Performance Characterization of the NAS Parallel Benchmarks in OpenCL[*]

Sangmin Seo Gangwon Jo Jaejin Lee

Center for Manycore Programming
School of Computer Science and Engineering, Seoul National University, Seoul, 151-744, Korea
{sangmin, gangwon}@aces.snu.ac.kr, jlee@cse.snu.ac.kr
`http://aces.snu.ac.kr`

Abstract

Heterogeneous parallel computing platforms, which are composed of different processors (e.g., CPUs, GPUs, FPGAs, and DSPs), are widening their user base in all computing domains. With this trend, parallel programming models need to achieve portability across different processors as well as high performance with reasonable programming effort. OpenCL (Open Computing Language) is an open standard and emerging parallel programming model to write parallel applications for such heterogeneous platforms. In this paper, we characterize the performance of an OpenCL implementation of the NAS Parallel Benchmark suite (NPB) on a heterogeneous parallel platform that consists of general-purpose CPUs and a GPU. We believe that understanding the performance characteristics of conventional workloads, such as the NPB, with an emerging programming model (i.e., OpenCL) is important for developers and researchers to adopt the programming model. We also compare the performance of the NPB in OpenCL to that of the OpenMP version. We describe the process of implementing the NPB in OpenCL and optimizations applied in our implementation. Experimental results and analysis show that the OpenCL version has different characteristics from the OpenMP version on multicore CPUs and exhibits different performance characteristics depending on different OpenCL compute devices. The results also indicate that the application needs to be rewritten or re-optimized for better performance on a different compute device although OpenCL provides source-code portability.

[*]This work was supported in part by grant 2009-0081569 (Creative Research Initiatives: Center for Manycore Programming) from the National Research Foundation of Korea. This work was also supported in part by the Ministry of Education, Science and Technology of Korea under the BK21 Project. ICT at Seoul National University provided research facilities for this study.

1 Introduction

As accelerating processors, such as GPUs, DSPs, and FPGAs, have become popular in high performance computing domains, heterogeneous parallel platforms that are composed of processors with different types are widening their user base. Moreover, hardware vendors already have launched hybrid processors, such as Intel Sandy Bridge [14] and AMD Fusion [2], which integrates multiple CPU and GPU cores in the same chip. Such hybrid processors are expected to reduce the communication overhead between processors with different types in a heterogeneous parallel platform. The overhead has been a serious concern in such a heterogeneous parallel platform.

With this trend, it is important for a parallel programming model to achieve both source-code and performance portability with reasonable programming effort for different processors in the heterogeneous parallel platform. If the programmer has to use a mix of parallel programming models (e.g., MPI and CUDA) to exploit heterogeneous processors in the same system, it would hurt productivity and make debugging hard.

OpenCL (Open Computing Language) [18] is an open standard to write parallel applications for the heterogeneous systems. If an application is written in OpenCL once, it can run on any processor or any mix of processors that supports OpenCL. Currently, OpenCL is gaining much attention of many processor vendors, such as AMD [3], Intel [15], IBM [13], and NVIDA [25]. They have released OpenCL frameworks for a variety of processors since late 2008.

In this paper, we characterize the performance of an OpenCL implementation of the NAS Parallel Benchmarks (NPB) [22] on a heterogeneous parallel platform that consists of general-purpose CPUs and a GPU. The NPB is a well-known benchmark suite for evaluating high performance multiprocessor systems. We choose the NPB because understanding the performance characteristics of con-

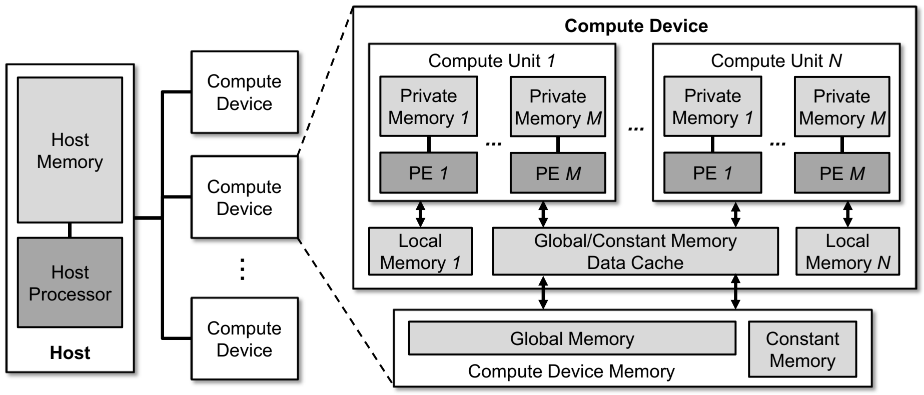


Figure 1. The OpenCL platform model

ventional workloads, such as the NPB, with an emerging programming model (i.e., OpenCL) is important for the adoption of the programming model to developers and researchers.

The current NPB suite includes diverse implementations of the same application in MPI, OpenMP, High Performance Fortran (HPF), and Java. However, they were written only for the execution on general-purpose CPUs in mind. Since a different processor architecture exhibits a different performance behavior for the same workload, to improve the performance of an entire application on a heterogeneous parallel platform, it is important to understand the performance characteristics of the workload for different types of processors. By implementing the NPB in OpenCL and characterizing its performance on CPUs and GPUs, we enhance the current NPB suite and our characterization will give new insights to the conventional workload in heterogeneous parallel computing. Furthermore, our OpenCL implementation of the NPB will help researchers to develop and evaluate their ideas in architectures (e.g., GPGPUs), compiler optimizations, and runtime designs.

The contributions of this paper are the following:

- We describe our OpenCL implementations and optimizations of the NPB applications (NPB-OpenCL) for multicore CPUs (two Intel Xeon X5660s) and a GPU (NVIDIA GeForce GTX 480).
- We evaluate NPB-OpenCL on the CPUs and the GPU. We analyze and characterize the performance evaluation result.
- We compare the performance of NPB-OpenCL with that of the NPB written in OpenMP for multicore CPUs.
- We discuss the performance portability of OpenCL applications and the effectiveness of OpenCL.

Our work is a complete implementation of the entire NPB in OpenCL and is the first evaluation of all the NPB applications for the heterogeneous computing system. The source code of NPB-OpenCL is publicly available at the URL `http://aces.snu.ac.kr`.

The rest of the paper is organized as follows. Section 2 briefly describes an OpenCL framework and its implementation on CPUs and GPUs. Section 3 explains the OpenCL implementation of the NPB. Section 4 presents and characterizes the performance evaluation result of NPB-OpenCL. Section 5 discusses the device-specific optimizations and the performance portability of OpenCL. Section 6 reviews related work and Section 7 concludes the paper.

2 OpenCL Architecture

This section briefly introduces the standard OpenCL architecture [18] and its implementations for general-purpose CPUs and GPUs.

2.1 OpenCL Models

Platform model. Figure 1 shows the OpenCL platform model that consists of a *host* connected to one or more *compute devices*. A compute device is divided into one or more *compute units* (CUs), each of which is further divided into one or more *processing elements* (PEs).

Execution model. An OpenCL program consists of two parts: *kernels* that run on one or more compute devices and a *host program* that runs on the host and manages the execution of kernels. The host program enqueues a *command* to a *command-queue* that is attached to a compute device. OpenCL has three types of commands: kernel execution, memory management, and synchronization.

An abstract index space called NDRange is used to execute a kernel. NDRange is an N-dimensional space, where N is one, two or three. It is defined by an N-tuple of integers and specifies the dimension and size of the index space. An instance of the kernel, called *work-item*, is executed for each point in the index space. One or more work-items are organized into *work-groups*. The work-group provides a more coarse-grained decomposition of the index space and is assigned a unique work-group ID. A work-item is assigned a unique local ID within a work-group to which it belongs. Thus, a work-item can be uniquely identified by its point in the index space (i.e., global ID) or by a combination of its local ID and work-group ID. All work-items in a work-group execute the same code concurrently on the PEs in a single CU, but the specific execution pathway and the data used can vary per work-item.

Memory model and synchronization. OpenCL defines four distinct memory regions in a compute device: global, constant, local, and private. As shown in Figure 1, *compute device memory* consists of the global and constant memory regions. These regions are shared by all CUs in a compute device (i.e., all work-items in all work-groups). Accesses to the global memory or the constant memory may be cached in *global/constant memory data cache* (Figure 1). The local memory is shared by all PEs in a CU (i.e., all work-items in a single work-group), and the private memory is accessed by only the PE (i.e., private to each work-item).

There are two domains of synchronization in OpenCL: work-items in a single work-group and commands enqueued to command-queue(s). A work-group barrier is used to synchronize work-items in a single work-group. A command-queue barrier or an event is used to synchronize commands. However, there is no synchronization mechanism between work-groups.

OpenCL defines a relaxed memory consistency model. An update to a memory location by a work-item may not be visible to all the other work-items at the same time. Instead, the local view of memory from each work-item is guaranteed to be consistent at synchronization points, such as work-group barriers, command-queue barriers, and events.

2.2 OpenCL on a Multicore CPU System

Table 1 shows the mapping between the OpenCL platform model (Figure 1) and a parallel system that consists of multicore CPUs. The mapping is based on the AMD's OpenCL framework [10]. Since our target system uses Intel multicore CPUs (Intel Xeon X5660) that support hyperthreading, each physical core supports two logical cores. In this mapping, a set of logical cores become a compute device and a logical core becomes the host processor. The host processor core may be shared with the compute device. Each logical core in the compute device plays the role of a CU. Both of the host memory and the device memory in the OpenCL platform model are allocated in the main memory of the multicore system.

Since the number of all work-items for a kernel is typically much larger than the number of logical cores in the compute device, assigning a single OS thread to each work-item incurs significant scheduling overhead due to context switching. Moreover, work-group barriers also incur a significant overhead of context switching between work-items. AMD's OpenCL framework avoids this scheduling overhead by assigning a single OS thread to all work-items in the same work-group and executing work-items one by one sequentially. It uses lightweight `setjmp()/longjmp()` functions [10] to switch contexts between work-items when there is a barrier. Another solution is using the work-item coalescing technique [20, 31] that coalesces work-items in the same work-group with a loop and executing the loop with a single OS thread on a logical core. Consequently, PEs in a CU are virtualized by the logical core that corresponds to the CU.

Table 1. Mapping the OpenCL platform model to the multicore CPU system

OpenCL	Multicore CPU System
Host processor	Logical core
Host memory	Main memory
Compute device	A set of logical cores
CU	Logical core
PE	Virtualized PE by a logical core
Global memory	Main memory
Constant memory	Main memory
Local memory	Main memory
Private memory	Main memory

Table 2. Mapping the OpenCL platform model to the GPU system

OpenCL	GPU System
Host processor	CPU
Host memory	Main memory
Compute device	GPU
CU	Streaming multiprocessor
PE	Scalar processor
Global memory	GPU global memory
Constant memory	GPU constant memory
Local memory	GPU shared memory
Private memory	GPU registers

2.3 OpenCL on a GPU System

Table 2 shows the mapping between the OpenCL platform model and a typical GPU system that consists of a CPU and a GPU. Since our target GPU system uses an NVIDIA GPU (GeForce GTX 480), we use the terminology used in the CUDA architecture [26] in Table 2. The CPU in the system becomes the host processor and the GPU becomes a compute device. Since OpenCL has a strong inheritance from CUDA, the mapping between the OpenCL components and those of the GPU architecture is one-to-one and straightforward.

A CU in the GPU executes hundreds of work-items concurrently. To manage such a large number of work-items, the GPU employs a SIMT (Single-Instruction, Multiple-Thread) architecture, where work-items execute one common instruction at a time. The CU creates, manages, sched-

```
#pragma omp parallel for \
    default(shared) private(i,j,k)
for (k = 0; k < d3; k++) {
  for (j = 0; j < d2; j++) {
    for (i = 1; i < d1; i++) {
      x[k][j][i] += x[k][j][i-1];
    }
  }
}
```

(a) OpenMP code

```
__kernel void scan1(__global int *g_x)
{
  int i, j, k;
  __global int (*x)[d2][d1];
  x = (__global int (*)[d2][d1])g_x;

  k = get_global_id(0);
  if (k < d3) {
    for (j = 0; j < d2; j++)
      for (i = 1; i < d1; i++)
        x[k][j][i] += x[k][j][i-1];
  }
}
```

(b) 1-D kernel

```
__kernel void scan2(__global int *g_x)
{
  int i, j, k;
  __global int (*x)[d2][d1];
  x = (__global int (*)[d2][d1])g_x;

  k = get_global_id(0);
  j = get_global_id(1);
  if (k < d3 && j < d2) {
    for (i = 1; i < d1; i++)
      x[k][j][i] += x[k][j][i-1];
  }
}
```

(c) 2-D kernel

Figure 2. OpenCL kernels for an OpenMP parallel for. `d1`, `d2`, and `d3` are constant.

ules, and executes work-items in a group of 32. This group is called a *warp* in the CUDA architecture. Work-items in a warp are executed in a SIMT way.

3 OpenCL Implementation of the NPB

This section describes our OpenCL implementation of the NPB. The NPB is a set of applications designed to evaluate the performance of parallel computers [22]. The NPB applications are derived from computational fluid dynamics (CFD) applications for aerospace research. The NPB consists of five kernels – EP, IS, CG, MG, and FT, and three pseudo applications – SP, BT, and LU. The five kernels perform computational cores of different numerical methods that are typically used by many CFD applications. The three pseudo applications simulate computation and data movement observed in typical CFD applications.

We port those eight applications in the NPB 3.3.1 to OpenCL. The NPB 3.3.1 includes MPI, OpenMP, High Performance Fortran (HPF), and Java versions of the applications. For MPI and OpenMP versions, only IS is written in C and other seven applications are written in Fortran. Our porting follows the OpenCL specification version 1.0 [18].

3.1 Baseline Implementation

The OpenMP version of the NPB 3.3.1 serves as the source of our baseline implementation in OpenCL. We assume the OpenCL platform has a host processor and a compute device. First, we port the OpenMP Fortran applications to OpenMP C. Then, OpenMP parallel loops and parallel regions in the resulting applications are converted into OpenCL kernels. Since typical OpenMP applications exploit data-level parallelism, the conversion is easy. The following is the rules under which our conversions are made:

- A thread in the OpenMP parallel region becomes a work-item in the converted OpenCL kernel.
- For the parallel loop with `#pragma omp for`, the loop body becomes an OpenCL kernel. The NDRange and the work-group size of the kernel are determined according to the number of iterations of the parallel loop.
- Memory objects shared between OpenMP threads are allocated in the global memory of the compute device.
- Pointers to the memory objects allocated in the device memory are passed as arguments to the associated OpenCL kernel. Thread-private variables or arrays in OpenMP are allocated in the OpenCL private memory.

Figure 2 (a) and (b) illustrates an example of the conversion process. The OpenMP code in Figure 2 (a) is converted to an OpenCL kernel `scan1` in Figure 2 (b). The OpenCL kernel contains the body of the outermost `k` loop (i.e., `j` and `i` loops). The index space (NDRange) and the work-group size of the kernel are determined by the host program. The index space is one dimensional and its size must be greater than or equal to `d3`.

Shared array `x` in Figure 2 (a) is allocated as an OpenCL memory object in the global memory and a pointer to the memory object, `g_x`, is passed to the kernel. The `__global` qualifier indicates that `g_x` is a global memory object. Thread-private variables, `i`, `j`, and `k`, are declared as private variables in the OpenCL kernel.

Algorithmic change. During the conversion process, we do not change algorithms used in all applications but LU. For LU, the OpenMP version exploits pipeline parallelism. However, pipeline parallelism is not suitable for an OpenCL kernel because OpenCL does not provide any synchronization mechanism between work-groups. Instead, we implement the OpenCL version of LU using the hyperplane algorithm [17].

Optimizations in the baseline implementation. Three simple optimization techniques are applied to our baseline implementations in OpenCL. First, if a memory object is shared only between work-items in a work-group, the local memory space is allocated to the memory object. Second, if the inner loop in the OpenMP parallel loop is also parallel, we increase the dimension of the kernel index space. Figure 2 (c) shows an OpenCL kernel `scan2` when this optimization is applied to the kernel in Figure 2 (b). Since `k` and `j` loops in Figure 2 (a) are all parallel (note that the innermost `i` loop has loop-carried dependences), `scan2` has a two-dimensional kernel index space. As a result, the kernel `scan2` has a bigger index space than `scan1` in Figure 2 (b). Finally, we maximize the overlap between host program execution and OpenCL command execution. The commands include kernel execution and memory management (e.g., asynchronous copy) commands.

3.2 Optimizations for CPUs

Although our OpenCL implementation in Section 3.1 can run on both multicore CPUs and GPUs, its performance may not be the best for a specific compute device. We optimize our OpenCL implementation of the NPB for multicore CPUs, and the optimization techniques are as follows:

1. To implement a reduction operation in a work-group, we make a single work-item integrate the data of all other work-items rather than making all work-items participate in the reduction. Although a parallel reduction algorithm [11] can be used, it is executed sequentially because work-items in a work-group are executed sequentially one by one on the corresponding logical CPU core (Section 2.2). Furthermore, since the parallel reduction algorithm typically uses the barrier synchronization mechanism, this incurs the overhead of `setjmp()/longjmp()`. Thus, for the reduction operation in a work-group, sequential reduction by a single work-item achieves better performance than parallel reduction.
2. We reduce the number of work-groups in the kernel index space if it is much larger than the number of CUs (i.e., logical CPU cores). Since a work-group is executed by a single logical CPU core (i.e., an operating system thread), too many work-groups incur significant thread scheduling overhead. We decrease the number of work-groups by decreasing the size of the index space. To decrease the size of the kernel index space, we merge multiple work-items into a single work-item. The kernel of the original index space is enclosed by a loop that iterates over the work-items merged. The loop becomes a new kernel for the index space with decreased size.
3. If the amount of work in a kernel is very small and the number of invocations of the kernel is large, we make the number of work-groups in the index space be the same as the number of logical CPU cores (i.e., CUs) in the compute device and make every work-group have a single work-item. This is a special case of the previous optimization. Since the amount of work in the original kernel is very small, the scheduling overhead and the PE virtualization overhead take a big portion of the kernel execution time.
4. We use the `CL_MEM_USE_HOST_PTR` flag when creating a memory object using `clCreateBuffer()` function. The host pointer, which points to the memory used by the host program, is passed as an argument of `clCreateBuffer()` function when the flag is used. This optimization enables the OpenCL runtime to directly use the memory space pointed to by the host pointer. Consequently, it avoids allocating another memory space for the object in the device memory and copying the contents of the memory object between the host memory and the device memory. Kernels accessing the memory object directly update the host memory with this optimization.

3.3 Optimizations for GPUs

The GPU-specific optimization techniques used in the OpenCL implementation of the NPB are well known. They are the following:

1. We apply memory coalescing technique to kernels by considering the warp scheduling mechanism [24, 26].
2. We try to remove as many branches as possible in a kernel [24,26]. Code divergence due to control flow divergence significantly hurts performance for the SIMT architecture.
3. We try to make the size of the kernel index space and the number of work-groups as large as possible. This can be done by making a nested loop to be a multi-dimensional kernel (an example is shown in Figure 2 (c)). This increases occupancy and improves performance by hiding global memory latency.
4. We try to use the local memory as much as possible for the workspace of a work-group. Accessing the local memory is much faster than accessing the global memory [28].
5. We try to avoid the usage of the global memory and to reduce the number of global memory accesses.
6. We exploit GPU-specific implementations of parallel algorithms, such as parallel reduction [11] or parallel prefix sum (or scan) [12]. For example, Figure 2 (c) can be further optimized using GPU-specific parallel scan algorithm [12].
7. We use atomic functions provided by OpenCL when the usage of the device memory needs to be reduced. In some parallel algorithms, such as parallel bucket sorting used in IS, each work-item is assigned an independent workspace in the memory, saves intermediate data in the workspace, and finally integrates its own data with other work-item's data using reduction. If the size of each work-item's workspace is large, the size of memory space used for all work-items may exceed the size of the device memory. However, if every work-item directly updates a single shared workspace using atomic functions, such as `atom_inc()` and `atom_dec()`, the memory usage will be significantly reduced. Although this optimization

Table 3. OpenCL platform configurations

	OpenCL	Host	Compute device	Clock Freq.	Global Memory	Local Memory
OCL-CPU	AMD-APP-SDK-v2.4	One X5660 logical core	The set of 24 X5660 logical core	2.8GHz	24GB	64KB
OCL-GPU	CUDA 4.0.1	One X5660 logical core	GeForce GTX 480	1.4GHz	1.5GB	48KB

Table 4. Summary of the applications

Application	EP		IS		CG		MG	
Description	Embarrassingly parallel		Integer sort		Conjugate gradient		Multigrid	
Problem Size	Class C: 2^{32}		Class C: 2^{27}		Class C: 150000		Class B: 512x512x512	
OpenCL Platform	OCL-CPU	OCL-GPU	OCL-CPU	OCL-GPU	OCL-CPU	OCL-GPU	OCL-CPU	OCL-GPU
# of Kernels	1	1	5	7	10	10	12	13
# of Kernel Executions	1	1	50	70	7950	7950	1665	2126
# of Barriers	1	3	0	6	0	14	0	6
Global Memory (MB)	384KB	48KB	1056.5	1088.0	445.7	445.8	451.0	452.0
Local Memory (KB)	K1: 1.5	K1: 12	0	K4,5: 2	K1: 16B	K1: 2 K5,9: 0.5 K4,6,7,10: 1	0	K1,2,3: 4 K4: 6.1 K9: 2
Index Space	K1: (65536)	K1: (65536)	K2,3,4: (128)	K3: (2^{27})	K5,9: (24)	K5,9: (9600000)	K1-4: (2,2) ~ (256,256)	K1,2: (8,2) ~ (66048,256)
Optimizations	1, 4	4, 6	2	1, 4, 5, 6, 7	3, 4	1, 4, 6	2, 3, 4	1, 3, 4, 5, 6

Application	FT		SP		BT		LU	
Description	3-D FFT PDE		Pentadiagonal solver		Block tridiagonal solver		LU solver	
Problem Size	Class B: 512x256x256		Class C: 162x162x162		Class B: 102x102x102		Class C: 162x162x162	
OpenCL Platform	OCL-CPU	OCL-GPU	OCL-CPU	OCL-GPU	OCL-CPU	OCL-GPU	OCL-CPU	OCL-GPU
# of Kernels	7	7	14	14	10	19	9	9
# of Kernel Executions	105	105	5600	5600	2000	3800	240502	240502
# of Barriers	1	10	0	0	0	0	1	1
Global Memory (MB)	2306.5	1282.5	1341.7	1341.7	183.2	1155.7	722.6	720.4
Local Memory (KB)	K4: 0.7	K4: 0.5 K5,6,7: 16	0	0	0	0	K1: 0.2	K1: 0.4
Index Space	K3,5: (256,256) K6,7: (512,256)	K5: (2^{13},256) K6,7: (2^{14},256)	K3-5,8,10,12: (192,160) K2: (192,162)	K3-5,8,10,12: (192,160) K2: (192,162,162)	K7-9: (100)	K10,14,18: (128,100) K7,11,15: (128,100,100)	K8,9: (1,1) ~ (192,160)	K8,9: (1,1) ~ (192,160)
Optimizations	1, 4	1, 3, 4, 5, 6	2	2, 3, 5	2	2, 3, 5	1, 2, 4	2, 3, 4, 5, 6

may not improve performance, it enables GPU to execute a kernel with a larger problem size.

4 Evaluation and Characteristics

In this section, we evaluate and characterize the performance of our OpenCL implementation of the NPB (NPB-OpenCL). We implement eight applications in the NPB 3.3.1: EP, IS, CG, MG, FT, SP, BT, and LU according to the method described in Section 3. We compare the performance of NPB-OpenCL with that of the OpenMP version of the NPB (NPB-OpenMP). Specifically, we are interested in the following: speedup, communication-to-computation ratio, host-device computation overlap, scalability, and occupancy.

4.1 Evaluation Methodology

Target machine. A heterogeneous computing system, which consists of two Intel Xeon X5660 CPUs running at 2.8GHz and an NVIDIA GeForce GTX 480 GPU, is used for the evaluation. The Intel Xeon X5660 CPU has 6 cores and supports hyper-threading. Thus, each CPU core provides the operating system with two logical cores. The system has 24GB of main memory and runs Red Hat Enterprise Linux Server 5.5 with the kernel version 2.6.18-194.el5.

Each of our two target OpenCL platforms has a host processor and a compute device. Table 3 shows the two OpenCL platform configurations used in our experiment. The first OpenCL platform (OCL-CPU) is configured by AMD APP SDK 2.4 [3] and uses the entire 24 logical cores as the compute device. It uses one of the logical cores as the host processor. This logical core is shared by the host and the device. The second OpenCL platform (OCL-GPU) is configured by NVIDIA CUDA Toolkit 4.0 [25], and uses a logical CPU core as the host processor and the GPU as the compute device.

The sequential version of the NPB (NPB-SER) and NPB-OpenMP are compiled with gcc 4.1.2. For NPB-OpenCL, the host program is compiled with gcc 4.1.2 and the kernel code is compiled with the device compiler provided by each OpenCL platform. We use the `clGetEventProfilingInfo()` function to measure the execution time and the overhead of each OpenCL command. NVIDIA's Compute Visual Profiler [23] is used to analayze the performance on the GPU platform.

Summary of applications in NPB-OpenCL. Table 4 summarizes the applications in NPB-OpenCL. Device (OCL-CPU or OCL-GPU) specific optimizations are applied to each application in NPB-OpenCL. All applications but MG, FT, and BT are evaluated for the Class C problem size. Since the Class C problem size for MG, FT, and BT is bigger than the global memory size (1.5GB) of the GPU (GeForce GTX 480), they are evaluated for the Class B problem size.

Table 4 also shows the number of barriers used in the kernels. It ranges from 0 to 14. NPB-OpenCL for OCL-CPU has the number of barriers less than or equal to that of NPB-OpenCL for OCL-GPU. Since a work-group barrier in a work-group causes context switching between work-items in the work-group for OCL-CPU, kenels with less number of barriers are more efficient on OCL-CPU.

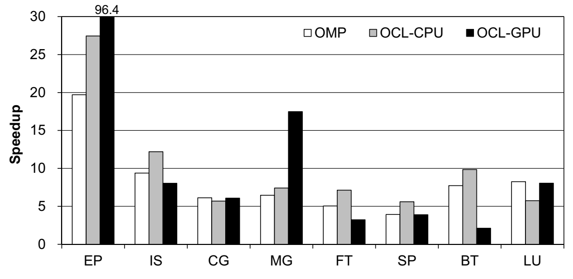
The other synchronization mechanism used in NPB-OpenCL is atomic functions for integers. These functions are optional extensions to OpenCL, but both AMD's and NVIDIA's OpenCL frameworks support them. These functions apply an atomic operation (e.g., atomic increment) to an integer variable allocated in the global memory or local memory. Only IS for OCL-GPU exploits these atomic functions, such as `atom_inc()` and `atom_dec()`.

Table 4 also shows the information of the OpenCL kernels in the applications for each OpenCL platform (OCL-CPU and OCL-GPU). The kernels are numbered according to the order of appearance in the source code and the numbers are prefixed with 'K'. Kernels for initialization are excluded because the original NPB does not include their execution time in the execution time of each application. 'Global Memory' shows the maximum global memory size used by each application. 'Local Memory' shows the size of the local memory used by a work-group in each kernel in kilo-bytes (KB).

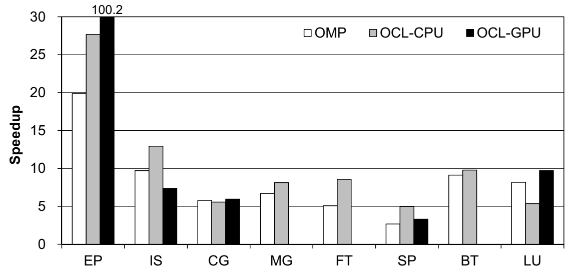
'Index Space' gives the index spaces (NDRanges) used by the most time consuming kernels for each application. The index spaces of most kernels do not change when the kernels are executed multiple times. However, some kernels in MG and LU alter their index space whenever they are executed. For all applications but EP and LU, kernels for OCL-GPU have a bigger index space than those for OCL-CPU. Kernels in SP and BT have 3-dimensional index spaces as well as 2-dimensional index spaces for OCL-GPU while the index spaces for OCL-CPU are only 2-dimensional for SP and 1-dimensional for BT. A bigger index space for OCL-GPU improves performance because the large number of work-items provide the GPU with enough warps to be context switched to hide memory latency [24]. However, the large index space may consume more global memory space. For example, BT on OCL-GPU consumes more global memory space than OCL-CPU. On the other hand, applications suffer from the work-item scheduling overhead on OCL-CPU if the index space is too large.

For IS, MG, and BT, implementations for OCL-GPU have more kernels than those of OCL-CPU. Since a small and simple kernel performs better on the GPU, a complicated kernel that shows good performance on the CPU sometimes needs to be divided into multiple simple kernels on the GPU.

'Optimizations' in Table 4 shows the optimizations described in Section 3 used for each application and for each OpenCL platform.



(a) Class B



(b) Class C

Figure 3. Speedup

4.2 Speedup

Figure 3 shows the speedups of NPB-OpenMP on 24 logical CPU cores (OMP), NPB-OpenCL on OCL-CPU, and NPB-OpenCL on OCL-GPU. The speedups are obtained over NPB-SER on a single logical CPU core. Figure 3 (a) shows the speedups for the Class B problem size and (b) for Class C. For MG, FT, and BT with Class C, OCL-GPU is not shown because GeForce GTX 480 cannot execute them due to the limited global memory size. They can be implemented in another way to overcome the memory constraints of the GPU. However, such an implementation requires a significant effort. We leave this type of OpenCL implementation as our future work. All OpenCL applications achieve performance improvement over NPB-SER, but the speedup is quite different depending on the application.

Even though both of OMP and OCL-CPU exploit data-parallelism and run on the same number of CPU cores, they demonstrate different performance. Interestingly, OCL-CPU gives better performance than OMP for EP, IS, MG, FT, SP, and BT. This performance difference comes from differences in the programming models of OpenCL and OpenMP (e.g., explicit parallelization vs. directive-based parallelization), scheduling mechanism, and compiler's ability to generate efficient code. For CG and LU, the

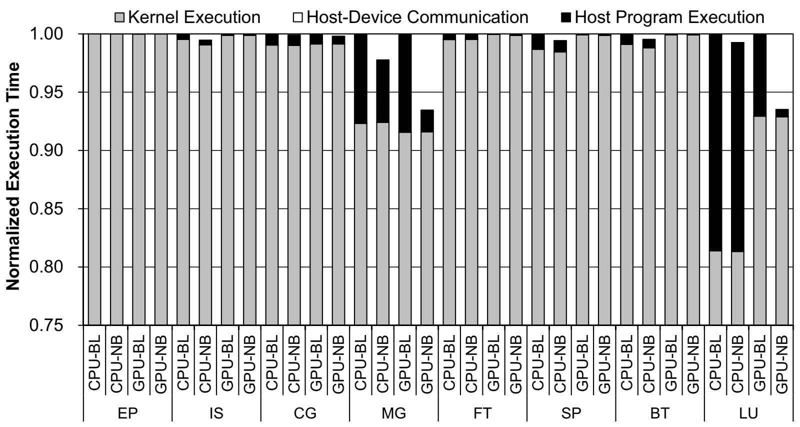


Figure 4. Execution time breakdown. EP, IS, CG, SP, and LU have the Class C problem size. MG, FT, and BT have Class B.

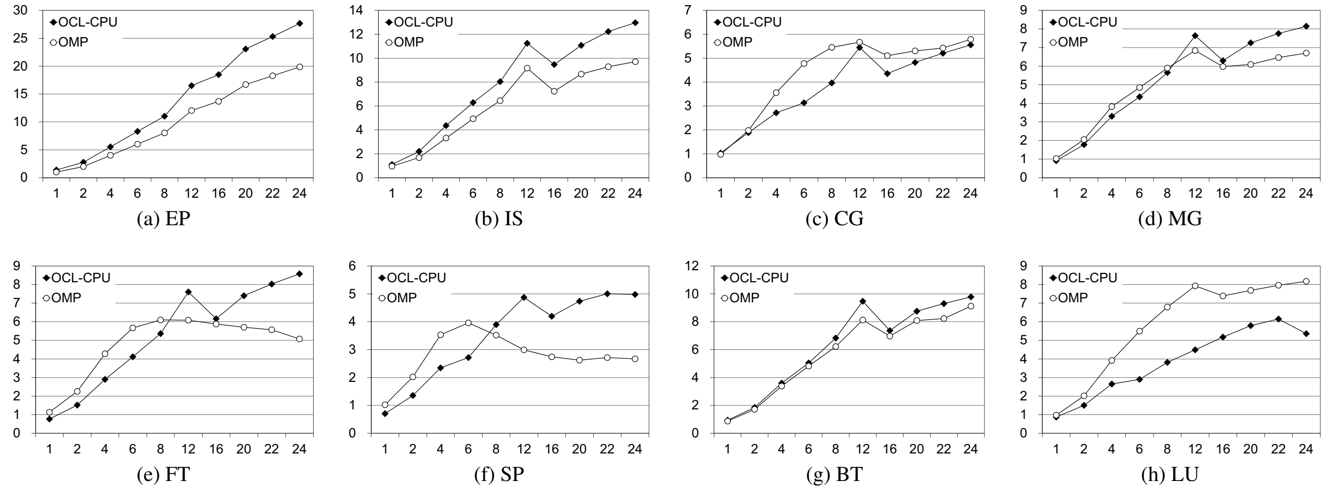


Figure 5. Scalability of NPB-OpenCL on OCL-CPU. The problem size is Class C. X-axis represents the number of logical CPU cores used for the compute device in OCL-CPU and the number of threads used for OMP. Y-axis shows the speedup of OCL-CPU and OMP over NPB-SER.

overhead of OpenCL runtime (this will be described later in Section 4.3) restricts the performance of OCL-CPU, while the overhead is amortized in MG. OMP is slightly better than OCL-CPU for CG and OMP is 1.5 times faster than OCL-CPU for LU.

OCL-GPU is the best for EP, CG (Class C), MG, and LU (Class C). Since these applications contain a significant amount of data parallelism, they are appropriate for running on the GPU to achieve better performance. For EP, CG, and LU, OCL-GPU achieves better performance when the problem size becomes large. However, OCL-GPU does not achieve better performance than OCL-CPU and OMP for other applications. IS suffers from the latency of atomic operations. The performance of FT and BT is limited by their low occupancy (this will be described later in Section 4.5). Control flow divergence [24] is also a performance bottleneck in FT. Since SP has many uncoalesced memory accesses, its performance is limited by the memory bandwidth.

4.3 Execution Time Breakdown

The execution time of an OpenCL application can be divided into three parts: kernel execution, host-device communication, and host execution. Figure 4 shows the execution time breakdown of NPB-OpenCL on OCL-CPU and OCL-GPU. NPB-OpenCL shows a different breakdown of execution time for each application and each device type. Specifically, the host execution time varies from 0% to

18.6% due to the overhead of the OpenCL runtime. The major source of the runtime overhead is frequent launches of kernels (see Table 4).

In Figure 4, CPU(GPU)-BL means that the host processor enqueues a command to the device command-queue and is blocked until the execution of the command finishes. In CPU(GPU)-NB, the host processor progresses without waiting for the completion of the enqueued command until there is a dependence between the result of the command and the host program. The execution time of CPU(GPU)-NB is normalized to that of CPU(GPU)-BL.

Computation and communication. As shown in Figure 4, the host-device communication time is negligible for all applications. This is because NPB-OpenCL contains little memory copies between the host and the device after initialization. In addition, most memory copies are incurred by reduction operations. The sizes of memory objects for the reduction operations are small in NPB-OpenCL. Furthermore, in OCL-CPU, the host memory and the compute device memory actually reside in the same address space. Optimizations done in NPB-OpenCL for OCL-CPU exploit this to reduce redundant memory copy operations, and the memory copy overhead is relatively small because memory copying between the host and device does not go through the PCI-E bus. This is quite different from other CUDA benchmark suites, such as Parboil [32] and Rodinia [6], where some applications spend large portion of their execution time on CPU-GPU communication.

Host-device computation overlap. CPU-NB and GPU-NB in Figure 4 show the execution time of NPB-OpenCL that overlaps the host program execution and the OpenCL command execution while CPU-BL and GPU-BL do not overlap them. The execution times of CPU-NB and GPU-NB are normalized to those of CPU-BL and GPU-BL, respectively. For all applications, the overlapping version reduces the execution time. CPU-NB and GPU-NB decrease the execution time up to 2.3% and 6.5%, respectively. The overlapping execution is more efficient when the portion of the host program execution is large.

4.4 Scalability

We measure the scalability of NPB-OpenCL by adjusting the number of CUs used in the OpenCL compute device and compare it with that of NPB-OpenMP. AMD's OpenCL framework supports setting the number of available CPU cores (i.e., CUs) by specifying the environment variable CPU_MAX_COMPUTE_UNITS, but this mechanism is not available for GPUs. For this reason, we conduct the scalability experiment only on OCL-CPU. The number of threads (i.e., logical CPU cores) used in NPB-OpenMP is determined by the environment variable OMP_NUM_THREADS.

Figure 5 shows the experimental result. It shows the speedup of NPB-OpenCL and NPB-OpenMP over NPB-SER for the same number of logical CPU cores. The reason for the drop at 12 logical CPU cores is that the multicore CPU (Intel Xeon X5660) supports only 12 physical cores. The result demonstrates that the performance of OCL-CPU is better than or comparable to that of OMP except CG and LU. This is mainly because NPB-OpenCL exploits parallelism in much finer-granularity than that of NPB-OpenMP. Note that we increase the size of the kernel index space if the inner loop in the OpenMP parallel loop is also parallel (Section 3.1). This increases the number of work-groups to be scheduled and decreases the amount of work to be done by each work-item. Consequently, the OpenCL runtime scheduler can achieve better load-balancing between CUs (i.e., logical CPU cores) in the compute device.

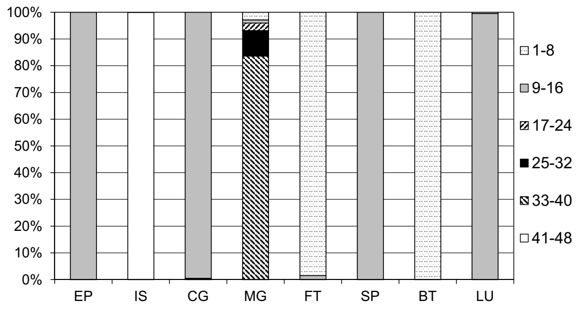


Figure 6. Occupancy on OCL-GPU. Class B is used for MG, FT, and BT. Class C is used for EP, IS, CG, SP, and LU.

Kernels in CG are very simple and are frequently launched. These make the OpenCL runtime overhead relatively large. In addition, the most important kernel in CG contains irregular memory access patterns (e.g., subscript of subscript references). For LU, the OpenCL version with the hyperplane algorithm introduces frequent kernel launching. This increases the host program execution time as shown in Figure 4 and makes the OpenCL runtime scheduler busy. For MG, FT, and SP, although NPB-OpenCL is slower than NPB-OpenMP with the smaller number of logical cores due to the runtime overhead, it achieves better speedup with a large number of cores thanks to well-balanced workload across CUs. The OpenCL runtime scheduling overhead is amortized by the resulting good load balancing. Both NPB-OpenCL and NPB-OpenMP achieve a linear speedup for EP. EP is a compute-bound application with little memory accesses.

4.5 Occupancy

Occupancy for a GPU is defined as the ratio of the number of active warps (i.e., CU) per streaming multiprocessor to the maximum number of possible active warps [24]. We obtain the occupancy of each kernel in NPB-OpenCL

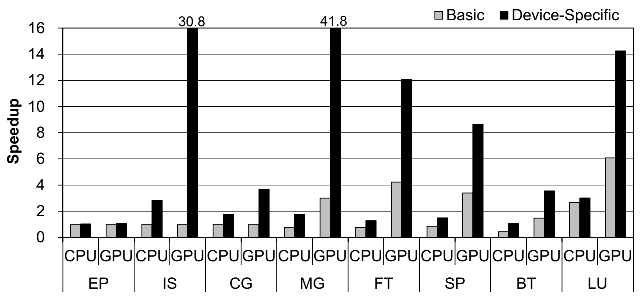


Figure 7. The effect of the optimizations. Class C is used for EP, IS, CG, SP, and LU. Class B is used for MG and BT. For FT, Class B is used for OCL-CPU **and Class A for** OCL-GPU **due to the large device memory requirement of the baseline implementation.**

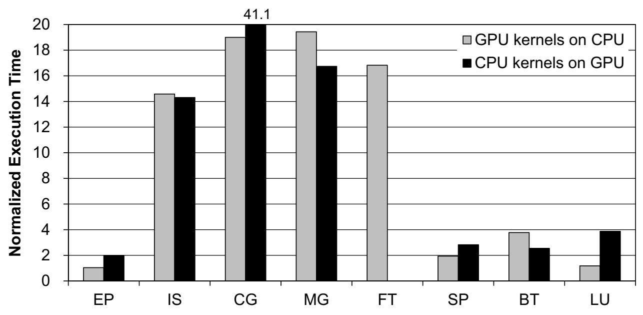


Figure 8. Performance portability. Class B is used for MG, FT, and BT. Class C is used for EP, IS, CG, SP, and LU.

using NVIDIA's Compute Visual Profiler [23] and calculate the number of active warps. For GeForce GTX 480, the maximum number of possible active warps per streaming multiprocessor is 48. Although higher occupancy does not guarantee higher performance, occupancy indicates how well the GPU is utilized.

Figure 6 shows the breakdown of the execution time for different number of active warps per streaming multiprocessor. It shows that most applications have a fixed occupancy during their execution. For EP and SP, their occupancies are small and fixed because their kernels use many registers. The occupancy of CG is relatively low. If we try to increase its occupancy by modifying the index spaces of its kernels, performance is degraded due to the increased control flow divergence. IS has a very large occupancy because its kernels use only a small number of registers and they have large index spaces. MG has many different occupancies because its kernels have different global index spaces. For most of the execution time, the occupancy of FT is low because the size of the local memory used in its kernels limits the occupancy. When we try to increase the occupancy of FT by modifying the index spaces of its kernels, its performance is degraded. For BT and LU, their kernel index spaces are relatively small. This restricts their occupancy.

5 Discussions

In this section, we discuss the effectiveness of device-specific optimizations used in NPB-OpenCL and the performance portability issue.

5.1 Device-specific Optimizations

Figure 7 shows the effectiveness of the device-specific optimizations described in Section 3.2 and Section 3.3. The speedup is obtained over the baseline implementation described in Section 3.1 without any optimization.

When we apply the basic optimizations (Basic) described in Section 3.1, this improves the performance of only one application, LU, on OCL-CPU and five applications, MG, FT, SP, BT, and LU, on OCL-GPU. On OCL-CPU, the basic optimizations degrade the performance of MG, FT, SP, and BT because making the kernel index space bigger incurs serious work-item scheduling overhead.

Device-specific optimizations (Device-specific) boost performance for all applications. On average (geometric mean), the speedups of CPU-specific and GPU-specific optimizations are 1.65 and 8.47, respectively. The results indicate that manual optimizations for different compute devices in OpenCL are inevitable for better performance because the current OpenCL framework does not automatically optimize the kernel code for a specific compute device. We suggest that future OpenCL frameworks should consider device-specific optimizations or auto-tuning techniques [33] for better performance.

5.2 Performance Portability

Figure 8 shows the execution times when executing the GPU-optimized version on OCL-CPU (GPU kernels on CPU, normalized to the execution time when the CPU-optimized version is executed on OCL-CPU) and when running the CPU-optimized version on OCL-GPU (CPU kernels on GPU, normalized to the execution time when the GPU-optimized version is executed on OCL-GPU). For FT, the CPU-optimized version requires 2.3GB of the global memory space (Table 4) that is bigger than the size of the GPU device memory (1.5GB). Thus, we cannot execute the CPU-optimized version on OCL-GPU.

The result shows that the execution time is significantly increased (up to 41.1 times) for most applications. This implies that we need to rewrite or re-optimize an OpenCL application that is optimized for a different compute device. This may impair OpenCL's big advantage, portability by standardization. Future research on OpenCL needs to keep the performance portability issues in mind.

The results in Section 4 indicate that OpenCL is promising for manycore architectures, especially in high performance computing domains. Although we show the effectiveness of NPB-OpenCL on a single compute device, utilizing both multicore CPUs and GPUs can be done with OpenCL, and this will synergistically increase the performance.

6 Related Work

The NAS Parallel Benchmarks (NPB) has been studied by many researchers. Shan *et al.* [29] use the NPB to compare three programming models, MPI, OpenMP, and UPC (Unified Parallel C) [1]. Wong *et al.* [34] present the architectural requirements and scalability of the MPI version of the NPB (NPB-MPI). Jin *et al.* [17] describe the OpenMP implementation of the NPB (NPB-OpenMP) and compare the performance of NPB-OpenMP to that of NPB-MPI. They demonstrate that OpenMP can achieve good performance for a shared memory multiprocessor.

Hybrid implementations of the NPB have also been examined. Cappello and Etiemble [5] compare the performance of NPB-MPI to that of the MPI+OpenMP version on the IBM SP machine. Jin and Wijngaart [16] show the performance characteristics of the multi-zone version [9] of the NPB. Since the multi-zone version uses different problem sizes from NPB-MPI, Wu and Taylor [36] implement hybrid MPI+OpenMP versions of SP and BT based on NPB-MPI. Pennycook *et al.* [27] describe the MPI+CUDA implementation of LU. Evaluations of hybrid implementations commonly imply that a unified MPI approach achieves better performance than the hybrid version if the code for a single node does not contain sufficient loop-level parallelism. However, if the communication overhead is significant or the computation for a single node is well-parallelized with OpenMP, the hybrid version gives better performance.

In this paper, we present the OpenCL implementation of NPB (NPB-OpenCL) and study its performance characteristics. We compare the performance of NPB-OpenCL to that of NPB-OpenMP. The performance of NPB-OpenCL is evaluated on a heterogeneous parallel platform consisting of multicore CPUs and a GPU. Although some applications of the NPB were implemented in OpenCL [19] previously, ours is a complete implementation of the entire NPB in OpenCL and is the first evaluation of all the NPB applications for the heterogeneous computing system.

Recent benchmark suites, such as Rodinia [6, 7] and SHOC [8], target heterogeneous parallel computing systems while previous benchmark suites focus on either CPUs (e.g, SPEC CPU [30], PARSEC [4], and SPLASH-2 [35]) or GPUs (e.g., Parboil [32]). Rodinia contains nine applications written in both OpenMP and CUDA. While Rodinia and Parboil can be executed only on NVIDIA GPUs, NPB-OpenCL runs on every processor that supports OpenCL.

Lee *et al.* [21] show that the performance gap between CPUs and GPUs is largely affected by two factors: the types of CPUs and GPUs, and optimizations applied to the code. They improve the performance of CPU code by applying CPU-specific optimizations and GPU code by applying GPU-specific optimizations. We also apply device-specific optimizations to NPB-OpenCL to improve performance, but our goal is characterizing performance rather than identifying the performance gap between CPUs and GPUs.

7 Conclusions and Future Work

In this paper, we present the OpenCL implementation (NPB-OpenCL) of the NAS Parallel Benchmarks (NPB) and characterize its performance on multicore CPUs and a GPU. We also compare NPB-OpenCL to the OpenMP version of NPB (NPB-OpenMP) in terms of performance and scalability. Experimental results show that performance characteristics of NPB-OpenCL are different from those of NPB-OpenMP and depend on the type of the OpenCL compute device. The results indicate that optimizations for different OpenCL compute devices are inevitable for better performance. This may impair OpenCL's big advantage, portability by standardization. Thus, future OpenCL frameworks should consider device-specific optimizations or auto-tuning techniques for better performance. Future research on OpenCL needs to keep performance portability issues in mind. NPB-OpenCL supplements the current NPB suite and gives insights to OpenCL performance for conventional scientific applications.

References

[1] Berkely UPC – unified parallel c. `http://upc.lbl.gov`.

[2] AMD. The AMD fusion family of APUs. `http://sites.amd.com/us/fusion/apu/Pages/fusion.aspx`.

[3] AMD. OpenCL: The open standard for parallel programming of GPUs and multi-core CPUs. `http://www.amd.com/streamopencl`.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *PACT '08: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81, Oct. 2008.

[5] F. Cappello and D. Etiemble. MPI versus MPI+OpenMP on IBM SP for the NAS benchmarks. In *SC '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, Nov. 2000.

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC '09: Proceedings of the 2009 International Symposium on Workload Characterization*, pages 44–54, Oct. 2009.

[7] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron. A characterization of the Rodinia bench-

mark suite with comparison to contemporary CMP workloads. In *IISWC '10: Proceedings of the 2010 IEEE International Symposium on Workload Characterization*, Dec. 2010.

[8] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter. The scalable heterogeneous computing (shoc) benchmark suite. In *GPGPU '10: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 63–74, Mar. 2010.

[9] R. F. V. der Wijngaart and H. Jin. NAS parallel benchmarks, multi-zone versions. Technical Report NAS-03-010, NASA Advanced Supercomputing (NAS) Division, Jul. 2003.

[10] 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 *PACT '10: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 205–216, Sep. 2010.

[11] M. Harris. Optimizing parallel reduction in CUDA. `http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf`.

[12] M. Harris. Parallel prefix sum with CUDA. `http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/scan/doc/scan.pdf`.

[13] IBM. OpenCL development kit for linux on power. `http://www.alphaworks.ibm.com/tech/opencl`.

[14] Intel. Intel microarchitecture codename sandy bridge. `http://www.intel.com/technology/architecture-silicon/2ndgen/index.htm`.

[15] Intel. Intel OpenCL SDK. `http://software.intel.com/en-us/articles/intel-opencl-sdk/`.

[16] H. Jin and R. F. V. der Wijngaart. Performance characteristics of the multi-zone NAS parallel benchmarks. In *IPDPS '04: Proceedings of the 18th International Parallel and Distributed Processing Symposium*, Apr. 2004.

[17] H. Jin, M. Frumkin, and J. Yan. The OpenMP implementation of NAS parallel benchmarks and its performance. Technical Report NAS-99-011, NASA Advanced Supercomputing Division, Oct. 1999.

[18] Khronos OpenCL Working Group. The OpenCL specification version 1.0. `http://www.khronos.org/opencl`.

[19] A. Kulchitsky, G. Newby, T. Li, M. Malik, M. Sharif, R. Shahid, W. Dang, and B. Marken. Arctic region supercomputer center work related to gpgpus and ibm cell. `http://saahpc.ncsa.illinois.edu/10/presentations/day2/session1/presentation_Kulchitsky.pdf`.

[20] 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 *PACT '10: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, pages 193–204, Sep. 2010.

[21] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, R. Singhal, and P. Dubey. Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *ISCA '10: Proceedings of the 37th Annual International Symposium on Computer Architecture*, pages 451–460, Jun. 2010.

[22] NAS Division. NAS parallel benchmarks. `http://www.nas.nasa.gov/Resources/Software/npb.html`.

[23] NVIDIA. Compute visual profiler user guilde. `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/Compute_Visual_Profiler_User_Guide.pdf`.

[24] NVIDIA. OpenCL best practices guide. `http://developer.download.nvidia.com/compute/DevZone/docs/html/OpenCL/doc/OpenCL_Best_Practices_Guide.pdf`.

[25] NVIDIA. OpenCL for NVIDIA. `http://developer.nvidia.com/opencl`.

[26] NVIDIA. OpenCL programming guide for the CUDA architecture. `http://developer.download.nvidia.com/compute/DevZone/docs/html/OpenCL/doc/OpenCL_Programming_Guide.pdf`.

[27] S. J. Pennycook, S. D. Hammond, S. A. Jarvis, and G. R. Mudalige. Performance analysis of a hybrid MPI/CUDA implementation of the NAS-LU benchmark. In *PMBS '10: Proceedings of the 1st International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems*, pages 23–29, Nov. 2010.

[28] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 73–82, Feb. 2008.

[29] H. Shan, F. Blagojević, S.-J. Min, P. Hargrove, H. Jin, K. Fuerlinger, A. Koniges, and N. J. Wright. A programming model performance study using the NAS parallel benchmarks. *Scientific Programming*, 18(3-4):153–167, Aug. 2010.

[30] Standard Performance Evaluation Corporation. SPEC CPU benchmark suite. `http://www.spec.org/cpu/`.

[31] J. A. Stratton, V. Grover, J. Marathe, B. Aarts, M. Murphy, Z. Hu, and W.-m. W. Hwu. Efficient compilation of fine-grained SPMD-threaded programs for multicore CPUs. In *CGO '10: Proceedings of the 8th annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 111–119, Apr. 2010.

[32] The IMPACT Research Group. Parboil benchmark suite. `http://impact.crhc.illinois.edu/parboil.php`.

[33] S. W. Williams. *Auto-tuning Performance on Multicore Computers*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008.

[34] F. C. Wong, R. P. Martin, R. H. Arpaci-Dusseau, and D. E. Culler. Architectural requirements and scalability of the NAS parallel benchmarks. In *SC '99: Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, Nov. 1999.

[35] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *ISCA '95: Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, Jun. 1995.

[36] X. Wu and V. Taylor. Performance characteristics of hybrid MPI/OpenMP implementations of NAS parallel benchmarks SP and BT on large-scale multicore clusters. In *PMBS '10: Proceedings of the 1st International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems*, pages 56–62, Nov. 2010.