J Supercomput
DOI 10.1007/s11227-014-1112-2

An OpenCL micro-benchmark suite for GPUs and CPUs

Xin Yan · Xiaohua Shi · Lina Wang · Haiyan Yang

Abstract Open computing language (OpenCL) is a new industry standard for task-parallel and data-parallel heterogeneous computing on a variety of modern CPUs, GPUs, DSPs, and other microprocessor designs. OpenCL is vendor independent and hence not specialized for any particular compute device. To develop efficient OpenCL applications for the particular platform, we still need a more profound understanding of architecture features on the OpenCL model and computing devices. For this purpose, we design and implement an OpenCL micro-benchmark suite for GPUs and CPUs. In this paper, we introduce the implementations of our OpenCL micro benchmarks, and present the measuring results of hardware and software features like performance of mathematical operations, bus bandwidths, memory architectures, branch synchronizations and scalability, etc., on two multi-core CPUs, i.e. AMD Athlon II X2 250 and Intel Pentium Dual-Core E5400, and two different GPUs, i.e. NVIDIA GeForce GTX 460se and AMD Radeon HD 6850. We also compared the measuring results with existing benchmarks to demonstrate the reasonableness and correctness of our benchmark suite.

Keywords Micro benchmark · OpenCL · GPU · Multi-core CPU

X. Yan · X. Shi (✉) · L. Wang · H. Yang
State Key Laboratory of Software Development Environment,
School of Computer Science and Engineering, Beihang University, Beijing, China
e-mail: xhshi@buaa.edu.cn

X. Yan
e-mail: giraffe410@126.com

L. Wang
e-mail: binglina.wang@163.com

H. Yang
e-mail: yhy@buaa.edu.cn

Published online: 28 January 2014

1 Introduction

Using the Open Computing Language (OpenCL) standard, we can write applications that access many available programming resources: CPUs, GPUs, and other processors such as DSPs and Cell processors [1]. OpenCL includes a language based on C99 for writing kernel functions that run on OpenCL devices, plus application programming interfaces (APIs) that are used to define and then control platforms.

The compute unified device architecture (CUDA) and ATI Stream platforms, released by NVIDIA and AMD, respectively, also provide their programming frameworks that allow a programmer to develop a GPU computing application without tricky graphics programming techniques. However, each programming framework only runs on some specified processors. For instance, CUDA programs can only run on NVIDIA's GPUs. Using OpenCL, a programmer can utilize different compute devices in a unified way. Thus, it can enable a programmer to avoid writing a vendor-specific code, resulting in improved code portability. Although OpenCL provides source-code portability and correctness of kernels across a variety of hardware, it does not guarantee that a particular kernel will achieve best performance on different architectures. The nature of the underlying hardware may make some programming strategies more appropriate for particular platforms than for others. For this purpose, we design an OpenCL micro-benchmark suite for GPUs and CPUs.

Micro benchmarks have been widely used in the past to measure hardware architectures of different processor structures. Sangmin Seo et al. [2] characterized the performance of an OpenCL implementation of the NAS parallel benchmark suite (NPB) on a general-purpose GPU. Volkov and Demmel [3] benchmarked NVIDIA's GPUs, with a focus on performance for linear algebra using CUDA. Parboil [4] and Che [5] proposed GPU benchmarks which are written in CUDA. Both of them are composed of scientific applications and micro benchmarks to understand the behaviour of scientific applications on GPUs. uBench [6] is a suite of micro benchmarks to explore the impact on performance of the thread-block geometry choice criteria and GPU hardware resources and configurations. So far, most previous studies focused on the performance of GPUs and written in CUDA, while the micro-benchmark suite we designed is based on OpenCL and not just specific for GPUs, but also for multi-core CPUs. Shen et al. [7] compared the performance of OpenCL and OpenMP. They selected three applications, which provide equivalent OpenMP and OpenCL implementations and carry out experiments with different datasets on three multi-core platforms. SHOC [8] is a benchmark suite that tests the performance and stability of scalable heterogeneous computing systems. While SHOC uses micro benchmarks to assess architectural features, it is primarily targeted at measuring the more high-level and system-wide performance. For instance, it does not try to identify the individual characteristics of mathematical operations or measure the bandwidth changes brought by different access pattern parameters. Conversely, our suite is aimed at determining useful low-level characteristics of devices and includes mathematical operations, build-in functions, branch synchronization, scalability, as well as the bandwidths of the global and local memories with different accessing modes.

The contributions of this paper are the following:

- We designed and implemented a series of OpenCL micro benchmarks, including mathematical operation test, bus bandwidth test, memory architecture test, branch synchronization test, and scalability test, etc., with more details of platform features compared with others.
- We evaluated the OpenCL micro-benchmark suite on two multi-core CPUs (AMD Athlon II X2 250 and Intel Pentium Dual-Core E5400) and two different GPUs (NVIDIA GeForce GTX 460se and AMD Radeon HD 6850). We analysed the measuring results on different computing devices, and evaluated the performance characteristics according to their hardware architectures. We also compared our measuring results with some existing benchmarks to demonstrate the reasonableness and correctness of our benchmark suite.

The rest of this paper is organized as follows. Section 2 introduces the OpenCL implementations of the micro-benchmark suit. Section 3 presents and discusses the performance evaluation of our micro benchmarks on multi-core CPUs and GPUs. Section 4 concludes this paper.

2 Implementations of OpenCL micro-benchmark suite

In OpenCL, one host (such as a CPU) controls one or more compute devices (such as CPUs and GPUs). OpenCL defines abstractions for device memory structures and provides a set of APIs that allocate memory and control data transfer. The device memory is divided into four distinct memory regions, including global memory, local memory, constant memory and private memory. In OpenCL, the global memory is a memory region accessible to all compute units of the compute device. Before a computation commences, the necessary data should be transmitted from host to device, where they are reachable for compute kernels. A compute unit contains process elements inside the compute device chip itself. Each compute unit could access to a local memory region, which is shared among all of its work items in one work group. This memory region could be an order of magnitude faster than the global memory, as it usually resides on-chip. According to the OpenCL architecture and its memory models and execution models, we design and implement a series of specific kernel functions which measure the performance characteristic of both GPUs and multi-core CPUs from different aspects, including mathematics, bus bandwidth, memory systems, branch penalty, thread hierarchies, etc.

Since the original intention of OpenCL is an open API suite for developing general purpose GPU computing software, most of OpenCL applications select GPUs as OpenCL devices, while neglecting its support for other computing devices, such as multi-core CPUs. However, the hardware architectures of GPUs and multi-core CPUs are both taken into account during the design of our micro-benchmark suite.

In our micro-benchmarks, the number of work item and the size of work group are two adjustable parameters. The default number of work items is 32768 and the default size of work group is 32 for all GPU and CPU processors unless there is a specific explanation in a benchmark description. We use a 1-dimensional arrangement in the data so the work_dim is always set to 1, except the micro benchmark of image object bandwidth testing. In each micro-benchmark kernel, we repeat instructions or operations that exactly we want to measure a number of times to reduce the influence of other non-related operations on measurement results. In actual measurement, we increase the repetitions until the performance of each kernel achieves a relatively stable state. For instance, in the kernel which is going to measure the performance of add instruction, the add instruction will loop 20,480 times. In kernel functions, after each thread has finished reading the data, it will do a simple calculation using the data to avoid compiler optimizations that could eliminate operations whose results are not used later.

2.1 Mathematics

The micro-benchmark suite measures the Floating Point Operations Per Second (FLOPS) performance for basic arithmetic instructions. Because different OpenCL runtimes may have different implementations of mathematical built-in functions, we measure the invoked times per second for each built-in function instead of FLOPS. Table 1 lists all of the arithmetic instructions and built-in functions supported by our micro benchmarks.

The micro-benchmark suite measures both single-precision and double-precision floating point calculations. We have also considered the vector types provided by OpenCL, like float2, float4 and float8, etc. The vector here means a data structure that contains multiple elements of the same data type. During a vector operation, each element (called a component) is operated upon in the same clock cycle. Many modern processors are capable of processing vectors, but ANSI C/C++ does not define any basic vector data types. With OpenCL, we can write vector routines once and run them on any compliant processor. We use different kernels to measure different arithmetic operations or built-in functions. Each kernel contains a large number of the same kind of arithmetic instruction or built-in function. For example, the sample code of testing add instructions is listed in Fig. 1, in which the variable *N* represents times the loop

Table 1 Arithmetic instructions and built-in functions in micro benchmark

Micro-benchmark	Arithmetic instructions and built-in functions
Benchmark-1	add/sub/mul/div/mad
Benchmark-2	fmax/fmin
Benchmark-3	sin/asin/cos/acos/tan/atan
Benchmark-4	exp/log/rsqrt/fract/frexp/hypot/ldexp/exp10/ilogb/log10
Benchmark-5	sinh/cosh/tanh/asinh/acosh/atanh/asinpi/acospi/atanpi
Benchmark-6	cbrt/ceil/copysign/fabs/fdim/floor/fma/fmod
Benchmark-7	atan2/atan2pi/sinpi/cospi/tanpi/sincos
Benchmark-8	log1p/logb/modf/lgamma/remainder/rint/round/trunc
Benchmark-9	erfc/erf/expm1/nextafter/pown/powr/remquo/rootn/tgamma

Fig. 1 Sample code of kernel testing add instruction

```
float t1 = input_arg1;
float t2 = input_arg2;
for(i = 0; i < N; i++) {
    repeat512(t1 += t2;)
}
*result_add = t1;
```

and the macro *repeat*512 means the instruction in the loop will repeat 512 times again. We changed the repeat times from 64 to 4,096, even higher or less, to evaluate the influence of loop instructions and other noises on arithmetic instructions and build-in functions, and found that 512 is one of the best cases for most arithmetic operations and built-in functions, with the trade-off in terms of testing precision, code length and execution time, etc.

In our micro benchmarks, kernel functions are executed with thread numbers that increase from 1,024 to 524,288 and work-group sizes that increase from 32 to 256 by a factor of 2, then record the best performance.

2.2 Bus bandwidth

OpenCL program needs to transfer data between host processor and compute devices through the interconnection bus (typically PCI-E bus for GPUs). The bus bandwidth can become a performance bottleneck if there are large amounts of data transferred between the host processor and OpenCL devices. The bus bandwidth micro benchmark measures the bus bandwidth by repeatedly transferring data buffers both internally and to/from the OpenCL devices. The buffer sizes increase from 4 to 64 MB with an adjustable increment which default is 4 MB. We use two patterns, including direct copy (like clEnqueueReadBuffer) and memory-mapped copy (like clEnqueueMapBuffer) to transfer data between a host and a device and the copy operations are blocking. Although PCI-E communication is encapsulated in packets, we do not manipulate its low-level settings like packet sizes, but use the default OpenCL API configurations.

2.3 Device memory bandwidth

Each memory region in OpenCL has its own features for reading and writing. Memory objects of OpenCL are categorized into two types: buffer objects and image objects. A buffer object stores a one-dimensional collection of elements whereas an image object is used to store a two- or three-dimensional texture, frame-buffer or image [9]. The micro benchmark evaluates the bandwidth of each memory region by measuring the execution time of kernels which include a series of read/write operations on different memory regions.

2.3.1 Global memory bandwidth

The global memory region permits read/write access to all work items in all work groups [9]. It is one of the major ways to achieve data exchange among threads using

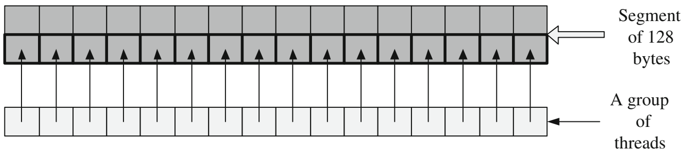


Fig. 2 Example of coalesced access pattern. All threads access the corresponding memory addresses within a segment

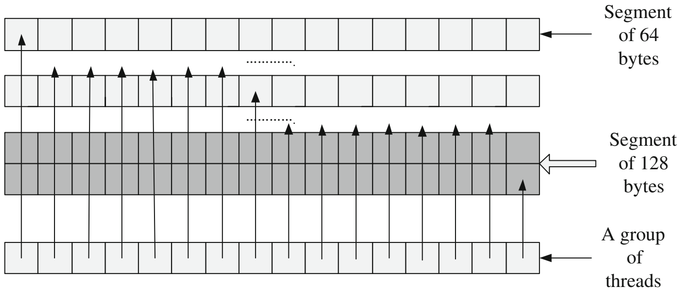


Fig. 3 Example of non-coalesced access pattern. Different threads access different segments

the global memory region. Currently, the global memory micro-benchmark primarily focused on the coalesced and non-coalesced, as well as the impact of cache. For GPU architectures, one of the major factors that affect the global memory bandwidth is whether or not accessing the global memory in a coalesced mode. Figures 2 and 3 show examples of coalesced and non-coalesced global memory access patterns.

When accessing a portion of global memory, all threads in a group (NVIDIA calls it $warp$, and ATI calls it $wavefront$) should access a segment of memory at the same time to take full advantage of the bus bandwidth. The global memory accesses by all threads within a group could be coalesced into a single access if meeting certain requirements. For example, for devices like NVIDIA GTX 460, the memory elements accessed by a group of threads should fall into a 128 byte segment of memory. The address of the first element should be aligned to 16 times the element's size.

The memory transaction segment size could be variable (32, 64, or 128 bytes). Depending on the amount of memory needed and the memory pattern access (scattering or contiguous data), the segment size could be automatically selected to avoid wasting bandwidth. For Fermi architecture (such as NVIDIA GTX460), the memory transaction segment sizes are determined as follows: When L1 cache memory is enabled, the hardware always issues segment transactions of 128 bytes, the cache-line size; otherwise, 32 bytes segment transactions are issued [10]. In our micro-benchmarks, we consider the situation of L1 cache enabled by default, and the memory transaction segment size is fixed as 128 bytes.

We measure the global memory bandwidth by accessing the global memory in a certain addressing pattern. For each loop in the kernel of the micro benchmark, there are 16 memory access operations. The access addresses will be calculated according to Formula 1, in which n is a value initialized to 0 and increments with a step of 32,768. Thus whether the memory access segments are aligned only depends on the

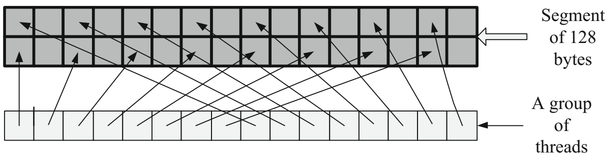
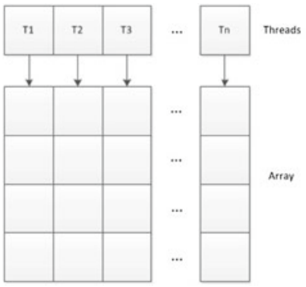


Fig. 4 Coalesced access pattern in which the accessing addresses fall in two segments

Fig. 5 Sample of column major form to access the array



factor s. The variable $size$ in Formula 1 means how many words we want to access on the global memory.

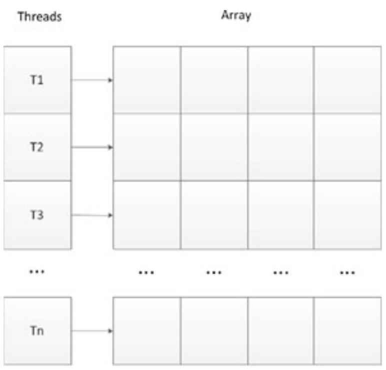
$$\text{address} = (s + n)\&(\text{size} - 1) \tag{1}$$

In Formula 1, $s = gid{*}\,stride$, in which gid is the global work-item ID and $stride$ increments from 0 to 16 by a step of 1. It means that the access address interval of threads within a warp gradually expands from 0 to 16. In this way, we can significantly measure the performance difference between coalesced and non-coalesced accesses to the global memory.

For instance, Figs. 2 and 4 show the specific access patterns for each thread in the kernel when $stride$ equals 1 and 2, respectively. As shown in Fig. 2, if all accessing addresses of 16 threads fall in a segment of 128 bytes, 16 sequential positions on global memory (one position per thread) will be accessed at the same time. However, if the accessing addresses of these threads fall in two segments as shown in Fig. 4, the global memory accesses will be coalesced into two independent accesses.

Considering CPU architectures, we also implement a micro-benchmark which measures the global memory bandwidth when accessing the global memory in row major form and column major form, respectively. In this micro-benchmark, we use an array with a fixed size of 1,024 × 1,024 bytes in global memory. 1,024 × 1,024/n (n increments from 1 to 512 as a multiple growth of geometric sequence by 2) threads are used to read and write this array, either in row major form or column major form. The variable n also represents the number of bytes in the array each thread has to access. Figures 5 and 6 show samples of column major and row major form to access the array. When accessing in column major form, the interval between two data accessed by one thread is relatively large, resulting in cache missing. However, each thread within a work group accesses the consecutive elements, resulting in coalesced access on GPUs. When accessing in row major form, the situation is reversed. For CPUs, all

Fig. 6 Sample of row major form to access the array



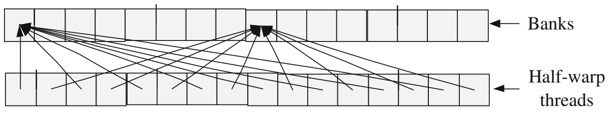


Fig. 7 Example of 8-way bank conflicts in which every 8 threads of a half-warp access the same bank

threads within a work group could be scheduled on a single multiprocessor core by the OpenCL runtime we used. To make full use of CPU cores, when the number of threads is less than 256, the work-group size is set to half the number of threads, otherwise 256. While for GPUs, because there are lots of compute units in each process element, the work-group size is set to the number of threads until the thread number grows to 256, after that it is fixed as 256.

So far, the global memory micro benchmark primarily focuses on the coalesced and non-coalesced models on GPUs, as well as the impact of caches on CPUs.

2.3.2 Local memory bandwidth

The local memory can be used to allocate variables that are shared by all work items in the same work group. It could be assigned to dedicated memory regions on the OpenCL device. Alternatively, the local memory region may be mapped to some sections of the global memory. To achieve high memory bandwidth for concurrent accesses, the local memory is divided into equal-size memory modules, called *banks*, which can be accessed simultaneously by threads within a group. However, if two or more addresses accessed by threads within a half-group or half-warp fall in the same memory bank like Fig. 7, it will result in a bank conflict and the accesses have to be serialized. The micro benchmark evaluates the impact of bank conflicts of local memory. For instance, there are 16 banks for NVIDIA GTX460 and 32 banks for AMD HD6850, which are interleaved with a width of 32-bit. For NVIDIA GTX460, byte 0–3 are in bank 1, 4–7 in bank 2,. . ., 64–69 in bank 1 and so on. The addressing patterns in this kernel are similar to Formula 1 of global memory bandwidth micro benchmark. The size parameter used in Formula 1 here means the local memory size for units in the device.

2.3.3 Constant memory bandwidth

Constant memory is a region of global memory that remains constant during the execution of a kernel [9]. Memory objects are allocated and initialized into constant memory by the host processor. Therefore, only the read bandwidth of constant memory is taken into consideration in our micro benchmark. In this micro benchmark, we measure the bandwidth of constant memory by reading an element in an array repeatedly. We initialize an array in constant memory according to Formula 2, in which $stride$ grows from 1 to 128. In the kernel, we take the global work-item ID as the first index to read the array, and then use the value read in the array as the next index to visit the array repeatedly.

$$\text{array}_i = (i + \text{stride})\%\text{arraySize} \tag{2}$$

2.3.4 Image object bandwidth

In this micro benchmark, we measure the read and write bandwidths of 2-dimensional image objects by repeatedly calling the built-in function $read\_imagef()$ or $write\_imagef()$ in the kernel. In kernels, each thread calls the $read\_imagef()$ to use the coordinate (x, y) to do an element lookup in the 2D image object we used. The coordinate (x, y) is determined by the two-dimensional global work-item ID of each thread. Unlike CUDA, OpenCL 1.0 and 1.1 only support 2D/3D image objects excluding 1D. For a 2D image object, we change the size of the image object and adjust its height and width, while observing the differences of the bandwidth.

2.4 Branch

In general, an OpenCL device is divided into one or more compute units (CU) that are further divided into many processing elements (PE). GPUs execute OpenCL applications in a single-instruction multi-threading (SIMT) model that could also be regarded as an implicit single-instruction multi-data (SIMD) model. For SIMT architectures, one instruction is fetched for a warp of sequential threads and executed on the SIMD units, thus diverging control flow with thread-dependent conditional branches in the kernel usually results in low performance because the branches are executed sequentially. The sample code of testing thread-dependent conditional branch is listed in Fig. 8, in which the variable lid stands for the unique local work-item ID within a specific work group. The $t1$ and $t2$ are two variables which are initialized to the local work-item ID and the global work-item ID in the kernel. We design six comparative kernels in this micro benchmark: one with no branch and the others diverging, respectively, 4, 16, 32 different paths in different threads within a warp according to their lid.

2.5 Scalability

In parallel computing area, scalability means the system's ability to increase speedup as the number of processors or cores increases. However, the number of PE and compute units is decided by native devices. The scalability that we measured in this

Fig. 8 Sample code of thread-dependent conditional branches

```
if(lid == 0) {
repeat128(t1/=t2; t2%=t1;)
}
else if(lid == 1) {
repeat128(t1/=t2; t2%=t1;)
}
...
else if(lid == 31) {
repeat128(t1/=t2; t2%=t1;)
}
```

micro benchmark is the ability to increase speed-up as the total number of work items is changed. The kernel contains a vector addition operation like Formula 3, in which both arrays a and b are inputs of the addition operations while array c saves the results, and index = gid * $s + i$, in which gid is the global work-item ID, s means the number of floats each thread should process which is calculated via the array length divided by total thread number, and i is a loop variable increasing from 0 to $s-1$. The total size of arrays a, b, and c is fixed as 1,024 $\times$ 1,024 $\times$ 64 bytes. We measure the execution time of this kernel when total number of work items increases from 1 to 65,536. For CPUs, when the number of threads is less than 256, the work-group size is half the number of threads, otherwise it is 256. While for GPUs, when the number of threads is less than 256, the work-group size is equal to the number of threads, otherwise it is 256.

$$c_{\text{index}} = a_{\text{index}} + b_{\text{index}} \tag{3}$$

2.6 Timing mechanisms

In our micro-benchmark suit, to calculate the FLOPS, memory bandwidth or other performance, we usually need to measure the execution time of kernels. We use the standardized mechanism for collecting kernel execution time provided by OpenCL. For each kernel function, it will run 10 times to calculate the average execution time. In practice, we find that from the fourth execution, the results are relatively stable. Therefore, we do not time the first three executions to prevent noises affecting the precision of measurement results.

3 Performance evaluation and discussion

In this section, we present and analyse the performance results drawn from running our micro benchmarks on four different computing devices, i.e. an AMD CPU, an Intel CPU, a NVIDIA GPU and an AMD GPU. The devices used for the experiments are listed in Table 2. We use AMD APP SDK V2.7 to run the micro-benchmark

Table 2 The devices used for the experiments

Compute device	GeForce GTX 460 SE	AMD Radeon HD 6850	AMD Athlon(tm) II X2 250	Pentium(R) Dual-Core E5400
Host	Pentium(R) Dual-Core E5400	Pentium(R) Dual-Core E5400	Itself	Itself
Device type	GPU	GPU	CPU	CPU
Engine clock	650 MHz	775 MHz	3,000 MHz	2,700 MHz
Memory	1 GB GDDR5	1 GB GDDR5	2 GB DDR2	2 GB DDR2
Memory clock	1,700 MHz	1,000 MHz	400 MHz	400 MHz
Memory bus width	256 bit	256 bit	64 bit	64 bit
Compute units	6	12	2	2
Process elements	288	960	2	2
PCI-E interface	PCI-E2.0 x16	PCI-E2.1 x16	–	–
SDK(driver version)	NVIDIA CUDA SDK 4.2.9(319.23)	AMD APP SDK V2.7(12.104)	AMD APP SDK V2.7	AMD APP SDK V2.7

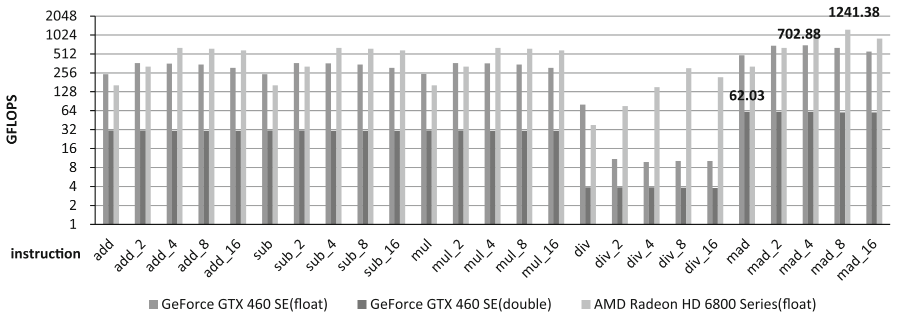


Fig. 9 GFLOPS of basic arithmetical instructions on GPUs

suite on both Intel's and AMD's CPU, because Intel's OpenCL SDKs do not support the Pentium Dual-Core 5400 CPU. All of these micro-benchmarks are compiled by GCC4.6.2 and tested under the Ubuntu12.04(Linux ubuntu 2.6.38-13-generic).

3.1 Mathematics

Figures 9 and 10 show the GFLOPS of basic arithmetical instructions that were measured by running our micro benchmark on GPUs and CPUs.

We can clearly find that we can get much higher GFLOPS on GPUs than CPUs, especially for single-precision floating operations. When using different vector sizes, the GFLOPS performance will be different both on GPUs and CPUs. However, the vector size is not always the bigger the better. Each device has its most suitable vector size. In the case of single-precision floating point test, GTX460SE gets a maximum 702.87 GFLOPS using float4 vectors when thread number and work-group size are set to 16,384 and 256, respectively. While AMD 6,850 gets a maximum 1,241.38 GFLOPS

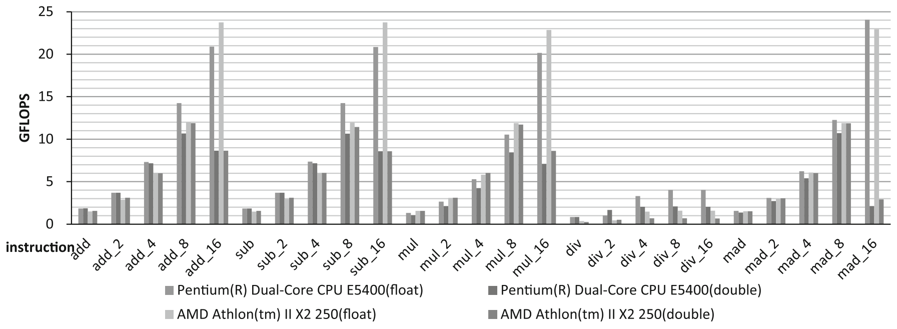


Fig. 10 GFLOPS of basic arithmetical instructions on CPUs

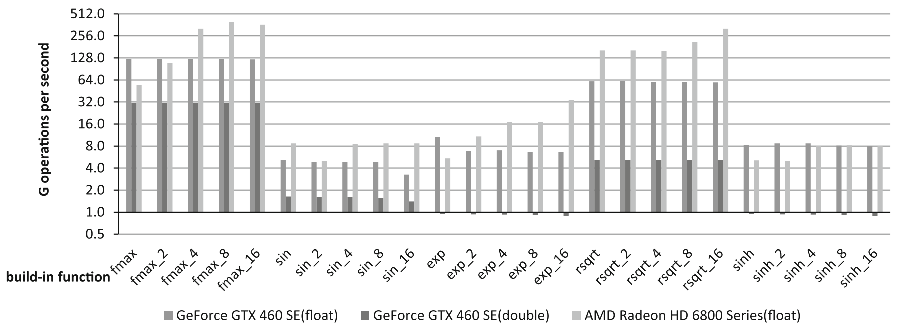


Fig. 11 Performance of built-in functions on GPUs

using float8 vectors when thread number and work-group size are set to 32,768 and 256, respectively. Both NVIDIA GTX460SE and AMD HD6850 get maximum GFLOPS when executing *mad* instruction, which combines the addition and multiply operations in one instruction. In the case of double-precision floating point test, we find that GTX460SE gets a maximum result of 62.03 GFLOPS without using vector for *mad* instruction while AMD HD6850 does not support double-precision floating point calculations.

Figures 11 and 12 show the performance of five typical built-in functions by our micro benchmarks. For the built-in functions, we measure how many function operations, instead of floating-point operations, executed per second. This micro benchmark evaluates the comprehensive floating-point calculating capability for both OpenCL runtimes and hardware devices.

3.2 Bus bandwidth

Figure 13 shows the performance results by the bus bandwidth micro benchmark on four computing devices. We can find that either on AMD platform or on NVIDIA platform, the bandwidths of GPU memory are much higher than the main memory on CPU, since the GPU memory has much wider bus bandwidth and higher memory clock rate than CPU. We also find that the transmission bandwidths between the host

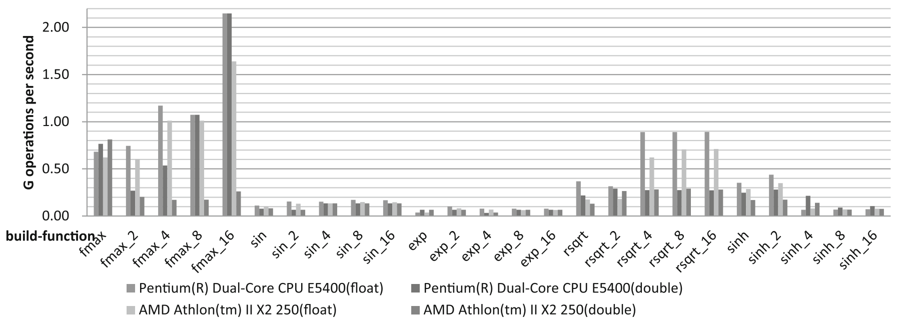
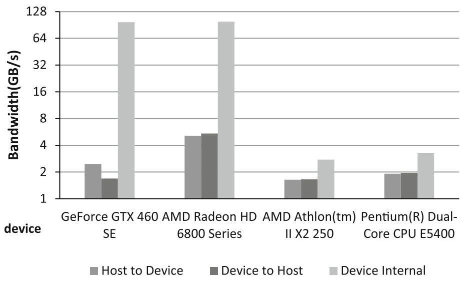


Fig. 12 Performance of built-in functions on CPUs

Fig. 13 Performance of bus bandwidth micro benchmark



and the GPU device are relatively low. The data exchanged between the host and devices are transmitted through the PCI-E bus on GPU platforms. The data transfer rates on the PCI-E 2.0 bus are about 8 GB/s, while the data transfer rates inside the GPUs are around 86 GB/s.

3.3 Device memory bandwidth

The device memory bandwidth micro benchmark focuses on the memory bandwidth and access characteristics for different memory regions. For a multi-core CPU system, all the memory regions are allocated in the main memory, and there is no significant performance distinguished among the global memory, local memory and constant memory. Therefore, when discussing the memory bandwidth and the performance characteristic of multi-core CPUs, we only need to discuss the global memory.

3.3.1 Global memory bandwidth

Figures 14 and 15 show the read and write bandwidths of the global memory on CPUs and GPUs when the variable $stride$ in Formula 1 changes, respectively.

We can clearly see that the global memory bandwidths gradually decrease when $stride$ increases from 1 to 16 both on AMD's GPU and NVIDIA's GPU. This is due to the existence of coalesced and non-coalesced memory access patterns. On GPUs, coalescing happens even if some threads in the group do not access any element. It means that some elements accessed within a segment would be discarded, and it results

Fig. 14 Performance of reading global memory on four devices with different strides

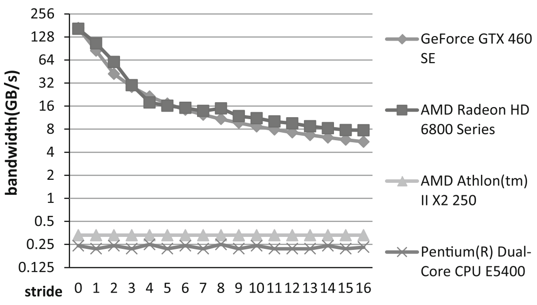
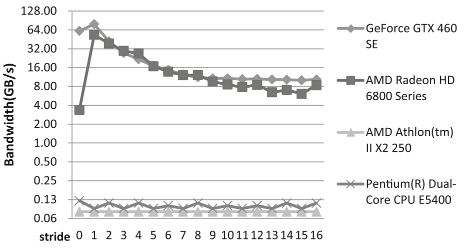


Fig. 15 Performance of writing global memory on four devices with different strides



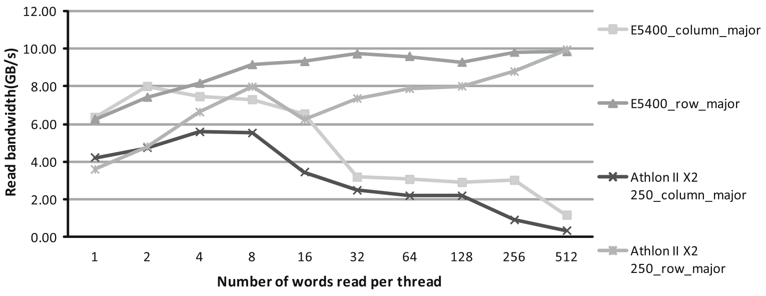


Fig. 16 Read bandwidth of CPUs when accessing global memory in column major and row major form

in a waste of bandwidth. With the increment of the *stride* value, access addresses of threads among a group become more disperse, and bandwidths of accessing global memory decrease. The coalesced and non-coalesced memory access modes do not exist for CPUs, so the memory bandwidths of CPU in Figs. 14 and 15 remain almost unchanged.

There is a special case when *stride* equals to zero. In this case, all threads in a group access the same address. It leads to relatively high read bandwidths and lower write bandwidths on both NVIDIA's and AMD's GPUs. The read and write behave differently due to the cache coherence protocols involved when writes are propagated.

Figures 16 and 17 show the read and write bandwidths of the global memory on CPUs when accessing the global memory in row major form and column major form, respectively.

Figures 18 and 19 show the read and write bandwidths of the global memory on GPUs when accessing the global memory in row major form and column major form, respectively.

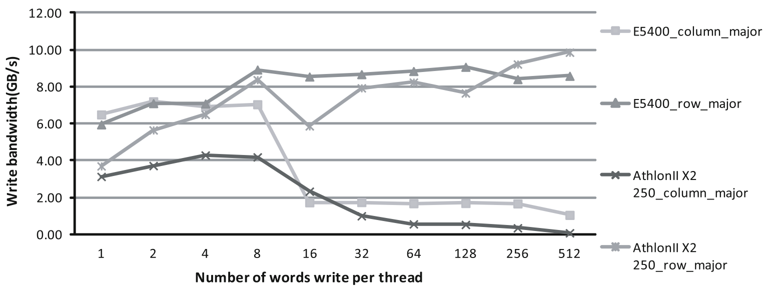


Fig. 17 Write bandwidth of CPUs when accessing global memory in column major and row major form

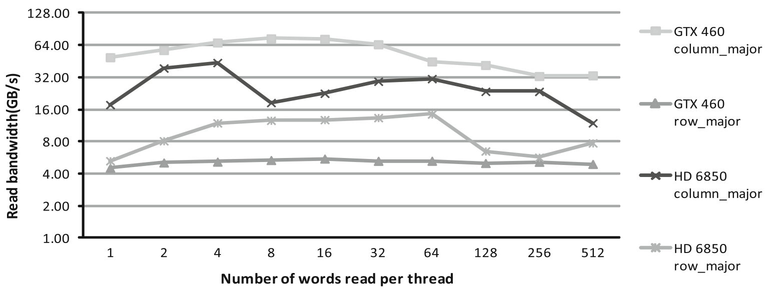


Fig. 18 Read bandwidth of GPUs when accessing global memory in column major and row major form

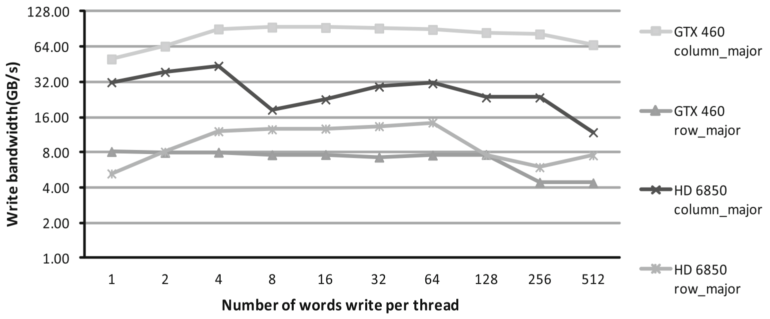


Fig. 19 Write bandwidth of GPUs when accessing global memory in column major and row major form

Both on Intel's and AMD's CPUs, the accessing bandwidths in row major form are higher than the bandwidths in column major form. Due to the cache effects on CPUs, as the number of bytes processed per thread rises, the bandwidth in row major form increases while the bandwidth in column major form decreases. When accessing array in row major form, the performance of cache is better with each thread to read or write more elements. However, when accessing array in column major form, cache misses always happen and with each thread to read or write more elements, they will occur more frequently, then led to the bandwidth degradation. In Fig. 17, the performance of E5400 in column major form degraded obviously when the number of float each thread writing reaches 16, this might be relevant to the cache specification of E5400. Both L1 and L2 data caches of E5400 have 8-way set associativity and 64 bytes line size. On the contrary, accessing in column major form gets higher bandwidth than the one

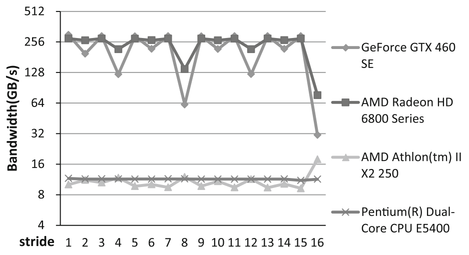


Fig. 20 Performance of reading local memory on four devices with different strides

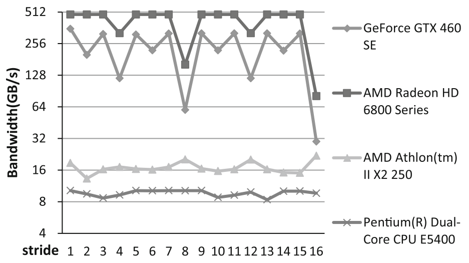


Fig. 21 Performance of writing local memory on four devices with different strides

obtained in row major form on both NVIDIA's and AMD's GPUs. We can infer that the global memory bandwidth is mainly influenced by coalesced and non-coalesced on GPUs and by cache on CPUs.

When implementing OpenCL applications on GPUs and CPUs, we should take the performance feature of different global memories into account. For instance, Helluy [11] implemented the radix sort algorithm using OpenCL. The author organizes the data in a special order first, and his purpose is to help coalesced access to global memory for better performance.

3.3.2 Local memory bandwidth

Figures 20 and 21 show the performance results of measuring the local memory bandwidths on different devices.

The bank conflict effect is only associated with GPUs, excluding CPUs. When the *stride* value equals to an odd number, all threads in a half-warp will access different banks. There is no bank conflict, leading to a high bandwidth. For NVIDIA GTX 460SE, when *stride* equals to 2, 4, 6, 8 or 16, there will be a 2-way, 4-way, 6-way, 8-way or 16-way bank conflicts, while for AMD HD6850, the bank conflicts occur only when *stride* equals to 4, 8, 12 and 16, which also verified that the local memory is divided into 16 banks on NVIDIA GTX460 and 32 banks on AMD HD6850, respectively. A memory request that has n-way bank conflicts will be split into n conflict-free requests. It will decrease the effective bandwidth by a factor of n.

3.3.3 Constant memory bandwidth

Figure 22 shows the constant memory bandwidths on four devices. The OpenCL programming guide states that the constant memory space should be cached [12]. We

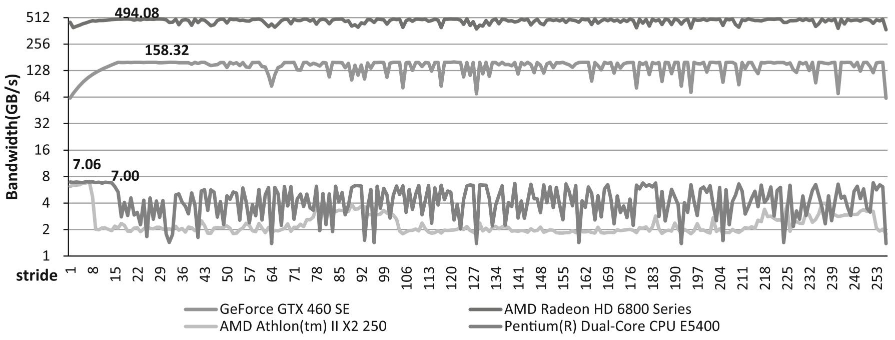


Fig. 22 Performance of reading constant memory on four devices with different strides

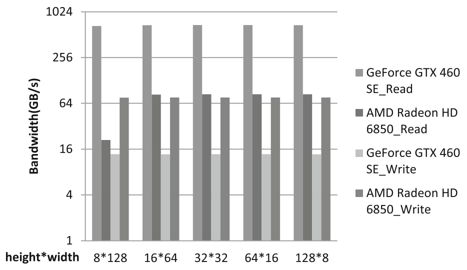


Fig. 23 Bandwidth of image objects which have the fixed sizes and different heights and widths on GPUs

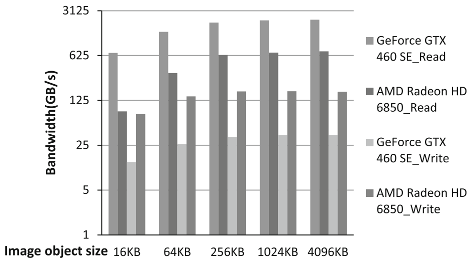


Fig. 24 Bandwidth of image objects which have the same heights and widths with different sizes on GPUs

can observe from Fig. 22 that the bandwidth grows a little up when *stride* increases to approximately 16 for GPUs, while for CPUs it goes down because with *stride* gradually increases, the interval between two sequential accessing addresses increases as well. But then it remains clearly stable for any device, with noise.

3.3.4 Image object bandwidth

Figure 23 presents the read/write bandwidths of image objects, which have a fixed size while *height* and *width* are changed. Figure 24 presents the read/write bandwidths of image objects, which have the same *height* and *width* when the object sizes increase from 16 to 4,096 KB.

Bandwidths of reading image object are much higher than writing. The image object is either read-only or write-only at any time, so we can speculate that the read and write accesses to image object may use different mechanisms. We also see that the

Table 3 Branch architecture test results

Devices	Without branch (ms)	4 branches (ms)	16 branches (ms)	32 branches (ms)
AMD AthlonII X2 250	731.75	740.77	747.01	754.78
Pentium(R) Dual-core CPU E5400	916.26	929.76	931.27	935.88
NVIDIA GeForce GTX 460SE	2.05	6.55	26.78	60.12
AMD Radeon HD 6850	2.53	8.77	34.95	67.69

changes of bandwidths with different height and width values are very small except for the 8 × 128 case on the AMD HD6850 GPU.

Although the performance of NVIDIA GTX460SE and AMD HD6850 manifested little difference, we think that the height and width of image object still need to be considered when designing the OpenCL program. For instance, Sun et al. [13] introduced how to port the original CUDA version of Kirchhoff Migration algorithm to OpenCL, and how to optimize the OpenCL program. In this paper, the authors made some statistics by folding one-dimensional image objects in two-dimensional ones with different height × width like 8,192 × 4, 4,096 × 8, 2,048 × 16 etc. When the height and width are 256 × 128, it got the peak performance on the NVIDIA 8800GT GPU.

3.4 Branch

The execution time of the branch micro benchmark is listed in Table 3. Branches in kernel have small impact on CPUs because CPUs have mature technologies like branch target buffer (BTB) dedicated to branch prediction and pipeline prefetching, and do not serialize the branches with thread-dependent conditions within a warp. However, for GPUs, the execution time of the kernel with thread-dependent conditional branches is almost 32 times as many as the execution time of the kernel without branch. For instance, for NVIDIA GeForce GTX 460SE, instructions are issued per 32 threads (a warp). In general, the 32 threads in a warp execute a single common instruction at the same time unless the threads within a warp diverge due to a thread-dependent conditional branch. Thus, the different execution paths within a warp will be executed sequentially.

3.5 Scalability

Figure 25 shows the execution time of kernels when the number of work item increases gradually from 1 to 65,536 both on CPUs and GPUs.

In this case, GPUs reflect a good scalability. By contrast, the scalability of CPUs is much worse compared with GPUs. For GPU platforms, the execution time will be gradually reduced with increasing the work items number. However, the execution time will stabilize after the numbers of work item increase to 2 on CPUs. Due to the limited hardware resources on CPUs, as the number of work item continues to increase

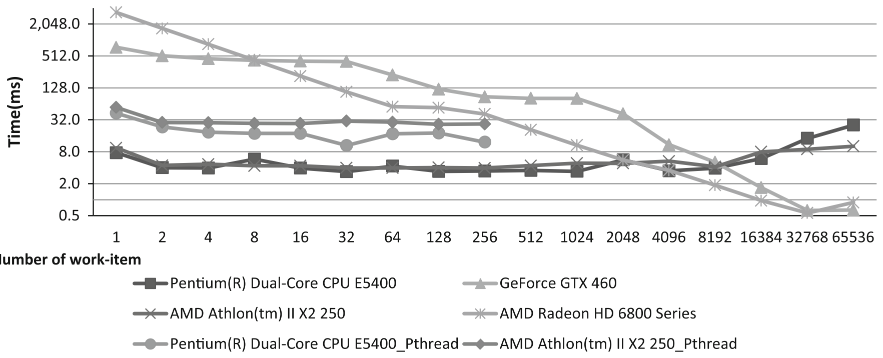


Fig. 25 Execution time of kernels on GPUs and CPUs when work-group size is fixed at 256

Table 4 Results comparison between our micro benchmarks and SHOC

Devices	Our benchmark		SHOC	
	GTX 460SE	HD6850	GTX 460SE	HD6850
BUS speed (HtoD)(GB/s)	2.7166	5.3704	2.4672	5.1282
BUS speed (DtoH)(GB/s)	1.7719	5.4329	1.6887	4.3151
MaxFlops (GFLOPS)	702.88	1,567.96	731.752	1,572.3
Globalmemory readbw (GB/s)	86.59	117.12	84.5758	114.922
Globalmemory writebw (GB/s)	78.95	53.51	78.76	53.0488
Localmemory readbw (GB/s)	300.63	380.53	300.157	374.587
Localmemory writebw (GB/s)	354.21	580.13	353.408	563.811

to 8,192, the execution time suddenly surges on CPUs. It may result from the great spending of thread switching.

In addition, we also tested the performance of the same vector addition function implemented using Pthread on the same CPUs, as Fig. 25 shown. The pthread version ran with a limitation of 256 threads. We can find that doing the same amount of computation on the same CPUs, OpenCL performs much better than pthread. Therefore, we believe that OpenCL is not only developed for GPU-based computing, it is also a good alternative for multi-core CPU programming.

3.6 Comparison with hardware configurations and SHOC

SHOC [8] also provides micro benchmarks to assess some architectural features of the computing devices, including bus speed, device memory bandwidth and peak FLOPS, etc. To demonstrate our contribution and further validate the accuracy of our micro benchmark suite, we compared the test results which are measured, respectively, by running our micro benchmarks and SHOC. Table 4 shows the comparisons. The columns of AMD HD6850 were measured using AMD AthlonII X2 250 CPU as the host for both SHOC and our micro-benchmark suite.

As what afore-mentioned, our micro benchmarks provide much more details compared with SHOC in terms of the mathematical operations, branch , scalability, as well as the bandwidths of the global and local memories with different accessing modes. For instance, we measure the GFLOPS for all the mathematical instructions and built-in functions, instead of the single peak value by SHOC. For instance, when measuring the global memory bandwidth, our micro benchmarks measure various scenarios by adjusting the value of $stride$ in Formular 1 from 0 to 16, while SHOC only considers the best and worst cases. For instance, when measuring the local memory bandwidth, our micro benchmarks consider the impact of bank conflicts to the bandwidth as well.

However, we still chose some common features of the two benchmark suites to compare the testing results by both suites on the same hardware and software platforms. For instance, the manufacturers declared that the theoretical peak GFLOPS of GTX 460 and HD 6850 is 748.8 and 1488, respectively. The theoretical peak device memory bandwidths of GTX 460 and HD 6850 are 108.8 and 128 GB/s, respectively. Both testing results of our suite and SHOC are close to the theoretical peaks, as Table 4. Table 4 also shows, in most cases, our micro benchmark gets more comprehensive results compared with SHOC, in terms of PCI-E bus speeds, global memory bandwidths, as well as local memory bandwidths.

4 Conclusion

The micro-benchmark suite demonstrates the ability of differentiating OpenCL runtimes and hardware architecture characteristics for multi-core CPUs and GPUs. Understanding these characteristics and differences is important for optimizing and implementing OpenCL applications on different micro architectures. For instance, we demonstrate the contrary behaviour of global memory on GPUs and CPUs when accessing an array in column and row orders. We introduce the designs and implementations of a series of micro benchmarks in detail, including mathematical operations, bus bandwidth, memory hierarchy, branch penalty and scalability, etc. We run the micro benchmarks on different computing devices, i.e. an AMD multi-core CPU, a Intel multi-core CPU, an AMD GPU and a NVIDIA GPU, and give a thorough analysis and research of the results.

Acknowledgments This material is based upon works supported by National Natural Science Foundation of China No.61073010 and No.61272166, National Science and Technology Major Project of China No.2012ZX01039-004, and the State Key Laboratory of Software Development Environment of China No.SKLSDE-2012ZX-02.

References

1. The OpenCL official site, at URL:http://www.khronos.org/opencl/
2. Seo S, Jo G, Lee J (2011) Performance characterization of the NAS Parallel Benchmarks in OpenCL. In: Proceedings of 2011 IEEE International Symposium on Workload Characterization (IISWC), IEEE, pp 137–148
3. Volkov V, Demmel JW (2008) Benchmarking GPUs to tune dense linear algebra. In: Proceedings of the 2008 ACM/IEEE conference on Supercomputing. IEEE Press, USA, p 31
4. Parboil Benchmark suite, at URL: http://impact.crhc.illinois.edu/parboil.php

5. Che S, Boyer M, Meng J et al (2009) Rodinia: a benchmark suite for heterogeneous computing. In: Proceedings of IEEE International Symposium on Workload Characterization 2009 (IISWC 2009), IEEE, pp 44–54
6. Torres Y, Gonzalez-Escribano A, Llanos DR (2013) uBench: exposing the impact of CUDA block geometry in terms of performance. J Supercomput 1–14
7. Shen J et al (2012) Performance gaps between OpenMP and OpenCL for multi-core CPUs. In: Proceedings of 2012 41st international conference on parallel processing workshops (ICPPW), IEEE, pp 116–125
8. Danalis A, Marin G, McCurdy C et al (2010) The scalable heterogeneous computing (SHOC) benchmark suite. In: Proceedings of the 3rd workshop on general-purpose computation on graphics processing units, ACM, pp 63–74
9. The OpenCL 1.2 specification, at URL: http://www.khronos.org/registry/cl/specs/opencl-1.2
10. Torres Y, Gonzalez-Escribano A, Llanos DR (2011) Understanding the impact of CUDA tuning techniques for fermi. In: Proceedings of 2011 international conference on high performance computing and simulation (HPCS), IEEE
11. Helluy P (2011) A portable implementation of the radix sort algorithm in OpenCL, at URL: http://hal.archives-ouvertes.fr/hal-00596730, Technical Report
12. OpenCL Programming Guide Version 2.3. at URL: http://www.nvidia.com/content/cudazone/download/OpenCL/NVIDIA_OpenCL_ProgrammingGuide
13. Peiyuan S, Xiaohua S (2012) An OpenCL approach of prestack Kirchhoff time migration algorithm on general purpose GPU. In: Proceedings of the 2012 13th international conference on parallel and distributed computing, applications and technologies, IEEE Computer Society
14. Wong H, Papadopoulou MM, Sadooghi-Alvandi M et al (2010) Demystifying GPU microarchitecture through microbenchmarking. In: Proceedings of 2010 IEEE international symposium on performance analysis of systems & software (ISPASS), IEEE, pp 235–246