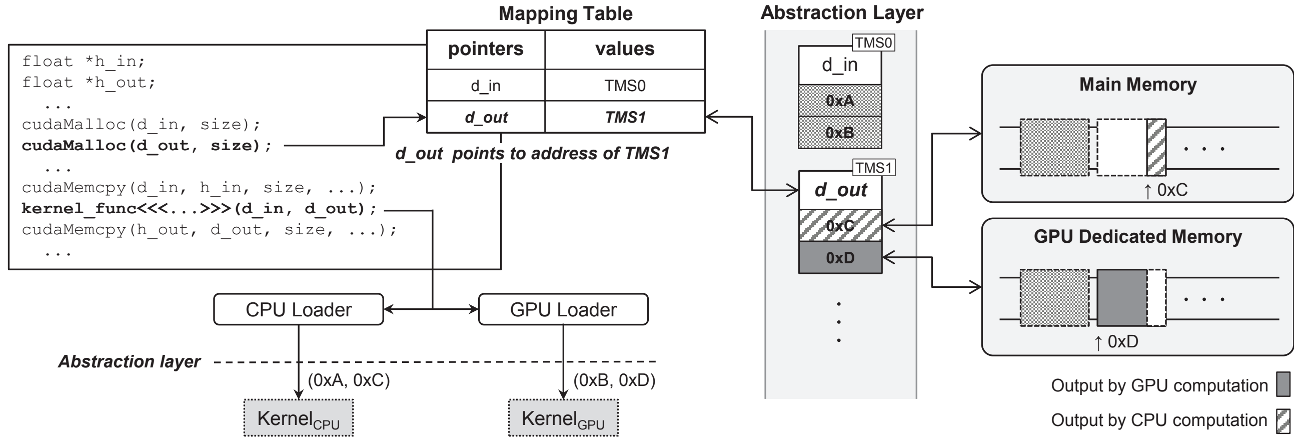

Figure 4. Anatomy of transparent memory space.

pose, we have implemented a work sharing scheme using a *Global Scheduling Queue* (GSQ) [3]. This scheduling algorithm enqueues a task (i.e., a thread block) into a global queue so that any worker thread on an available core can consume the task. Thus, this scheduling scheme allows a worker thread in each core to pick up only one thread block and achieve load balancing. In addition, any core which finishes the assigned thread block so early would handle another thread block without being idle.

4.4. Limitations on global memory consistency

CHC emulates the global memory on the CPU-side as well. Thread blocks in the CPU can access the emulated global memory and perform the atomic operations. However, our system does not allow the global memory atomic operations between the thread blocks on the CPU and the thread blocks on the GPU to avoid severe performance degradation. In fact, discrete GPUs have their own memory and communicate with the main memory through the PCI express, which causes long latency problems. This architectural limit suggests that the CHC prototype need not provide global memory atomic operations between CPU and GPU.

5. Results

The proposed CHC framework has been fully implemented on a desktop system with two Intel Xeon^{TM} X5550 2.66 GHz quad-core processors and an NVIDIA GeForce^{TM} 9400 GT device. The aim of the CHC framework is to demonstrate the feasibility of the parallel kernel execution on CPU and GPU to improve CUDA execution on low-end GPUs. This configuration is also applicable to a single-chip heterogeneous multi-core processor that has an integrated GPU, which is generally slower than discrete GPUs.

We adapt 14 CUDA applications which do not have the global memory synchronization across CPU and GPU at runtime; twelve from the NVIDIA CUDA Software Development Kit (SDK) [16], SpMV [2], and MD5 hashing [9]. Table 1 summarizes these applications and kernels.

From left to right the columns represent the application name, the number of computation kernels, the number of thread blocks in the kernels, a description of the kernel, and work distribution ratio used in the CHC framework. We measured the execution time of kernel launches and compared CHC framework against the GPU-only computing. The validity of the CHC results was compared to a computation result that has been executed on a CPU only.

5.1. Initial analysis

For the initial analysis, we have measured the execution delay using only the GPU device and the delay using only the host CPU (through the LLVM JIT compilation technique [5, 6, 11]). In addition, the workload has been configured either as executing only one thread block or as executing the complete set of thread blocks.

The maximum performance improvement achievable based on the initial execution delays can be experimentally deduced, as depicted in Table 2. Fig. 5 shows the way to find it; the x-axis represents the workload ratio in terms of thread blocks assigned to the CPU cores against thread blocks on the GPU device. With having more thread blocks on the CPU cores, fewer thread blocks would be assigned to the GPU device. Therefore, the execution delay for GPU is proportionally reduced along the x-axis.

From the above observation, the maximum value between the CPU execution delay and the GPU execution delay at a given workload ratio can be considered as the total

Table 1. Test Applications

Applications (Abbreviation)	# Kernels	# Thread Blocks	Description
3DFD (3DFD)	1	20x20	3D finite difference computation
Binomial Options Pricing (BINO)	1	512x1	European options under binomial model
Black Scholes (BLKS)	1	480x1	European options by Black-Scholes formula
Mersenne Twister (MERT)	2	32x1	Mersenne twister random number generator and Cartesian Box-Muller transformation
Matrix Multiplication (MAT)	1	128x128	Matrix multiplication: C = A * B.
Monte Carlo (MONT)	2	256x1	European options using Monte Carlo approach
Scalar Product (SCALAR)	1	128x1	Scalar products of input vector pairs
Scan (SCAN)	3	256x1	Parallel prefix sum
Convolution Texture (CONV)	2	192x128	Image convolution filtering
Transpose (TRANS)	2	128x256	Matrix transpose
Sobol QRNG (QRNG)	1	1x100	Sobol's quasi-random number generator
Vector Addition (VEC)	1	196x1	Vector addition: C = A + B
Sparse matrix-vector multiplication[a] (SPMV)	2	1024x1	Matrix-vector multiplication: y += A * x
MD5 Hashing (MD5)	2	33312x1	MD5 hashing (MD5 calculation and search)

a. Compressed Sparse Row (CSR) format is used.

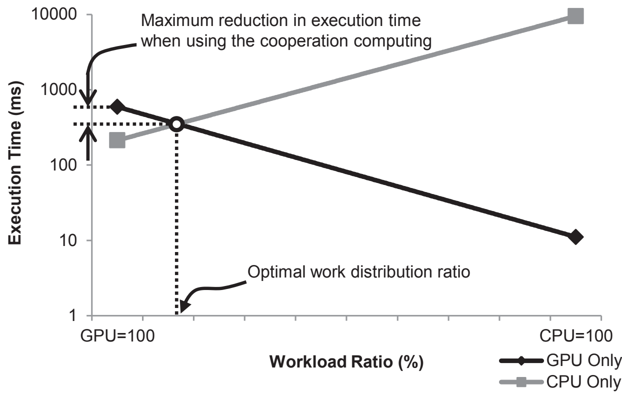

Figure 5. Prediction of performance improvement with initial analysis.

execution delay. Therefore, the crossing point of two lines in Fig. 5 gives the optimal distribution ratio as well as the maximum performance with the initial data and the mathematical analysis.

5.2. Performance improvements of CHC framework

Table 2 shows the maximum performance and the optimal distribution ratio obtained from the initial analysis. In addition, the actual execution time and the actual work distribution ratio using CHC are also presented. In fact, the optimal distribution ratio is used to determine the work distribution ratio on CHC.

Fig. 6 shows the performance improvements of CHC normalized to GPU-only computations according to the actual distribution ratio; as expected, the performance of CHC improves compared to the execution delay using only GPU. The speedup is achieved, ranging from 0.46x for `MERT` up to 3.08x for `VEC`. The average speedup of the CHC framework is 1.42x.

More in detail, the applications with exponential, trigonometric, or power arithmetic operations (`BINO`, `BLKS`, `MERT`, `MONT`, and `CONV`) show little performance improvement. In fact, the execution time of these applications on CPU is much higher compared to the execution time on GPU. This is due to the fact that the GPU device normally provides special functional units for those operations. On the other hand, the applications without those arithmetic operations (`TRANS`, `VEC`, `SPMV`, and `MD5`) show relatively higher speedups.

Table 2. Initial analysis and CHC results

Abbreviation	Thread Blocks	Workload	Execution Time (ms)		Initial Analysis		CHC Results	
			CPU Only	*GPU Only*	*Maximum Performance*	*Optimal Ratio (CPU:GPU)*	*Actual Performance*	*Actual Ratio (CPU:GPU)*
3DFD	20x20	20x1	19.468	7.144	104.52	26.8:73.2	117.32	10.0:90.0
		20x20	389.36	142.88				
BINO	512x1	1x1	17.43	1.15	556.06	6.2:93.8	559.44	4.7:95.3
		512x1	8926.6	593.0				
BLKS	480x1	1x1	1.08	0.03	16.98	3.3:96.7	17.02	0.6:99.4
		480x1	520.20	17.557				
MERT	32x1	1x1	13.62	1.55	44.73	10.3:89.7	108.26	3.1:96.9
		32x1	435.94	49.85				
MAT	128x128	128x1	44.23	12.84	1274.15	22.5:77.5	1453.19	18.0:82.0
		128x128	5661.8	1644.1				
MONT	256x1	1x1	8.47	0.28	69.90	3.2:96.8	79.39	3.1:96.9
		256x1	2170.1	72.23				
SCALAR	128x1	1x1	0.22	0.04	4.79	16.6:83.4	4.86	25.0:75.0
		128x1	28.86	5.74				
SCAN	256x1	1x1	0.014	0.004	0.87	23.2:76.8	1.01	9.4:90.6
		256x1	3.75	1.13				
CONV	192x128	192x1	8.45	0.15	19.51	1.8:98.2	19.51	0.8:99.2
		192x128	1082.3	19.53				
TRAN	128x256	128x1	0.20	0.25	28.81	55.5:44.8	29.29	50.0:50.0
		128x256	52.21	64.29				
ORNG	1x100	1x1	0.05	0.01	0.87	21.8:78.2	0.90	16.0:84.0
		1x100	5.22	1.46				
VEC	196x1	1x1	0.005	0.011	0.68	68.5:31.5	0.70	44.9:55.1
		196x1	1.01	2.17				
SPMV	1024x1	1x1	0.003	0.001	1.17	38.2:61.8	1.45	25.0:75.0
		1024x1	3.07	1.90				
MD5	33312x1	1x1	0.002	0.004	52.14	64.7:35.3	56.34	64.0:36.0
		33312x1	80.55	147.84				

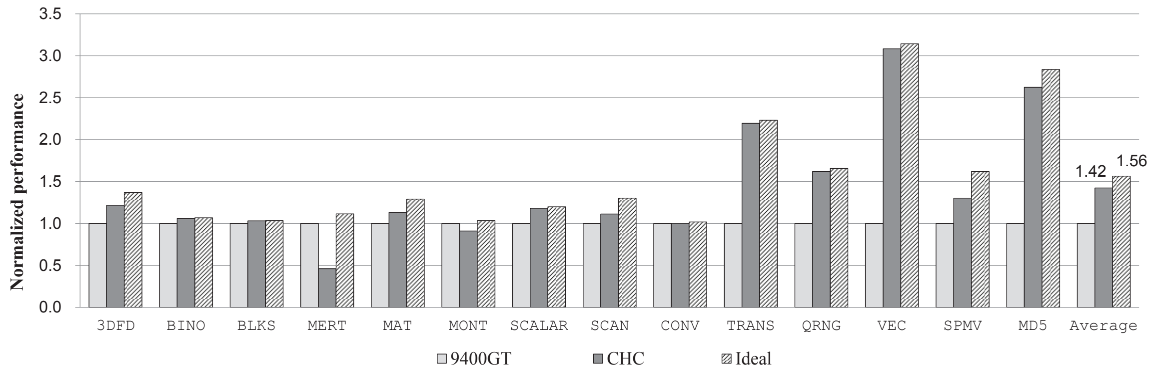

Figure 6. Normalized performance speedup of CHC over the GPU-only processing.

6. Conclusions

The paper has introduced three key features for the efficient exploitation of the thread level parallelism provided by CUDA on the CPU multi-cores in addition to the GPU device. The proposed CHC framework provides a tool set which enables CUDA binary to run on CPU and GPU, without imposing source recompilation. The experiments demonstrate that the proposed framework successfully achieves efficient parallel execution and that the performance results obtained are close to the values deduced from the theoretical analysis. We believe the cooperative heterogeneous computing can be utilized in the future heterogeneous multi-core processors which are expected to include even more GPU cores as well as CPU cores.

As future work, we will first develop a dynamic control scheme on deciding the workload distribution ratio. We also plan to design an efficient thread block distribution technique considering data access patterns and thread divergence. We believe CHC can eventually provide a solution for the degradation of performance due to the irregular memory access and thread divergence in the original

CUDA execution model. In fact, the future CHC framework needs to address the performance trade-offs considering the CUDA application configurations on various GPU and CPU models. In addition, we will discuss the overall speedups considering the transition overhead to find the optimal configuration for the CHC execution model.

7. Acknowledgments

We thank all of the anonymous reviewers for their comments. This work was supported by the Basic Science Research Program through the National Research Foundation (NRF) of Korea, which is funded by the Ministry of Education, Science and Technology [2009-0070364]. This work is also supported in part by the US National Science Foundation under Grant No. CCF-0541403. Any opinions, findings, and conclusions as well as recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163 –174, april 2009.

[2] N. Bell and M. Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations, 2010.

[3] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46:720–748, September 1999.

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

[5] C. Cifuentes and V. M. Malhotra. Binary translation: Static, dynamic, retargetable? In *Proceedings of the 1996 International Conference on Software Maintenance*, ICSM '96, pages 340–349, Washington, DC, USA, 1996. IEEE Computer Society.

[6] G. F. Diamos, A. R. Kerr, S. Yalamanchili, and N. Clark. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 353–364, New York, NY, USA, 2010. ACM.

[7] M. Garland and D. B. Kirk. Understanding throughput-oriented architectures. *Commun. ACM*, 53:58–66, Nov. 2010.

[8] J. Gummaraju, L. Morichetti, M. Houston, B. Sander, B. R. Gaster, and B. Zheng. Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 205–216, New York, NY, USA, 2010. ACM.

[9] M. Juric. Cuda md5 hashing.

[10] A. Kerr, G. Diamos, and S. Yalamanchili. A characterization and analysis of ptx kernels. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 3–12, Washington, DC, USA, 2009. IEEE Computer Society.

[11] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[12] J. Lee, J. Kim, S. Seo, S. Kim, J. Park, H. Kim, T. T. Dao, Y. Cho, S. J. Seo, S. H. Lee, S. M. Cho, H. J. Song, S.-B. Suh, and J.-D. Choi. An opencl framework for heterogeneous multicores with local memory. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 193–204, New York, NY, USA, 2010. ACM.

[13] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. Merge: a programming model for heterogeneous multi-core systems. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 287–296, New York, NY, USA, 2008. ACM.

[14] C.-K. Luk, S. Hong, and H. Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 45 –55, dec. 2009.

[15] J. Nickolls and W. Dally. The gpu computing era. *Micro, IEEE*, 30(2):56 –69, march-april 2010.

[16] NVIDIA. Nvidia cuda sdks.

[17] V. T. Ravi, W. Ma, D. Chiu, and G. Agrawal. Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 137–146, New York, NY, USA, 2010. ACM.

[18] J. A. Stratton, S. S. Stone, and W.-M. W. Hwu. Languages and compilers for parallel computing. chapter MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs, pages 16–30. Springer-Verlag, Berlin, Heidelberg, 2008.

[19] C. Tian, M. Feng, and R. Gupta. Supporting speculative parallelization in the presence of dynamic data structures. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 62–73, New York, NY, USA, 2010. ACM.

[20] P. H. Wang, J. D. Collins, G. N. Chinya, H. Jiang, X. Tian, M. Girkar, N. Y. Yang, G.-Y. Lueh, and H. Wang. Exochi: architecture and programming environment for a heterogeneous multi-core multithreaded system. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 156–166, New York, NY, USA, 2007. ACM.