Applied Mathematics & Information Sciences
An International Journal

http://dx.doi.org/10.12785/amis/081L42

The Comparisons of OpenCL and OpenMP Computing Paradigm

Slo-Li Chu[*] *and Chih-Chieh Hsiao*

Department of Information and Computer Engineering, Chung Yuan Christian University, Chung Li, 32023, Taiwan

Received: 18 May. 2013, Revised: 12 Sep. 2013, Accepted: 13 Sep. 2013
Published online: 1 Apr. 2014

Abstract: Graphics processing units (GPUs) in a computer system are increasingly adopted to create a realistic environment in 3D applications. Despite the extremely high parallelism of these devices with a tremendous amount of processing elements, GPUs are seldom used in scientific applications owing to their difficulty in programming. Additionally, vendor-provided APIs are too specific to cross platform. An open standard, OpenCL, has subsequently been developed to provide universal APIs and programming paradigms for various GPUs. This work demonstrates the capabilities of OpenCL with several platforms, based on a preliminary example. The proposed benchmarks with different attributes are implemented by OpenMP and OpenCL sequentially to compare their differences. The benchmarks are then targeted on several GPUs and conventional servers. Experimental results indicate that inexpensive GPUs perform better than servers if OpenCL paradigms are adopted.

Keywords: Heterogeneous Platform, Parallel Programming, OpenCL, OpenMP, Graphics Processing Units, Supercomputing.

1 Introduction

The growing 3D applications have led computer systems into a new computing era. Complex objects in 3D scenes have significantly increased the requirement of computational power of computer systems. Such computer systems adopt add-on graphics processing unit (GPU) cards or streaming processors, subsequently forming a heterogeneous platform to satisfy higher workload demands. More realistic 3D scenes have increased the requirement of computing power of GPUs or streaming processors, even dominating the main processor of a computer system. Therefore, some parallel programming paradigms have been developed to utilize the capabilities of these heterogeneous platforms, including CUDA [1], and OpenCL [2]. This work describes several workloads with different parallel programming paradigms, then targets on conventional multi-core processors and these heterogeneous platforms. Finally, their performances are compared, especially difficulties in coding programs.

Previously, programmers needed to familiarize themselves with graphics APIs or vendor specific APIs to program parallel applications for high-performance heterogeneous computer systems. These APIs and programming paradigms are extremely difficult to implement. Meanwhile, conventional parallel programming paradigms e.g., OpenMP [6] and MPI are infeasible for GPU-based heterogeneous platforms. GPU vendors have provided CUDA and CAL in recent years to utilize the computational power of GPUs. However, designed for specific platforms, these parallel paradigms have difficulty in migrating to other platforms [11,12]. Therefore, to solve this problem, industries have collaborated in establishing an open standard for programming such platforms, the open computing language also known as OpenCL. Since establishment of this standard, the code written in OpenCL can be easily migrated between these diverse architectures without modifying parallel programs.

This study, based on our early work [5], examines more compiling combinations for these architectures and presents an example to illustrate source code migration from OpenMP to OpenCL, which utilizes these ubiquitous devices. Several benchmarks are also designed to evaluate the performance of these platforms.

The rest of this paper is organized as follows. Section 2 reviews the background of these architectures. Section 3 then describes two parallel programming paradigms and compares their differences. Next, Section 4 summarizes

[*] Corresponding author e-mail: slchu@cycu.edu.tw

the experimental results. Conclusions are finally drawn in Section 5.

2 Evaluation Platforms

This section reviews the hardware platforms used in this work

2.1 ATi Radeon HD5800 series

As the latest GPU family announced by ATi in 2009, Radeon HD5800 series [3] is designed mainly for 3D game rendering with over 2720 GFLOPs computing capability. Figure 1 shows the HD5800 architecture. The main computing power originates from 20 SIMD engines. Each engine consists of 16 thread processors with 5 stream cores as fundamental processing elements, in which all of the chips contain 1600 stream cores. Each stream core can perform IEEE754 compatible floating point operations, even fused-multiply-add operations.

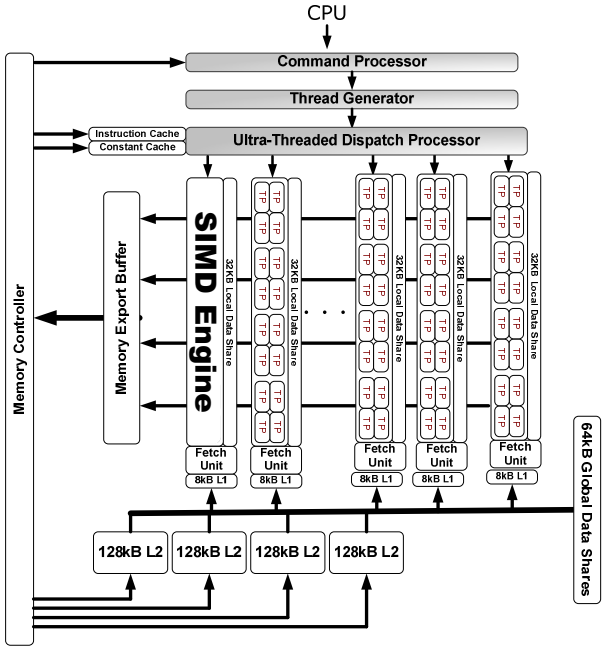

Fig. 1: The organization of ATi HD5800 series.

This architecture is designed by mixing SIMD and VLIW approaches. From a high-level perspective, the architecture consists of 20 SIMD engines; each SIMD engine has a width of 16 SIMD machines, where each of its elements is called a thread processor with a five-issue, VLIW-style design, consisting of four basic stream cores and responsible for typical operations like floating and integer add, sub, multiply. The architecture also contains one core for special functions, including sine, cosine, and reciprocal functions. Additionally, a branch unit in each thread processor solves simple branch problems during execution. Moreover, the architecture contains large general purpose registers (GPRs) to maintain the thread status for thread switching.

2.2 nVidia GeForce GT200 series

nVidia announced its GT200 series [4] GPU in early 2009. Its design focuses on both 3D graphics rendering and general purpose computing capabilities with over 1060 GFLOPs. Figure 2 shows the architecture of nVidia GT200.

The GT200 architecture consists of ten texture/processor clusters (TPCs). Each TPC consists of three streaming multiprocessors (SMs) and a texture unit. The main computing elements in GT200 are streaming processors (SPs), as shown in Fig. 2, where the eight SPs work in a single instruction multiple data (SIMD) manner, or so referred to as single program multiple data (SPMD) by nVidia. Both SPs and special function units (SFUs) share the same instructions, data and instruction L1 cache, instruction retrieval and dispatch unit in a SM. A GT200 GPU has a total of 240 SPs.

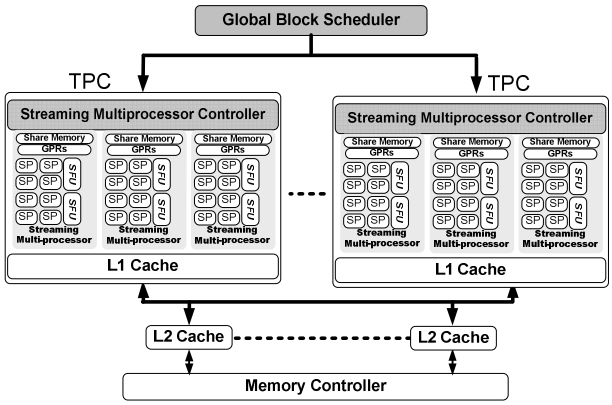

Fig. 2: The organization of nVidia GT200 series.

2.3 IBM Cell Processor

Figure 3 illustrates the architecture of IBM Cell processor [7]. Power processor element (PPE) is the main control unit of the cell processor, and synergistic processor elements (SPEs) are designed for data-intensive and streaming processing computations. PPE consists of a power processor unit (PPU) which is an in-order, 2-way simultaneous multi-threading, 64-bit power architecture with VMX extension, with L2 cache. SPE consists of a

synergistic processor unit (SPU) and its memory flow control synergistic memory flow (SMF) control. Each SPU has a dual-issue 128-bit SIMD architecture with deep pipelining. These processor elements, memory controller, and I/O are attached onto an element interconnect bus (EIB) [9], which is implemented as a circular ring comprising four 16B-wide unidirectional channels with counter-rotating in pairs. To access the external memory, SPEs can rely only on DMA to move data from/to the memory controller. Meanwhile, the memory controller is shared by all processor elements, thus limiting the memory bounded applications and programming paradigm. Additionally, the Cell processor in Sony PlayStation3 enables only six SPEs.

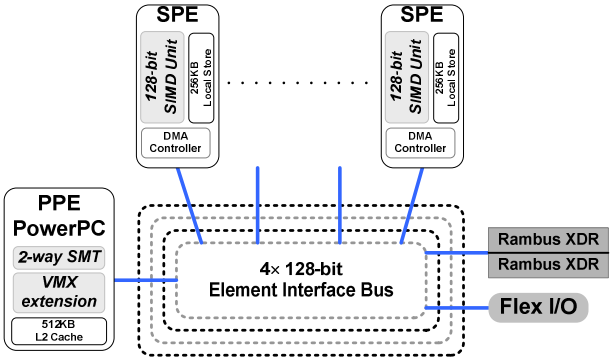

Fig. 3: The organization of IBM Cell processor.

2.4 Sun UltraSPARC T2

Sun UltraSPARC T2 [10] is used in this work as one of the reference machines for demonstrating OpenMP. Figure 4 show the architecture of UltraSPARC T2. Each SPARC core is a 2-way superscalar with fine-grained multi-threading of eight threads. A total of 64 threads run in the same time over eight cores. The memory subsystem of UltraSPARC T2 is quite efficient since it uses multiple memory banks and four memory controllers to increase the total memory bandwidth. Although the working frequency is lower than other workstation products, this feature improves the performance in memory bounded applications.

2.5 Summary

Table 1 [3,4,7,10,13] lists the architectural configurations of the computing devices studied in this work. Some of the entries are inconsistent with previous discussions since its high-end version is lacking in this work. For instance, HD5870 has 1,600 PEs while HD5850 in this work has only 1,440 PEs.

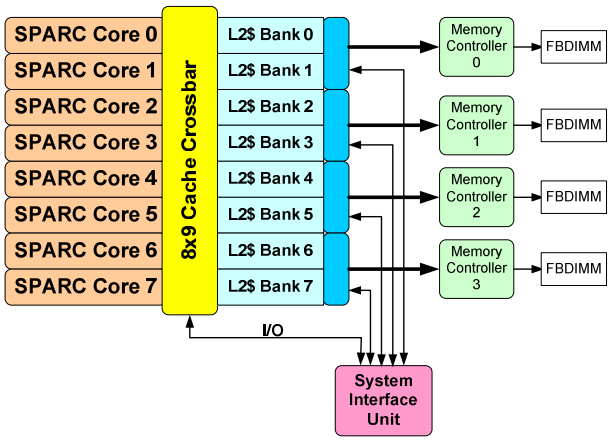

Fig. 4: The organization of Sun UltraSPARC T2.

Table 1: The summary of evaluated architectures.

	Sun UltraSPARC T2	Intel Xeon E5440	IBM Cell in PS3	nVidia GeForce GTX285	ATi Radeon HD5850
Clock rate of PE	1.4GHz	2.83GHz	3.2GHz	1.47GHz	725MHz
Number of PE	8	4	1+6	240	1440
GFLOPS (single)	11.2	45.28	204	1062.72	2088
GFLOPS (double)	11.2	45.28	15	89	418
Transistor count	503M	825M	241M	1.4B	2.15B

3 Parallel Programming Paradigms

This section introduces two programming paradigms and, then, discusses their differences. Finally, an example illustrates how to transform OpenMP parallelized programs into OpenCL codes.

3.1 OpenMP Programming Model

Open multi-processing (OpenMP) [6], a parallel programming paradigm, supports shared memory multiprocessing programming in C, C++, and Fortran, as well as focuses on many parallel systems, including Unix and Microsoft Windows based platforms. This paradigm consists of a set of compiler directives, library routines, and environment variables that influence the run-time behavior of a program.

As a multithreading paradigm, OpenMP consists of a ”master thread” (i.e. a series of instructions executed consecutively) and a specified number of ”slave threads” spawned by the master thread. Accordingly, the task is divided and parallelized by these threads, Fig. 5. These threads are then executed concurrently, managed by a

runtime environment that can spawn threads and allocate necessary resources for these threads.

To apply OpenMP accurately, programmers must analyze their program and identify the parallelizable part to add appropriate compiler directives. Consider a program that must contain shared variables or reductions in the loops that attempted to be parallelized. That program must be protected and written in a directive to inform a compiler and OpenMP runtime. When the program is highly parallelizable, OpenMP can easily enhance its performance with additional threads on an increasing number of processing elements.

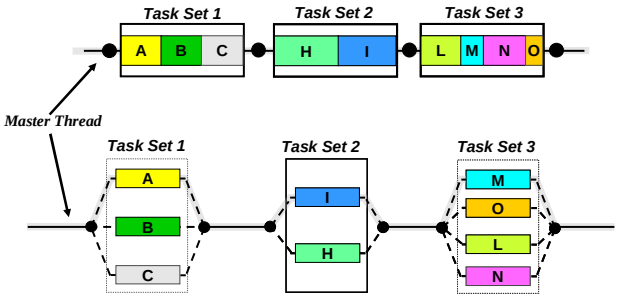

Fig. 5: The OpenMP programming paradigm.

3.2 OpenCL Programming Model

OpenCL [2] is an open standard for cross-platform, parallel programming paradigm of modern processors in a computer system. The target of OpenCL is written accelerated portable code across different devices and architectures with a wide range of applications. Therefore OpenCL can utilize of CPUs, GPUs and other DSPs to accelerate compute-intensive or data parallel applications.

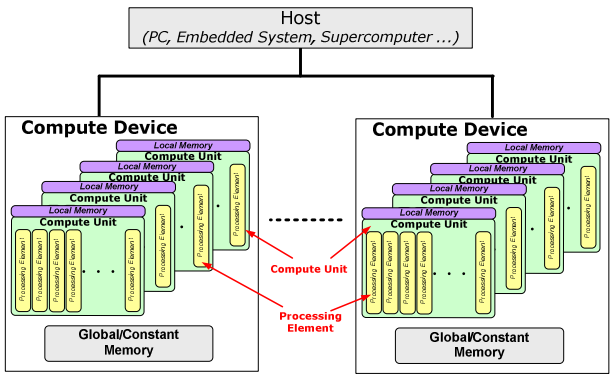

Fig. 6: The OpenCL programming paradigm.

Figure 6 shows the OpenCL programming model. The host may be a personal computer, embedded system or super computer, which provides OpenCL APIs and a runtime compiler. A computer device might be a CPU, GPU, DSP or Cell processor, which executes OpenCL kernels written in C99-based OpenCL language. Each computer device has multiple computer units consisting of multiple processing elements. The local data shares and shared memory in ATi and nVidia GPUs are designed to map this memory hierarchy. All memory data transfers by work-items must be copied explicitly through the host memory, global memory, and local memory and, then, sent back. It implies that programmers must handle memory management explicitly. To write a program with a satisfactory performance in OpenCL, programmers must familiarize themselves with underlying architectures to adjust appropriate parameters for execution. A host program is necessary to establish the entire environment for OpenCL execution. However, as long as it is written, the host program can act as a template to migrate to different platforms with simple modifications, as shown in Fig. 7.

Despite the difficulty of writing an OpenCL program, porting to different platforms requires only a few modifications, as shown in Fig. 7. For GPUs, an appropriate number of work-items and work-items per work-group must be set, Fig. 7(a)(b). To port on the IBM Cell, not only must the number of work-items and work-items per work group be changed, but the device type must also be modified as an ACCELERATOR, Fig. 7(c). The current ATi OpenCL implementation also supports x86 CPUs, which involves modifying the device type as a CPU, Fig. 7(d).

```
…
#define TOTAL_WOKERS   1024
…
 size_t localThreads[1] = {64};
…
 if (!strcmp(pbuf, \
    "Advanced Micro Devices, Inc."))
     break;
…
hContext = clCreateContextFromType( cprops,\
  CL_DEVICE_TYPE_GPU, NULL, NULL, \
  &status);
…
```

(a) OpenCL Codes for ATi GPUs

```
…
#define TOTAL_WOKERS   1024
…
 size_t localThreads[1] = {64};
…
 if (!strcmp(pbuf, "NVIDIA Corporation"))
     break;
…
hContext = clCreateContextFromType( cprops,\
  CL_DEVICE_TYPE_GPU, NULL, NULL, \
  &status);
…
```

(b) OpenCL Codes for nVidia GPUs

```
…
#define TOTAL_WOKERS   6
//No. of available SPUs
…
 size_t localThreads[1] = {1};
…
 if (!strcmp(pbuf, "IBM"))
     break;
…
hContext = clCreateContextFromType( cprops,\
  CL_DEVICE_TYPE_ACCELERATOR, \
  NULL, NULL, &status);
…
```

(c) OpenCL Codes for IBM Cell

```
…
#define TOTAL_WOKERS   4   //No. of CPU
cores
…
 size_t localThreads[1] = {1};
…
 if (!strcmp(pbuf, \
    "Advanced Micro Devices, Inc."))
     break;
…
hContext = clCreateContextFromType( cprops,\
  CL_DEVICE_TYPE_CPU, NULL, NULL,\
  &status);
…
```

(d) OpenCL Codes for AMD x86 CPUs

Fig. 7: Porting OpenCL host code to different platforms.

3.3 Migrating from OpenMP to OpenCL

This subsection discusses how OpenMP and OpenCL differ by using a simple example, ”Prime Number.” Figure 8(a) shows the corresponding OpenMP version. Since variable ***prime*** is independent of each parallel section and inaccessible to other threads, this variable must be declared as ”private” by OpenMP directive. A parallel reduction is then performed on variable sum, which must be protected by OpenMP when multi-threaded execution performed with ”reduction” declaration in a directive. OpenMP runtime divides the loop into multiple threads with appropriate iterations when executed.

```
int prime_number ( int n )
{
   int I,  j, prime;
   int total = 0;

# pragma omp parallel  \
   shared ( n )            \
   private ( i, j, prime )    \
   reduction ( + : total )

# pragma omp for
   for ( i = 2; i <= n; i++ )          {
        prime = 1;

        for ( j = 2; j < i; j++ )    {
            if ( i % j == 0 )   {
                prime = 0;
                break;   }
        }
        total = total + prime;
   }
   return total;
}
```

(a) OpenMP Codes of Prime Number.

```
__kernel void prime_number(
__global uint *partial_result, const
uint num, const uint chunk)
{
   uint gid = get_global_id(0);
   uint i, j, prime;
   uint total = 0;
   uint slice = num/chunk;
   int lower_bound, upper_bound;
   lower_bound = gid * slice;
   upper_bound = (gid +1) * slice;

   for ( i = lower_bound;
         i < upper_bound; i++ )     {
         prime = 1;
         for ( j = 2; j < i; j++ )    {
             if ( i % j == 0 )     {
                 prime = 0;
                 break;      }
         }
         total = total + prime;
   }
   partial_result[gid] = total;
}
```

(b) OpenCL Codes of Prime Number.

Fig. 8: Migrating OpenMP program to OpenCL version.

Two considerations must be concerned for migrating this OpenMP program to OpenCL. First, programmers specify the number of iterations in each work-item, and programmers must be taken care when writing programs. Second, the parallel reductions must be hand-coded. During this migration, the parallel reduction operations are left for the host processor to finish in order to simplify the OpenCL kernel.

Each OpenCL kernel must begin with *__kernel* declaration, and attributes of input data must be indicated. Input argument ***partial_result*** is used to return the partial results of each work-item to the host for parallel reduction. Another argument ***chunk*** specifies the number of global work-items in execution. In this kernel, each work-item calculates a small portion of iterations, which are specified by ***lower_bound*** and ***upper_bound***. The global ID ***gid***, as retrieved by *get_global_id* call, calculates the initial value in each work-item. Once calculated, the results are stored into ***partial_result*** with indexing by ***gid***, and wait to copy back to the host for parallel reduction. The OpenCL kernels are normally saved as a text file with *.cl* file extension. The kernel file can be easily migrated to different architectures with an appropriate *NDRange* execution specified by OpenCL API of the host without modifications.

4 Experimental Results

This section describes the compiling combinations of five platforms and two parallel programming paradigms by using four benchmarks. Table 2 lists the environmental conditions for the experiment. In the following figures, the ”OMP” and ”OCL” denote the programs are coding by ”Open MP” and ”OpenCL” paradigms, respectively. The ”icc” and ”gcc” denote the programs are compiled by Intel C Compiler and GNU C Compiler, respectively. The ”suncc -fast” and ”suncc” represent the programs are compiled by Sun C compiler with and without -fast options, respectively. The ”Static” and ”Dyn” mean that the programs are compiled statically and compiled dynamically, respectively. The ”Static” compilation can save execution time since the compiling time will not be counted into execution time. In contrast, the ”Dyn” compilation will waste the execution time to compile the program on-the-fly.

Table 2: The experimental characteristics of five target platforms.

	Sun UltraSPARC T2	Intel Xeon E5440	IBM Cell in PS3	nVidia GeForce GTX285	ATi Radeon HD5850
OS	SunOS 5.10	Linux 2.6.9 (32-bit)	Linux 2.6.21 (64-bit)	Windows7 (32-bit)	Windows7 (32-bit)
Host	UltraSPARC T2	Intel Xeon E5440 x 2	Power Processor Element	Intel Core-i5 750	Intel Core-i5 750
Compiler	Sun C 5.9	GCC 4.1.1, Intel C Compiler 11.0	GCC 4.1.2	Visual Studio 2008	Visual Studio 2008
OpenCL runtime	N/A	N/A	Cell SDK 3.1.0 + OpenCL SDK 0.1.1	GPU Computing SDK 2.3A	StreamSDK 2.0.1

The first benchmark, pi calculation, is a highly computation-intensive program. Performance of the OpenCL test is observed using various numbers of work-items. Figure 9 summarizes the experimental results of pi calculating with OpenCL, where Y-axis denotes the execution time and X-axis refers to the number of work-items on four platforms. The ”optimized” denotes that benchmark is optimized exhaustively to reveal the lower bounds of four platforms.

Since both GPU vendors do not provide offline OpenCL compiler, the execution times of two GPUs must include kernel compilation time. IBM provides offline

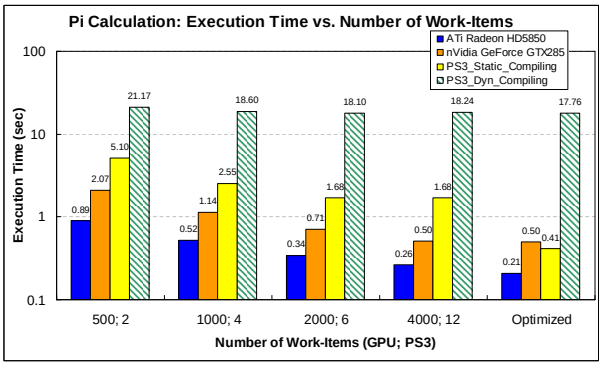

Fig. 9: The execution time of Pi Calculation by OpenCL paradigm.

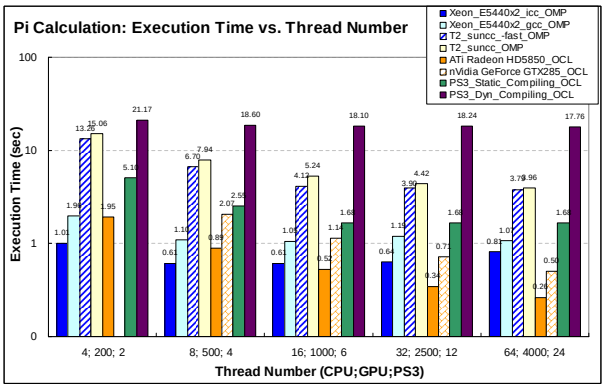

Fig. 10: The execution time of Pi Calculation with OpenMP and OpenCL paradigms on various platforms.

and runtime kernel compilers; the results of PS3 thus have dynamic and static compilation versions. In both GPUs, the number of work-items is allocated from 500 to 4,000. However, the number of processing elements is less in PS3 cases and the numbers of work-items are 2, 4, 6 and 12. The evaluated results indicate that the best number of work-items per work group for ATi GPU is multiple of 64, according to the suggestion of the developer's guide [8]. The 64 work-items can be packed into an ATi GPU thread called "wavefront" in ATi HD5800 series. Although the best number of work-items per work group of nVidia GPU is 32 since 32 threads can be packed into a warp, nVidia's runtime is set to 1, significantly affect the performance. To improve performance, this value is set to 100 manually for GTX285. For PS3, the number of work-items per work group is specified by its runtime, which is 1. Because SPEs are not designed for multithreading execution, increasing the number of work-items per work group degrades the performance. According to the experimental results, an increasing number of work-items imply a more improved performance, even when a GPU has more work-items than processing elements. In the IBM Cell processor, the amount of SPEs is the upper bound of improved performance, since it is not a multithreaded design and unable to achieve fast thread switching.

According to the profiling results, the computing kernel does not fully utilize these devices, since the OpenCL runtime compiler fails to pack instructions into ATi GPU's five-issue VLIW and IBM Cell's four-way SIMD engine efficiently. Moreover, according to our analytical results, if this kernel can be vectorized, the performance is improved, as shown in "Optimized".

Figure 10 shows the performance results in each platform and parallel paradigms. The x-axis in the bottom portion of this figure denotes the number of threads for CPU, GPU, and OpenMP, respectively. When the number of work-items is insufficient, the conventional superscalar out-of-order processor Intel Xeon is the fastest one. Also, the Intel C compiler can generate better quality codes than GNU C compiler. However, the GPU outperforms CPUs with an increasing number of work-items. ATi GPU is nearly quadruple faster than the optimized dual-socket Xeon server when there are 4,000 work-items. This execution time includes OpenCL kernel compilation and memory data movement between the host and GPU. If only the execution times are compared, ATi GPU performs nearly 20 times faster than UltraSPARC T2 due to its higher frequency. Moreover, PS3_Static performs worse than GPUs since the number of computer units is significantly less than that of GPUs; however, it still outperforms UltraSPARC T2.

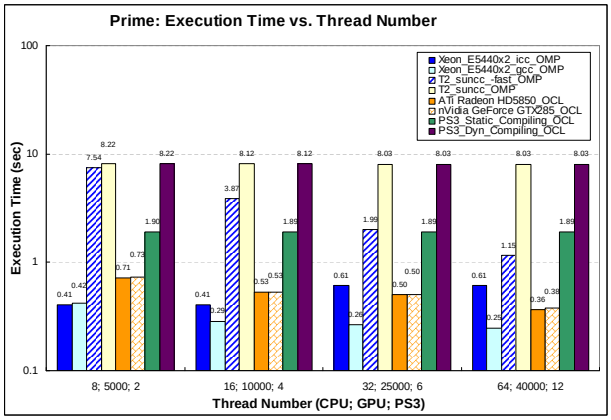

Fig. 11: The execution time of Prime Number with OpenMP and OpenCL paradigms on various platforms.

The second benchmark calculates the number of prime numbers within integers 2 to 100,000, as shown in Fig. 11. Since the brute-force method requires branch operations to search for and determine the prime numbers in each iteration, the performance of GPUs is limited due to the divergent threads in GPU [9].

The best time achieved by the dual-socket Xeon server is 0.25 (gcc version) while that for ATi GPU is 0.36 and that for nVidia GPU is 0.38. Results of this application indicate that GPU performance is limited in applications such as those with frequently used branches. In PS3_Static, the performance is improved by only 0.5% when increasing the number of SPEs from 2 to 12, since it has a weak branch capability. However, another highly-threaded UltraSPARC T2 adopts a fine-grained multi-threading scheme to mitigate the performance loss from branches and even enhanced by fast switching among threads. The performance enhancement saturated at 64 threads is achieved using the maximum number of concurrent threads in UltraSPARC T2 and obtained 6.6 times speedup in -fast version.

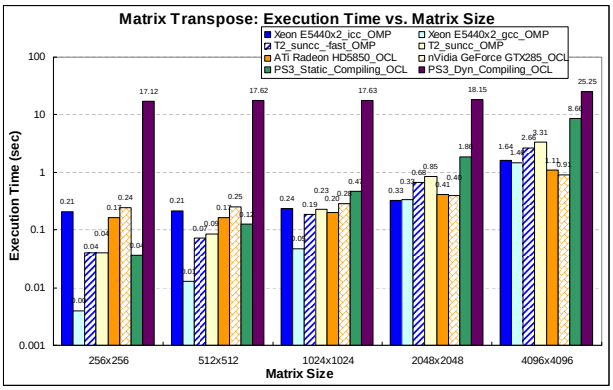

Fig. 12: The execution time of Matrix Transposes with OpenMP and OpenCL paradigms on various platforms.

The next two benchmarks are matrix transpose and matrix multiplication with various data sizes, which achieve maximum capabilities of these platforms. For the dual-socket Xeon server, 8 threads and 16 threads are used for UltraSPARC T2. In the matrix transpose, the number of work-items is equivalent to the size of the matrix in the OpenCL kernel. Additionally, matrix multiplication is performed with tiling of size 4x4, such that the number of work-items equals (matrix width/4) x (matrix height/4). The number of work-items per work group is set to 8x8 for both benchmarks. Figures 12 and 13 summarize those results.

The matrix operations the above two benchmarks are memory bounded, in which a large memory bandwidth is required. When the matrix size is less than or equal to 1024x1024, the operation on CPU is faster than that of GPUs since it must transfer data into the GPU onboard memory before processing and copy back when done. Also, the growing matrix size consumes more latency for data movement than computation. Accordingly, matrix multiplication, UltraSPARC T2 outperforms the Intel Xeon server. Since the memory subsystem in UltraSPARC T2 is quad on-chip dual-banked memory controllers, rather than the memory subsystem in dual-socket Xeon server, a single off-chip memory controller is shared. This characteristic also diminishes the limitation of a lower CPU frequency of UltraSPARC T2. In PS3, all memory accesses must be sent to EIB and arbitrated to share external memory access. However, both GPUs are over 100 times faster than the dual-socket Xeon server and 30 times faster than UltraSPARC T2 in multiplication with the matrix size of 2048x2048, due to their extreme parallelisms by massive processing elements.

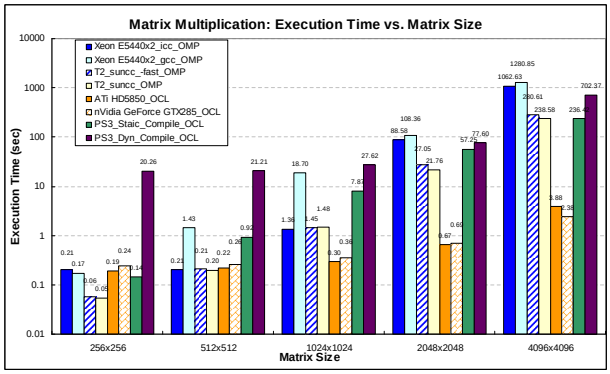

Fig. 13: The execution time of Matrix Multiplications with OpenMP and OpenCL paradigms on various platforms.

In these benchmarks, except for the ”Prime” with many branches, GPUs significantly outperform commodity multi-core processors, even when using expensive servers. Since at least one-third of the execution time in GPU is spent in compiling the kernel and data transfer, if the data size increases and the operations become more complex, GPUs significantly outperform commodity CPUs. GPUs have large number of processing elements and perform comparative to supercomputers. However, whereas a server costs more than $US 10,000, the GPU can achieve a similar performance with less than $US 500. Additionally, OpenCL can extract computational power from GPUs rather than OpenMP on a heterogeneous platform with supercomputing capability. Nevertheless, the parallel programming paradigm of OpenCL is more complex than that of OpenMP, and the performance extracted from GPU is less expensive and efficient than the conventional parallel programming paradigm with homogeneous platforms. Moreover, OpenCL kernels are used across different architectures without any modifications, thus achieving portability on different computing devices.

5 Conclusions

The computational capabilities of GPUs increase annually. An open standard for computation on

heterogeneous platforms, OpenCL, utilizes heterogeneous CPUs and GPUs. Four benchmarks and five hardware platforms are evaluated. According to our results, fewer branches operations improve the performance; meanwhile, these devices are not handled well for branch operations. If the kernel code is developed properly, the computational capabilities can be released from these device and achieve a high performance at a significantly lower cost than when using a supercomputer. In contrast to conventional OpenMP parallelized programs that target on expensive servers, the OpenCL parallelized applications that target on relatively inexpensive GPUs.

Acknowledgement

This work is supported in part by the National Science Council of Republic of China, Taiwan under Grant NSC 100-2221-E-033-043.

References

[1] nVidia, Inc., CUDA Compute Unified Device Architecture Programming Guide Ver. 2.0, nVidia, (2008). www.nvidia.com.
[2] The OpenCL Specification 1.0 rev.48, Khronos OpenCL Working Group, (2009).
[3] AMD Corp., AMD Graphics for Desktop PCs, www.amd.com.
[4] nVidia Corp., nVidia GeForce Family, www.nvidia.com.
[5] S.-L. Chu, and C.-C. Hsiao, Proc. 12th IEEE International Conference on High Performance Computing and Communications, 556 (2008).
[6] OpenMP Specification Ver. 3.0, OpenMP Architecture Review Board, (2008).
[7] IBM Corp., Cell Broadband Engine Resource Center, www.ibm.com.
[8] ATi Stream Computing, OpenCL Programming Guide, ATi., (2010).
[9] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer and K. Skadron, Journal of Parallel and Distributed Computing, **68**, 1370 (2008).
[10] Oracle Corp., UltraSPARC T2: a highly-threaded, power efficient, SPARC SOC, www.oracle.com.
[11] S. H. Abbas, Applied Mathematics & Information Sciences, **2**, 225-235 (2008).
[12] W. K. Wootters & W. H. Zurek, A single quantum cannot be cloned, Nature, **299**, 802-803 (1982).
[13] Intel Corp., Intel Microprocessor Export Compliance Metrics, www.intel.com.

Slo-Li Chu received his PhD degree in Electrical Engineering from National Sun Yat-sen University, Taiwan, in 2002. He is currently an associate professor of Department of Information and Computer Engineering, Chung Yuan Christian University, Taiwan. His research interests include computer architectures, parallelizing compilers, system level modeling, system-on-chip design, embedded system design, interconnection networks, multicore architectures and GPU architectures.

Chih-Chieh Hsiao received his BS degree in Information and Computer Engineering from Chung Yuan Christian University, Taiwan, in 2007, and the MS degree in Computer Science from National Chiao Tung University, Taiwan, in 2009. He received his PhD degree in Electronic Engineering at Chung Yuan Christian University, in 2013. His research interests include computer architectures and GPU architectures.