GLOpenCL: OpenCL Support on Hardware- and Software-Managed Cache Multicores

Konstantis Daloukas
Department of Computer and Communications Engineering
University of Thessaly
Volos, Greece
kodalouk@inf.uth.gr

Christos D. Antonopoulos
Department of Computer and Communications Engineering
University of Thessaly
Volos, Greece
cda@inf.uth.gr

Nikolaos Bellas
Department of Computer and Communications Engineering
University of Thessaly
Volos, Greece
nbellas@inf.uth.gr

ABSTRACT

OpenCL is an industry supported standard for writing programs that execute on multicore platforms as well as on accelerators, such as GPUs or the SPEs of the Cell B.E. In this paper we introduce GLOpenCL, a unified development framework which supports OpenCL on both homogeneous, shared memory, as well as on heterogeneous, distributed memory multicores. The framework consists of a compiler, based on the LLVM compiler infrastructure, and a run-time library, sharing the same basic architecture across all target platforms. The compiler recognizes OpenCL constructs, performs source-to-source code transformations targeting both efficiency and semantic correctness, and adds calls to the run-time library. The latter offers functionality for work creation, management and execution, as well as for data transfers. We evaluate our framework using benchmarks from the distributions of OpenCL implementations by hardware vendors. We find that our generic system performs comparably or better than customized, platform-specific vendor distributions. OpenCL is designed and marketed as a write-once run-anywhere software development framework. However, the standard leaves enough room for target platform specific optimizations. Our experimentation with different, customized implementations of kernels reveals that optimized, hardware mapped implementations are both possible and necessary in the context of OpenCL – especially on non-conventional multicores – if performance is considered a higher priority than programmability.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*; D.3.4 [**PROGRAMMING LANGUAGES**]: Processors—*Code generation, Compilers, Optimization, Run-time environments*

General Terms

Design, Experimentation, Languages, Measurement, Performance

Keywords

OpenCL, Compilers, Runtime, Hardware-managed cache multicores, Software-managed cache multicores

1. INTRODUCTION

During the last few years, we have witnessed a paradigm shift towards parallel computing. Technology advances have enabled the integration of multiple cores in a single die, thus enabling the development of a plethora of computation substrates for parallel, high performance, computing. Architectures such as homogeneous or heterogeneous multicores and manycores, and, more recently, GPUs have allowed the time- and power-efficient execution of computationally intensive applications at a minimum expense. However, applications need to be rewritten in order to expose their inherent parallelism and exploit capabilities of multicore and accelerator-based architectures. Developing parallel programs is a far more complex undertaking than sequential programming, as the programmer is responsible for coping with hurdles such as race conditions, synchronization issues, communication, work management and scheduling, as well as with the peculiarities of each parallel architecture.

In order to enable the efficient development of applications, a multitude of programming models have been proposed, with the purpose of relieving the programmer from the duty of handling – at least some of – these issues thus allowing her to focus on algorithmic issues rather than technicalities. These models include PThreads [15], OpenMP [20], UPC [24] or Cilk [4] for homogeneous and heterogeneous multicores, the Cell SDK [10] and CellSs [21] for the Cell B.E., as well as CUDA [18] and the AMD/ATI Stream SDK [2] that target GPUs.

None of the aforementioned models targets both homogeneous and heterogeneous multicores, as well as accelerator-based systems at the same time, mainly due to their vast architectural differences. For example, applications that target the Cell B.E. are mainly developed using the IBM SDK, thus making their porting to other architectures a tedious task. There is, thus, strong demand for a programming model that enables application development without prior knowledge of the computing substrate where the application will be executed. Such a programming model would enhance portability, provided that it would allow applications to be transferred and efficiently executed on different architectures without requiring rewriting or widespread modifications. OpenCL [12] is an industry backed effort to develop a unified programming standard and infrastructure for heterogeneous multicore platforms integrating CPUs and, possibly, accelerators.

This paper discusses the design and presents the implementation of GLOpenCL (GLobal OpenCL), a unified compiler and run-time infrastructure. GLOpenCL is, to the best of our knowledge, one of the first OpenCL implementations that enables native execution on architecturally different systems, with both hardware- and software-controlled memory hierarchies. We discuss in detail the compiler transformations and run-time support that enable this functionality, as well as design decisions. Similar infrastructures include the OpenCL SDK [11] from IBM and the unified CUDA/OpenCL environment [17] by Nvidia that are customized to support OpenCL applications on the Cell B.E. and Nvidia GPUs respectively. The AMD/ATI Stream SDK [2] from AMD and ATI supports OpenCL on both x86 CPUs and ATI GPUs, however, as we show in Section 5 GLOpenCL significantly outperforms it on CPUs. MCUDA [23] and Ocelot [7] are two frameworks for efficiently supporting the CUDA programming model on x86 CPUs and the Parallel Thread Execution ISA (PTX) [19] on the Cell B.E. respectively. They differ from GLOpenCL as they target only a single hardware platform and are solely focused on compile-time transformations. Moreover they target CUDA and PTX instead of OpenCL codes.

Through the execution of a series of benchmark applications on two representative architectures, namely Intel's Core i7 (Nehalem) processor and the Cell B.E., we find that GLOpenCL performs comparably and often better than vendor implementations customized for the specific systems. Our system achieves an average speed up of 1.84x (maximum 2.67x) over the infrastructure from AMD/ATI for x86 systems, while resulting in low overhead compared to the IBM OpenCL implementation (for the Cell), especially for compute-bound applications. Based on experimental results, we can infer that the main bottleneck for the Cell implementation is the external software cache module that we utilized. OpenCL aims at being a reference, portable programming model for heterogeneous architectures. Our experimental evaluation indicates, however, that performance-conscious development, combined with a good insight on the underlying architecture, can result in significant benefits, especially on non-conventional multicores.

We briefly outline the OpenCL programming model in Section 2. Section 3 discusses the compiler support that is needed to enable efficient execution of OpenCL applications, while in Section 4 we introduce the run-time system's architecture. Next, in Section 5 we discuss the evaluation of our infrastructure on two multicore systems, one homogeneous with hardware- and one heterogeneous with software-controlled caches. Finally, Section 6 concludes the paper.

2. THE OPENCL PROGRAMMING MODEL

The OpenCL (Open Computing Language) [12] programming model tackles the problems of programmability and portability by providing a parallel programming framework suitable for conventional multicore architectures as well as for accelerators, such as the GPUs and the Cell processor. Applications developed with the OpenCL programming model can be ported, ideally unmodified, to any platform that supports this framework.

OpenCL models the underlying parallel architecture as a host and a number of OpenCL *compute devices*. A device integrates a number of *compute units*, each one divided into *processing elements*. The processing elements execute a single stream of instructions and can operate either as SIMD or as SPMD units.

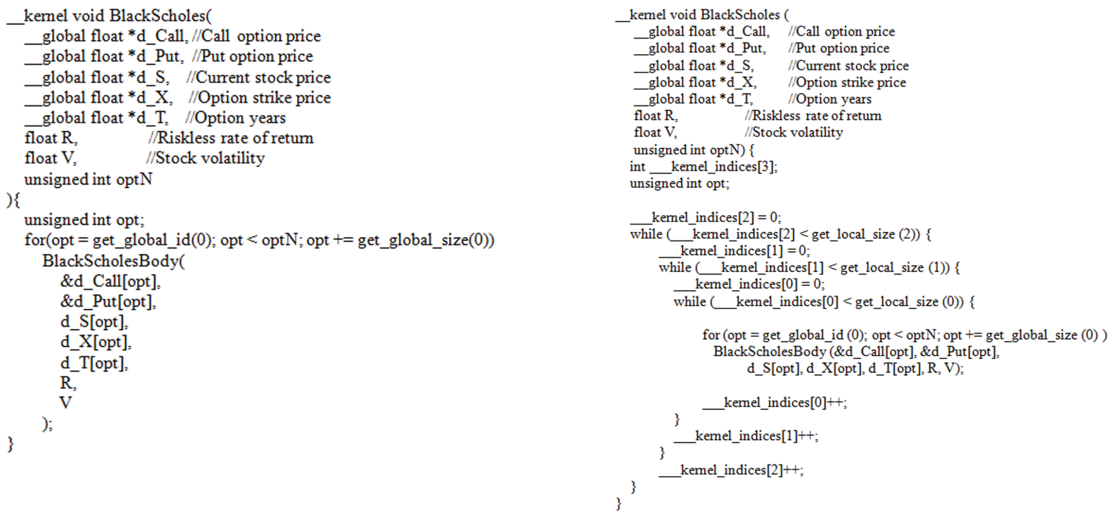
An OpenCL application consists of two parts: the main program that executes on the host and a number of *kernels* that execute on the compute devices. The main constructs in the OpenCL execution model are *command queues*, *kernels* and *memory buffers*. OpenCL kernels express the computational parts of the application. Command queues are utilized for coordinating the execution of kernels. The main program enqueues commands to the queues. Commands are subsequently scheduled for execution. A command queue can store either commands that trigger a kernel's execution or commands for manipulation of memory buffers. Memory buffers provide the means of data exchange between the host and the compute devices. The corresponding commands are either read or write operations for a particular memory buffer.

OpenCL programmers typically use kernels for expressing parallelism at its finest granularity. The "geometry" of the execution is described by a 2-level, 3D *index* space, which is defined upon execution of a kernel command. Each point in the index space is called a *work-item* and corresponds to the execution of a particular instance of the kernel. Each work-item is described by a unique tuple of ids. Work-items are organized into *work-groups*, each having up to three dimensions (3D thread index within the work-group geometry). The overall computation can, in turn, be partitioned in work-groups, also organized in a 3D space (3D work-group index within the global computation geometry). OpenCL provides functionality for synchronization among work-items that belong to the same work-group. On the other hand, work-groups are completely independent on each other and can execute in parallel. Therefore, the model does not provide primitives for synchronization among work-groups. Only work-items that belong to the same work-group can communicate directly, through memory which is visible only inside the work-group.

3. COMPILATION INFRASTRUCTURE

Typically, a kernel function in the OpenCL programming model describes the computation to be executed by a logical thread and expresses the application's parallelism at its finest granularity. However, the efficient exploitation of parallelism and its mapping to the execution contexts of the underlying architecture is not trivial. A straightforward approach would map each kernel invocation to a user- or kernel-level thread. This approach is acceptable on systems which provide explicit hardware support for fine-grained threading, such as GPUs. However, other architectures, such as conventional multicores or the Cell B.E., require coarser-grained parallelism, in order to limit the overhead of work chunk creation, management and execution. Moreover, the naive, fine-grained execution does not necessarily benefit from the potential spatial and temporal locality inside a work-group.

To enable efficient execution of OpenCL kernel functions, we apply a series of source-code transformations. These transformations collectively aim at coarsening the granularity of the kernel function from a per-logical-thread to a per-



```
__kernel void BlackScholes(
    __global float *d_Call, //Call option price
    __global float *d_Put,  //Put option price
    __global float *d_S,    //Current stock price
    __global float *d_X,    //Option strike price
    __global float *d_T,    //Option years
    float R,                //Riskless rate of return
    float V,                //Stock volatility
    unsigned int optN
){
    unsigned int opt;
    for(opt = get_global_id(0); opt < optN; opt += get_global_size(0))
        BlackScholesBody(
            &d_Call[opt],
            &d_Put[opt],
            d_S[opt],
            d_X[opt],
            d_T[opt],
            R,
            V
        );
}
```

(a)

```
__kernel void BlackScholes (
    __global float *d_Call,     //Call option price
    __global float *d_Put,      //Put option price
    __global float *d_S,        //Current stock price
    __global float *d_X,        //Option strike price
    __global float *d_T,        //Option years
    float R,               //Riskless rate of return
    float V,               //Stock volatility
    unsigned int optN) {
   int ___kernel_indices[3];
   unsigned int opt;

   ___kernel_indices[2] = 0;
   while (___kernel_indices[2] < get_local_size (2)) {
        ___kernel_indices[1] = 0;
        while (___kernel_indices[1] < get_local_size (1)) {
            ___kernel_indices[0] = 0;
            while (___kernel_indices[0] < get_local_size (0)) {

                for (opt = get_global_id (0); opt < optN; opt += get_global_size (0) )
                  BlackScholesBody (&d_Call[opt], &d_Put[opt],
                           d_S[opt], d_X[opt], d_T[opt], R, V);

                ___kernel_indices[0]++;
            }
            ___kernel_indices[1]++;
        }
        ___kernel_indices[2]++;
   }
}
```

(b)

Figure 1: Serialization of logical threads inside a kernel function. The kernel function before (a) and after (b) the transformation.

work-group basis, thus reducing overheads. This approach has the additional benefit of reducing the memory footprint of the application as it allows different logical threads to share the memory used for common variables. After the transformations, the modified kernel function represents the work that must be executed by each work-group in the index space of the application. OpenCL work-groups can execute in parallel. Therefore, it is possible to map each invocation of the modified kernel to an execution context of the architecture, as described in more detail in Section 4.

The transformation process consists of three main steps: serialization of logical threads, elimination of synchronization functions within the kernel, and identification of variables that cannot be shared among logical threads. We have modified the Clang [6] front-end of the LLVM compiler infrastructure [13] to support parsing of OpenCL kernel functions and perform transformations on the program's abstract syntax tree (AST).

Additional compiler transformations, necessary to integrate our framework with an external software cache module, and to support the execution of kernel functions with varying numbers and types of arguments are described in Sections 4.2 and 4.3 respectively.

3.1 Serialization of Logical Threads

The first step in the series of transformations aims at increasing the amount of computation in a kernel function by serializing the execution of logical threads. Figure 1 depicts a kernel function before (1(a)) and after (1(b)) the transformation. In the absence of synchronization operations, work-items (i.e. logical threads) inside a work-group can be executed in any sequence. We enclose the instructions in the body of a kernel function within a triple-nested loop – given that the maximum number of allowable work-group dimensions is currently three – thus executing the logical threads in sequence. Each loop-nest enumerates the logical threads in the corresponding work-group index dimension.

The selection of a work-group as the preferred degree of granularity for logical threads serialization may seem arbitrary. However, in the next section it will become evident that other options may present hard to overcome complications in the presence of synchronization operations or multiple exit points within the kernel. At the same time, work-group granularity is usually explicitly set by OpenCL programmers, often considering data reuse, or matching the work-group data footprint to the capacity of specific levels of the memory hierarchy. Therefore, introducing different degrees of work granularity at the run-time, despite being semantically correct, might introduce performance side-effects.

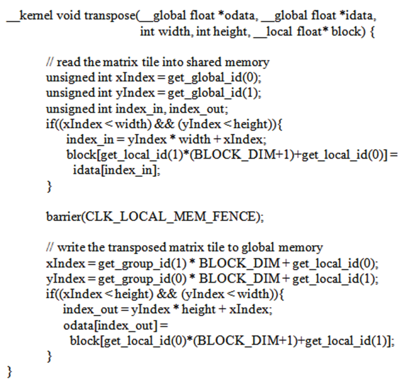
The reader can observe that the serialization transformation typically decreases the cumulative memory footprint of the kernel function invocations, since just one logical thread is active at any time, thus the memory allocated to local variables can be reused.

3.2 Elimination of Synchronization Operations

The next transformation addresses the problems introduced by synchronization operations or multiple exit points within a kernel. The OpenCL programming model provides the *barrier()* function to allow barrier-type synchronization of work-items inside a work-group. In the presence of a barrier, all work-items in the work-group must execute the barrier instruction before any of them is allowed to continue execution beyond the barrier. Similarly, a barrier command inside a loop implicitly enforces all work-items to execute the barrier before the next iteration of the loop. Finally, if a barrier is present in the block of a conditional statement, the programmer must ensure that there is no control flow divergence within the work-group at the particular condition, otherwise a deadlock is possible.

A kernel function coarsened at a work-group granularity implicitly enforces synchronization of the code corresponding to logical threads before its first and after its last iteration. By analogy, a barrier instruction requires that logical threads synchronize before traversing it.

To ensure correct execution of the coarsened kernel functions, we apply loop fission around each synchronization



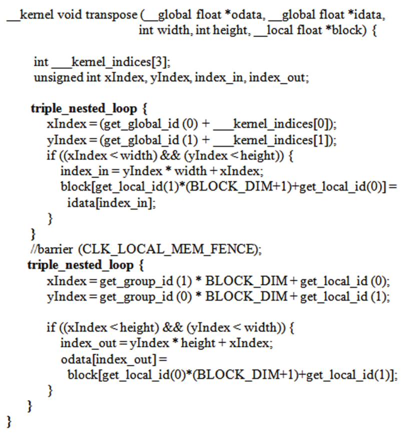
```
__kernel void transpose(__global float *odata, __global float *idata,
                         int width, int height, __local float* block) {

     // read the matrix tile into shared memory
     unsigned int xIndex = get_global_id(0);
     unsigned int yIndex = get_global_id(1);
     unsigned int index_in, index_out;
     if((xIndex < width) && (yIndex < height)){
         index_in = yIndex * width + xIndex;
         block[get_local_id(1)*(BLOCK_DIM+1)+get_local_id(0)] =
          idata[index_in];
     }

     barrier(CLK_LOCAL_MEM_FENCE);

     // write the transposed matrix tile to global memory
     xIndex = get_group_id(1) * BLOCK_DIM + get_local_id(0);
     yIndex = get_group_id(0) * BLOCK_DIM + get_local_id(1);
     if((xIndex < height) && (yIndex < width)){
         index_out = yIndex * height + xIndex;
         odata[index_out] =
          block[get_local_id(0)*(BLOCK_DIM+1)+get_local_id(1)];
     }
}
```

(a)



```
__kernel void transpose (__global float *odata, __global float *idata,
                          int width, int height, __local float *block) {

    int ___kernel_indices[3];
    unsigned int xIndex, yIndex, index_in, index_out;

    triple_nested_loop {
       xIndex = (get_global_id (0) + ___kernel_indices[0]);
       yIndex = (get_global_id (1) + ___kernel_indices[1]);
       if ((xIndex < width) && (yIndex < height)) {
          index_in = yIndex * width + xIndex;
          block[get_local_id(1)*(BLOCK_DIM+1)+get_local_id(0)] =
           idata[index_in];
       }
    }
    //barrier (CLK_LOCAL_MEM_FENCE);
   triple_nested_loop {
       xIndex = get_group_id (1) * BLOCK_DIM + get_local_id (0);
       yIndex = get_group_id (0) * BLOCK_DIM + get_local_id (1);

       if ((xIndex < height) && (yIndex < width)) {
          index_out = yIndex * height + xIndex;
          odata[index_out] =
           block[get_local_id(0)*(BLOCK_DIM+1)+get_local_id(1)];
       }
   }
}
```

(b)

Figure 2: Loop distribution (loop fission) around a synchronization statement. The Matrix Transpose kernel function before (a) and after (b) the transformation. `triple_nested_loop` stands – for brevity – for the triple nested loop introduced by the logical threads serialization pass.

statement. We partition the statements into blocks so that each block contains no synchronization operations. Figure 2 depicts this transformation for a kernel function that contains a barrier instruction. Since there is one synchronization statement, two loop constructs are required and sufficient to ensure correct execution of the kernel function for work-items inside a work-group.

A similar problem occurs in statement blocks with multiple exit points, i.e. when statements that change the control flow are present, such as *continue*, *break*, or *return*. If such a statement block is enclosed within the triple-loop nest, execution will be inconsistent with the program's semantics, since only the first logical thread will have the opportunity to execute the instructions before the statement. In order to overcome this issue, we treat such statements similarly to synchronization points and enclose the instructions before and after them in additional loop constructs.

Loop fission is applied as an iterative procedure, requiring several traversals of the AST. A transformation necessitated by a synchronization or a control flow statement may reveal other points in the code that have to be treated as synchronization points, thus requiring another traversal.

3.3 Variable Privatization

After applying loop fission around synchronization or control flow statements, the compiler needs to cope with variables whose life crosses loop fission points.

Each logical thread in the initial kernel function had its own private storage for its local variables. However, once serialization is applied, logical threads that belong to a work-group share the memory corresponding to local variables. This is normally both possible and legal, as each logical thread has finished its execution – and therefore no longer needs the value of the variable – before the iteration corresponding to the next logical thread starts executing. However, there is a complication for variables whose life extends beyond a synchronization or a control flow statement, i.e. variables defined and having values assigned before such a statement and reused after it. Values assigned by logical-threads at the first loop construct introduced by loop fission cannot be used during the execution of the second loop construct, as their content has been polluted by the execution of subsequent logical threads, thus violating semantics.

Our compilation infrastructure conducts a live variable analysis to identify the variables that are live beyond the boundaries of the loops introduced by loop fission. Following, we apply variable privatization [1] for these variables, namely we allocate them to a separate memory area for each logical thread. Each logical thread is therefore provided with a private copy of such variables. As a final step, references to those scalar variables are rewritten to references to the appropriate, thread-local position for each logical thread.

4. RUN-TIME SUPPORT

Figure 3(a) depicts the architecture of the GLOpenCL run-time system for a system with hardware-controlled memory hierarchy (Intel x86), while Figure 3(b) depicts the architecture for a software-controlled memory hierarchy multicore (Cell B.E.).

The run-time spawns a number of kernel-level threads, which are used either as execution vehicles, or as helper threads. The *main thread* is the one that executes the host side of the OpenCL application. *Worker threads* are created upon initialization and are responsible for executing the main computational tasks of the application. GLOpenCL run-time system spawns a number of worker threads equal to the number of execution contexts available on the underlying architecture. Finally, management and monitoring tasks are undertaken by a *helper thread.* The exact responsibilities and implementation of the helper thread are architecture specific.

4.1 Work Management

For each instruction of the host-side code that enqueues a command, the main thread creates an appropriate command descriptor and enqueues it in the command queue. The latter typically is a very lightly contended data structure. Mutually exclusive access to the queue is implemented using futexes [8]. Once the main thread has executed the last instruction of the host-side of the application, it blocks until all enqueued commands have been executed.

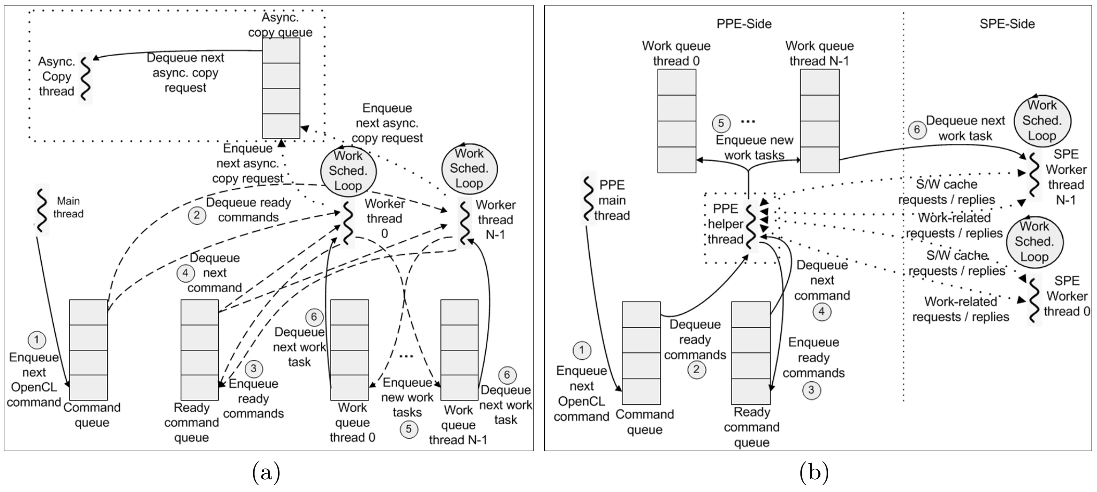


Figure 3: The run-time system's architecture for the Intel (a) and the Cell B.E. (b) architectures. Dotted lines identify architecture-specific modules or operations. The numbering denotes a typical sequence of operations.

Commands are transferred from the command queue to the ready command queue when they are ready to be executed. A command queue can be configured by the programmer to support either in-order or out-of-order execution. If in-order execution is enabled, a command is transferred to the ready queue if it is at the top of the command queue, and provided that all previously issued commands have finished their execution.

When the command queue has been configured for out-of-order execution, a dependence-driven self-scheduling scheme is applied. Each command can be marked as dependent on one or more prior commands. Each executed command updates the dependencies of its dependents. As soon as a dependent command is identified to have no unsatisfied dependencies, it is enqueued to the ready queue.

Work-tasks are created when a command related to a kernel execution is processed from the ready queue. Each work-task represents the computation that must be executed for a single work-group in the index space of the application. A work-task corresponds to an invocation of the corresponding modified kernel function. The number of work-tasks depends on the partitioning of the global index space. An OpenCL programmer can explicitly partition the global index space, by defining the dimension of work-groups. If this is not the case, the run-time performs an implicit, static partitioning of the index space, taking into account the number of worker threads. In order to avoid work-task starvation, execution of work-tasks has higher priority over the execution of ready commands. This means that a ready command is processed only when work queues are empty of work-tasks. In the x86 implementation commands are transferred from the command queue to the ready queue and processed by worker threads. As soon as a ready command is executed, the corresponding work-tasks are created and distributed to the work queues. This series of operations is carried out by the helper thread in the Cell implementation.

Worker threads continuously execute a scheduling loop. In each iteration, they retrieve the next available work-task from the top of the work queue and execute the corresponding wrapper kernel function. In the x86 implementation, a worker thread has direct access to the work queues. This is not the case for the Cell, where each worker thread is executed on an SPE, thus having no direct access to work queues residing in the address space of the PPE. When a worker has finished executing its task, it communicates with the helper thread through the mailbox mechanism to request the next available task. If there is one, the helper thread informs the worker thread, which subsequently transfers, through the DMA mechanism, the work-task descriptor and continues its execution. Otherwise, the helper thread marks the worker as idling until there is a new work-task.

We have opted for a per-thread work queue scheme, where each work queue is local to the corresponding worker thread. Multiple work queues reduce the synchronization overhead, enhance locality and improve the scalability of the run-time system. Locality benefits are particularly evident in systems with software-controlled cache hierarchies, such as the Cell B.E. Work tasks are statically partitioned to the available worker threads. Static partitioning tends to enhance locality, as neighboring work-groups often access data in close proximity.

To ensure proper load balancing among worker threads, we allow work-stealing on the work queues. Each worker that finds itself idling may attempt to steal work from the bottom of a work queue of another worker. In the x86 implementation workers perform work-stealing directly. A lock-free queue implementation enforces synchronization between threads that attempt to access the same queue simultaneously. In the Cell implementation, idle workers are assisted in work-stealing by the helper thread. Only the helper thread accesses the queues, therefore no synchronization is required.

4.2 Manipulation of Memory Buffers

In OpenCL, memory buffers provide the communication medium between the host and compute devices, as well as between work-items. They can be either *global* or *local.* Global memory buffers can be accessed by any work-item in the global index space while local buffers are only accessible by work-items inside a work-group. Global buffers present a great challenge in the run-time system's implementation for software-controlled cache architectures, such as the Cell B.E.

The capacity of the higher levels of the memory hierarchy (the Local Store in the SPEs) is limited and often insufficient to accommodate the working set of a work-group. The problem is further complicated when there is no hardware support for coherence between caches on different compute devices (Local Stores of different SPEs). At the same time, as we will discuss in Section 5, global buffers fit better to the notion of shared memory most programmers are familiar with. Therefore, they tend to be preferred over local buffers, even when this would not be technically necessary.

Both problems can be addressed by using a software cache module. The design and implementation of a new software cache mechanism is out of the scope of this work. Therefore, we have utilized the COMIC shared memory system, one of the few software caches available for the Cell B.E. [14]. COMIC is an unified software cache and threading system. We have modified it to operate solely as a software cache and integrated it with GLOpenCL. Each reference to a global buffer in the kernel function is rewritten by the compiler and gets redirected to the software cache mechanism. The software cache, in turn, performs all necessary operations on the Local Stores of different SPEs to guarantee the coherence of accesses to the corresponding global buffer. A client side of the software cache is executed at each worker thread (on SPEs) while the server side is executed in the context of a helper thread on the host (PPE). Requests from a software cache client are directed to the server, which returns the appropriate data to the corresponding client and updates other clients on potential coherence complications. Such a mechanism is obviously redundant in the x86 implementation as the caches are hardware-controlled and a coherence protocol is already in place.

The last component of the run-time system for the x86 architecture is the *async. copy thread* and the corresponding queue. OpenCL provides functions for asynchronously copying data from global buffers to local ones and vice versa inside a work-group. Any asynchronous function found inside a kernel must be executed by every work-item in the work-group. An asynchronous copy operation returns an event descriptor which can be used later to poll whether the operation has completed its execution. We enable asynchronous copy operations by utilizing a helper thread, the *async. copy thread*, to handle the corresponding requests. When a worker thread encounters such an asynchronous call, it enqueues a request in the *async. copy queue*. The async. copy thread is responsible for dequeuing the next available request, copying the appropriate data, and updating the corresponding event.

The implementation of asynchronous memory copies in architectures with software-controlled memory hierarchies is usually more straightforward. Such architectures tend to programmatically expose the DMA interface, as it is the preferred mechanism for transferring data between different levels of the memory hierarchy. On the Cell for example, an async. copy operation is implemented as one or more DMA transfers, which are inherently asynchronous.

4.3 Dynamic Kernel Invocation

Once a worker thread is assigned a work-task, it must invoke the corresponding modified kernel function with the appropriate number of arguments. The optimal approach to support the dynamic invocation of a kernel function is through a function pointer. This approach requires that both the address of the function and the number of its arguments are known. An OpenCL application can invoke multiple kernel functions during its execution life, identifying them by their name, namely a string. Moreover, different kernel functions can have different numbers and types of arguments. The complications are thus two-fold: a) The run-time system must have prior knowledge of the number and type of each kernel function's arguments, and b) It should be able to obtain the address of the function based on its name.

In order to support kernel functions with varying numbers and types of arguments, we provide a uniform interface for the invocation of a kernel function. Therefore, the compiler creates a wrapper for each modified kernel function in the application. A wrapper function has only one argument (an array of void pointers) and is responsible for dispatching the kernel's arguments as the corresponding function parameters, as well as for invoking the function. Therefore, work-task descriptors only need to contain the address of the corresponding wrapper function and a pointer to its argument.

Overcoming the second complication is trivial on systems that support dynamic linking, as is the case in the x86 implementation. At link-time, all symbols are added to the symbol table of the executable. Then, we exploit the functionality of the run-time dynamic linker to obtain the stub function's address, based on its name.

An alternative approach needs to be followed on statically linked binaries, or when different executables are produced for the host and the compute devices. This is, for example, the case for the Cell implementation, as the executable for the SPEs, which contains the kernel functions, is statically linked with the executable of the PPE to produce a single binary. Therefore, no dynamic lookup is possible. The proposed approach is based on the observation that both PPE and the SPEs have access to the regions of the ELF binary that is produced. We allocate a separate region in the binary and make its address known to the SPEs upon initialization. For each wrapper function in the program, this section stores a tuple that consists of its name and its address. When a pointer to a function is needed, the PPE accesses the common section, locates the position of the wrapper function, and stores the offset in the work-task descriptors that are produced. This offset is subsequently used by the SPEs in order to index the section and obtain the pointer to the appropriate function.

The two aforementioned approaches are complementary. The latter has the additional positive effect of allowing us to overcome the limited capacity of the Cell B.E. Local Store with respect to application code size. The Local Store in the SPEs is unified and stores both the data and the instructions of the program. As a consequence, this space may not suffice for applications that execute multiple kernels throughout their life, as the Local Store may be unable to concurrently accommodate the source code for every function. Invoking a function through a pointer can be used along with the SPU code overlay mechanism to enable dynamic loading of a function's code in the Local Store of the SPE, thus virtually extending its capacity.

5. EXPERIMENTAL EVALUATION

In this section we present the experimental evaluation of GLOpenCL on two different platforms: an homogeneous

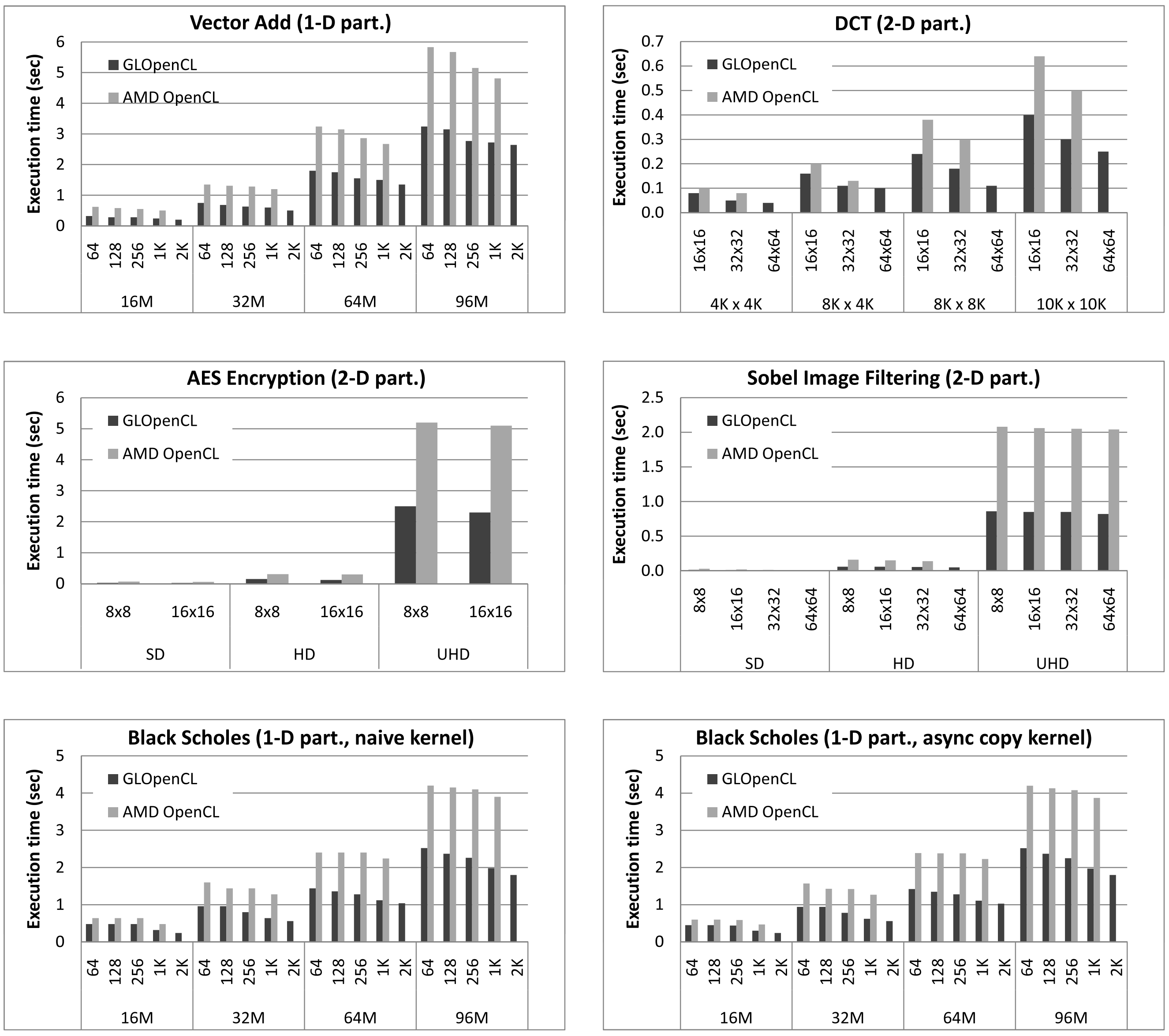


Figure 4: Performance comparison between GLOpenCL and the ATI/AMD OpenCL SDK on the x86. Experiments have been executed for varying data sets (lower x-axis labels) and work-group sizes (upper x-axis labels).

multicore with hardware-controlled memory hierarchy, and an heterogeneous multicore with software-controlled caches. Our homogeneous multicore is an Intel-based workstation, using an Intel E5520 i7 processor clocked at 2.27 GHz and 4 GB of RAM. The processor integrates four identical cores. Each core has a 32 KB instruction + 32 KB data private L1 and a unified 256 KB L2 cache. All four cores share an 8 MB L3 cache. Cell B.E. is a typical representative of heterogeneous multicores with software controlled memory hierarchy. We experimented with a IBM QS20 Blade with two Cell processors clocked at 3.2 GHz. Only one of them was used throughout the evaluation. The processor integrates a 2-way SMT PowerPC core (PPE) and 8 synergistic processing element cores (SPEs). Each SPE accesses a private, software-controlled, 256 KB local storage (LS). The blade is also equipped with 1 GB of RAM.

We compare the performance of GLOpenCL with that of vendor provided, platform-customized implementations, namely the AMD/ATI Stream SDK (v2.1) on x86 and the IBM OpenCL SDK (v0.1.1) on the QS20 Cell blade. We used the following set of representative applications, with diverse characteristics, from the two SDK distributions:

- `Vector Add`: Addition of 2 vectors. No temporal locality, some spatial locality, communication intensive.
- `2D DCT`: Discrete Cosine Transform on an image. Blocked, temporal and spatial locality, compute bound.
- `Sobel`: Sobel filter [22] on an image. Blocked, similar characteristics with 2D DCT.
- `2D AES`: AES encryption [16] of an image. Similar characteristics with 2D DCT and Sobel, the most compute intensive application in the set, as it was reported both by Intel VTune [9] for the x86 and the Cell Performance Counter tool (CPC) [5] for the Cell B.E.
- `BlackSholes`: Solution of Black-Scholes PDEs [3] for a number of options. Some spatial locality, no temporal locality, compute intensive, complex math operations. The async. version uses double buffering and explicit, asynchronous blocked transfers for data buffers, instead of implicitly outsourcing communication to the hardware or software cache mechanism.

All codes are precompiled, i.e. the just-in-time compilation capabilities of OpenCL have not been used. GLOpenCL application binaries have been created with the Intel C Compiler v11.1 on x86 and with xlc on Cell. In all cases, we applied the set of optimization flags that resulted in the highest performing binaries.

We have executed each application with varying data sets and work-group sizes. To ensure a fair comparison between the different implementations, we measure only the kernel execution time in each application. For each configuration, we have executed a series of ten experiments and we report the mean execution time. In each case, the variance of the execution time was insignificant, which means that all results are reliable.

Figure 4 depicts the performance evaluation results on the x86 platform. GLOpenCL consistently outperforms the AMD/ATI OpenCL SDK. The average performance improvement is 1.84x whereas the maximum speedup is 2.67. We cannot know the exact reason for this performance difference as internal details of the AMD/ATI OpenCL SDK are not publicly available. Execution time in all applications scales linearly to the amount of work. Moreover, larger work-group sizes consistently tend to be beneficial for performance, as less work-tasks need to be created, thus reducing work-chunk creation, management and execution overhead. At the same time, larger work-groups implicitly favor temporal locality, especially in blocked codes.

Another interesting observation is that the substitution of hardware assisted buffer transfers with asynchronous, double-buffered copies in the case of BlackScholes does not seem to offer any measurable performance benefits. Modern, hardware-controlled memory hierarchies have the potential of effectively hiding memory access latencies, at least for memory transfers that do not require multi-hop interconnection network transactions and for regular loops in which traditional optimizations – such as unrolling – or more aggressive techniques – such as compiler assisted prefetching – can be applied. This is particularly true for multicore chips with large, shared outer-level caches. Therefore, such architectures are, by nature, more forgiving to suboptimal implementations.

Finally GLOpenCL proved able to work with both larger work-groups and tackle larger data sets / problem sizes. The largest problem size reported in the charts, with the exception of AES and Sobel, is the limit beyond which the AMD/ATI OpenCL SDK did not manage to allocate buffers. Moreover, in certain cases the AMD/ATI OpenCL SDK failed to exploit larger work-groups for a certain problem size. Such behavior can be observed for Vector Add, DCT and both versions of BlackScholes.

Figure 5 summarizes the evaluation on the Cell blade. The IBM OpenCL SDK is performing on average 27.5% better than GLOpenCL. This can be mainly attributed to the difference in the efficiency of the software cache implementations used by each system. IBM uses a custom software cache, which significantly outperforms the one publically available within the IBM Cell SDK. Moreover, the software cache is tightly integrated with the OpenCL-enabled xlc compiler, thus allowing detailed compile-time data access pattern analysis and optimizations. It is characteristic that for the communication-intensive Vector Add application IBM OpenCL outperforms the GLOpenCL and COMIC combination by 52.6%. Excluding Vector Add, the performance gap between the GLOpenCL and IBM infrastructure drops to 19.1%, or 13.2% if the best performing work-group geometry is used for each problem size. Moreover, for compute-intensive applications, such as AES and Sobel, the performance of the two systems is practically indistinguishable, especially for larger problem sizes.

IBM OpenCL outperforms GLOpenCL even in the BlackScholes implementation using asynchronous copy, thus reducing the effect of the software cache. As discussed in Section 4.2, COMIC integrates the software cache with a threading system. Although we have deactivated the threading system and combined COMIC with the GLOpenCL runtime, some helper threads primarily used for monitoring COMIC threading are tightly integrated with the software cache module and cannot be deactivated. They, thus, interfere with GLOpenCL main and helper threads even when the cache is not heavily used. An interesting observation is the more than 10-fold performance improvement for both infrastructures when the asynchronous copy / double-buffering optimization is used for data buffers. Despite the fact that

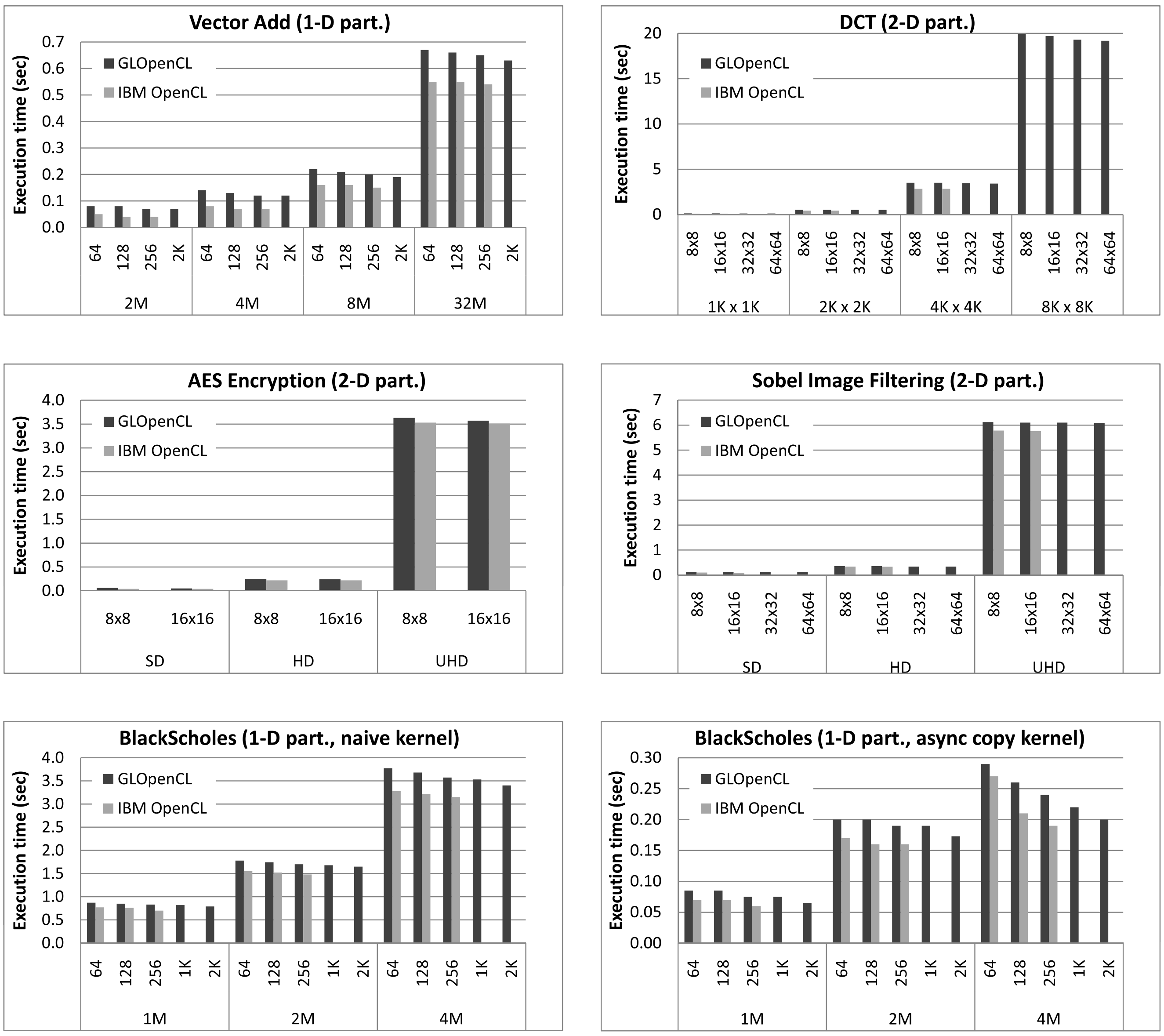


Figure 5: Performance comparison between GLOpenCL and the IBM OpenCL SDK on the Cell. Experiments have been executed for varying data sets (lower x-axis labels) and work-group sizes (upper x-axis labels). The largest data size in each chart is the limit beyond which IBM OpenCL SDK is unable to allocate memory buffers.

OpenCL has been designed as a write-once run-everywhere framework, this result is indicative of the vast performance benefits that can often be attained by carefully mapping the computation to the specific characteristics of the underlying architecture.

Finally, similarly to the case of x86, GLOpenCL was able to work with both larger work-groups and be usable with larger problem sizes than the IBM OpenCL SDK on the Cell blade.

6. CONCLUSIONS

We presented the design, implementation and evaluation of GLOpenCL, a unified compiler and run-time framework for supporting OpenCL applications on a set of parallel systems with diverse architectural characteristics. Our generic framework performed comparably, or often significantly better than customized, architecture specific, vendor OpenCL implementations. Moreover, the experimental evaluation indicated that, even when developers use programming models – such as OpenCL – targeted at efficiently supporting fundamentally different and often heterogeneous computing substrates, careful mapping of applications to the underlying architecture may yield hard to overlook performance benefits.

In future work, we plan to investigate the possibility of reducing the functionality or even removing the software cache in systems with software-controlled memory hierarchies, since the specific module proved to be the main performance bottleneck on the Cell implementation. Although programmers tend to use shared data structures even when they could avoided, automatic code slicing and precomputation of memory accesses – either at the host- or the compute devices-side – may prove a valuable tool in reducing communication overhead.

7. ACKNOWLEDGEMENTS

We would like to thank Barcelona Supercomputing Center for providing us with access to their IBM QS20 Cell blade servers and the anonymous reviewers for their constructive comments. The first author is supported by a grant from 'Bodossaki' public benefit foundation.

8. REFERENCES

[1] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach.* Morgan Kaufmann, 2002.

[2] ATI-AMD. ATI Stream Software Development Kit (SDK) v2.1. `http://developer.amd.com/gpu/ATIStreamSDK/Pages/default.aspx`.

[3] F. Black and M. Scholes. The Pricing of Options and Corporate Liabilities. *The Journal of Political Economy*, 81(3):637–654, 1973.

[4] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[5] Cell B.E. Performance Counter Tool. `http://www.ibm.com/developerworks/power/tutorials/pa-sdk3tool/`.

[6] Clang: A C Language Family Frontend for LLVM. `http://clang.llvm.org/`.

[7] G. Diamos, A. Kerr, and M. Kesavan. Translating GPU Binaries to Tiered SIMD Architectures with Ocelot. Technical report, Georgia Institute of Technology, 2009.

[8] H. Franke and R. Russel. Fuss, Futexes and Furlocks: Fast User-Space Locking in Linux. In *Proceedings of the Otawa Linux Symposium*, pages 85–97, 2002.

[9] Intel Corporation. Intel VTune Performance Analyzer. Document Number 310866-001.

[10] International Business Machines Corporation (IBM). IBM SDK for Multicore Acceleration Version 3.1. `http://www.ibm.com/developerworks/power/cell/`.

[11] International Business Machines Corporation (IBM). OpenCL Development Kit for Linux on Power. `http://www.alphaworks.ibm.com/tech/opencl`.

[12] Khronos OpenCL Working Group and A. Munshi. The OpenCL Specification Version: 1.0 Document Revision: 48, 2009.

[13] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*, pages 75–86, 2004.

[14] J. Lee, S. Seo, C. Kim, J. Kim, P. Chun, Z. Sura, J. Kim, and S. Han. COMIC: A Coherent Shared Memory Interface for Cell BE. In *PACT '08: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 303–314, 2008.

[15] F. Mueller. A Library Implementation of POSIX Threads under Unix. In *Proceedings of the USENIX Conference*, pages 29–41, 1993.

[16] National Institute of Standards and Technology (NIST). Announcing the ADVANCED ENCRYPTION STANDARD (AES). Federal Information Processing Standards Publication 197, November 2001.

[17] NVIDIA. CUDA Technology. `http://www.nvidia.com/object/cuda\_home\_new.html`.

[18] NVIDIA. *CUDA Programming Guide, Version 3.0*, 2010.

[19] NVIDIA. *NVIDIA Compute PTX: Parallel Thread Execution ISA Version 2.0*, 2010.

[20] OpenMP. The OpenMP API. `http://openmp.org/wp/`.

[21] J. P. Perez, P. Bellens, R. M. Badia, and J. Labarta. CellSs: Making it Easier to Program the Cell Broadband Engine Processor. *IBM Journal of Research and Development*, 51(5):593–604, 2007.

[22] I. Sobel and G. Feldman. A 3x3 Isotropic Gradient Operator for Image Processing. Talk presented at the Stanford Artificial Project, 1968.

[23] J. A. Stratton, S. S. Stone, and W.-M. W. Hwu. MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs. In *Languages and Compilers for Parallel Computing: 21th International Workshop, LCPC 2008, Revised Selected Papers*, pages 16–30, 2008.

[24] UPC Consortium. UPC Language Specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Lab, 2005.