International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 1, Number 1, pages 114–130, January 2011

Analyzing Use of OpenCL on the Cell Broadband Engine and a Proposal for OpenCL Extensions

Jens Breitbart, Claudia Fohry
Research Group Programming Languages / Methodologies, University of Kassel
Wilhelmshöher Allee 73, 34121 Kassel, Germany

Email:{jbreitbart, fohry}@uni-kassel.de

Received: July 16, 2010
Revised: October 20, 2010
Accepted: December 14, 2010
Communicated by Akihiro Fujiwara

Abstract

Current processor architectures are diverse and heterogeneous. Examples include multicore chips, GPUs and the Cell Broadband Engine (CBE). The recent Open Compute Language (OpenCL) standard aims at efficiency and portability. This paper explores its efficiency when implemented on the CBE, without using CBE-specific features such as explicit asynchronous memory transfers. We based our experiments on two applications: matrix multiplication, and the client side of the Einstein@Home distributed computing project. Both were programmed in OpenCL, and then translated to the CBE. For matrix multiplication, we deployed different levels of OpenCL performance optimization, and observed that they pay off on the CBE. For Einstein@Home, our translated OpenCL version achieves almost the same speed as a native CBE version. We experimented with two versions of the OpenCL to CBE mapping, in which the PPE component of the CBE does or does not take the role of a compute unit.

Another major contribution of the paper is a proposal for two OpenCL extensions that we analyzed for both CBE and NVIDIA GPUs. First, we suggest an additional memory level in OpenCL, called static local memory. With little programming expense, it can lead to significant speedups such as for reduction a factor of seven on the CBE and about 20% on NVIDIA GPUs. Second, we introduce static work-groups to support user-defined mappings of tasks. Static work-groups may simplify programming and lead to speedups of 35% (CBE) and 100% (GPU) for all-parallel-prefix-sums.

Keywords: OpenCL extensions, Cell Broadband Engine Architecture, GPU, programming model evaluation

1 Introduction

Since a few years, advances in processor speed are chiefly due to adding multiple cores to a single chip. Moreover, a new generation of accelerators, in particular GPUs, FPGAs and the Cell Broadband Engine (CBE), has entered the market and provides a multiple of the processing power of even multicore CPUs. Current architectures are diverse and heterogeneous, including features such as VLIW or SIMD processing, special-purpose cores, complex memory hierarchies, hierarchical arrangement of processors, and asynchronous memory transfers.

To use the high performance potential of these architectures, special programming techniques are required that, at present, are low-level and architecture-specific. With the goal to achieve portability and efficiency, the Open Compute Language (OpenCL) [13] was suggested recently as a standard for programming both multicore CPUs and accelerators. The standard is currently supported by NVIDIA for their GPUs, by AMDs Stream SDK for x86 CPUs and AMD GPUs, as well as by Apple's OS X 10.6. Furthermore, IBM released OpenCL for Power/VMX and CBE systems, but this implementation is still at an early stage of development and requires developers to use features that are uncommon on other hardware. Hence, the question remains open as to which degree OpenCL achieves its goals of efficiency and portability.

This paper comes to a mainly positive answer for data parallel computations on the CBE. Before we can explain our results, some background on OpenCL and CBE is needed, more details are given in Sects. 2 and 3.

OpenCL is an industry standard from the Khronos group, supported by AMD, IBM, Intel, NVIDIA, and others. It models heterogeneous processing platforms, consisting of a host and one or several devices. The devices (e.g. GPUs) act as co-processors to the host. A device comprises multiple compute units (CUs), composed of multiple processing elements (PEs) each. Computation is organized into work-groups (WGs) that contain work-items (WIs) to be run in parallel. WGs are mapped to CUs, and WIs are mapped to processing elements. Memory is organized as a hierarchy, with a major bottleneck between the local memories of CUs and global memory. To speed up memory accesses, techniques such as switching between multiple WIs mapped to the same PE, asynchronous memory accesses, and prefetching are used.

The CBE is the first incarnation of the Cell Broadband Engine Architecture (CBEA), designed by a collaboration between Sony, Toshiba and IBM. It is a single-chip multiprocessor that comprises one general-purpose CPU, called PowerPC Processing Element (PPE), and up to eight special-purpose compute engines, called Synergistic Processor Elements (SPEs). Memory is divided into local stores of SPEs and main memory, there is no hardware cache at the SPEs. Memory transfers must be managed by the programmer, who can interleave transfers and computation for efficiency, or use software caches in the local store.

Our concept of mapping OpenCL to the CBE is quite obvious: We identify the host with the PPE, and each CU with an SPE. The SPEs sequentially process the WIs that OpenCL would run in parallel on the PEs. Global memory is identified with main memory, and local memory is identified with local store, the complete mapping is more complicated and explained in Sect. 3. Additionally, we investigated a heterogeneous variant, in which the PPE takes the role of both the host and an additional CU. While a compiler will be clearly needed for practical use, the experimental study of this paper relies on a manual implementation of the mapping with C macros.

The question whether the mappings lead to an efficient implementation of OpenCL on the CBE has different aspects: First, OpenCL programs translated to the CBE should have no significant performance loss as compared to native CBE programs. We studied this question with a real application, the client side of the distributed gravitational physics simulation Einstein@Home. We found that the OpenCL version runs at almost the same speed as a native version developed by the same author, with about the same importance having been attached to optimization. In the heterogeneous variant, inclusion of the PPE has a similar effect as adding an SPE.

Second, OpenCL exposes many hardware features that are found in CBE in a somewhat different form. Variants of an OpenCL program may use these features to a different degree, depending on their level of optimization. The faster a variant appears to be at the OpenCL level, the faster it should run on the architecture. This relation obviously holds for NVIDIA GPUs, as their native CUDA language is very close to OpenCL. For CBE, we studied the relation with the example of matrix multiplication at different optimization levels. We found that ordering memory accesses for locality and explicitly managing local memory pays off, whereas asynchronous memory transfers do not.

Third, for all major performance optimizations applicable on the CBE, it should be possible to express them at OpenCL level. While the Einstein@Home result suggests this to be basically the case, we found two optimizations that can not yet be expressed: *static local memory* and *static work-groups*. In fact, these optimizations are not restricted to CBE, and we suggest to extend OpenCL accordingly. We implemented the optimizations on both CBE and NVIDIA GPUs.

Static local memory is an additional memory level in-between local memory and global memory.

While local memory belongs to a WG, occasionally data are reused between WGs. If they run on the same CU, it is sufficient to store the data once. Static local memory is different from global memory in that it can not be accessed by the host and is typically faster, even faster than global memory cache. We suggest to extend OpenCL by an additional address space modifier, and provide two examples for the effectiveness of this easy-to-use technique: constant tables and reduction. For reduction, speedups up to a factor of seven for CBE and 20% for GPUs is achieved.

Static WGs are similar to CPU threads in the sense that they perform not one but many tasks and allow users to specify any mapping of tasks to static WGs. Knowing the mapping, one can specify communication and synchronization between tasks, as they are possible between SPEs on the CBE. OpenCL, in contrast, applies a fixed, user-intransparent mapping of WGs to multiprocessors and disallows communication and synchronization between them. We demonstrate the usefulness of static WGs with all-parallel-prefix-sums, also known as scan, and show that static WGs both simplify the algorithm and lead to a higher performance. Our improved scan implementation uses task interleaving and global barriers to achieve a performance increase of about 35% on the CBE and 100% on NVIDIA GPUs. Simplicity is mainly due to no longer requiring recursive kernel calls.

The paper is organized as follows. First, Sects. 2 and 3 provide background on the OpenCL standard, and the GPU and CBE architectures, respectively. Then, Sect. 4 describes our mapping from OpenCL to the CBE, including the heterogeneous variant. Experiments with different optimization levels in OpenCL and their performance impact on the CBE are reported on in Sect. 5. Section 6 introduces the Einstein@Home application and compares implementations with and without OpenCL. Next, Sects. 7 and 8 are devoted to the static local memory and static WGs extensions, respectively. These extensions are not only useful for the CBE, but also for GPUs, as is shown in Sect. 9. Thereafter, Sect. 10 discusses implications of the extensions for the OpenCL standard. The paper finishes with an overview of related work and conclusions, in Sects. 11 and 12, respectively.

2 OpenCL

OpenCL is a framework consisting of a language, API, libraries and a runtime system. The first version of the standard was released in May 2009. Figure 1 depicts the platform model, as it has already been introduced in Sect. 1 Note that the static local memory component depicted in the figure is not part of the standard, but an extension proposed in this paper.

An application is run on the host, which sends instructions, bundled in special functions called *kernels*, to the device for execution. The OpenCL standard defines a data parallel and a task parallel programming model. The former is the primary model, and we restrict consideration to this one. In the data parallel model, the device runs multiple instances of the kernel, the WIs, in parallel on different data. While all WIs run the same kernel, they may perform different instructions at a time. WIs can be arranged in WGs. The number of WIs in a WG is limited by the hardware. A typical limit is e. g. 512 WIs per WG on NVIDIA hardware. OpenCL defines indexing schemes by which a WI can be uniquely identified through either a global ID, or a work-group ID together with a local ID.

The WGs are assigned to CUs, where the WIs of each group are run in parallel on the PEs. Typically, multiple WGs are assigned to the same CU, and multiple WIs are assigned to a PE. Conceptually, both are executed in sequence, but an implementation may use the excess parallelism for hiding memory latency (by switching between WGs or WIs, respectively).

How many WGs a CU can run in parallel depends on the hardware and the memory requirements. If, for instance, a WG requires 64 kB of local memory and the hardware provides 128 kB on a CU, two WGs can be executed in parallel. If there are more groups than the hardware can accommodate, an external queue of groups is maintained. The order in which they are scheduled is undefined, and users can therefore not specify synchronization such as having one WG wait for the completion of another or having all WGs waiting at a global barrier (since it can not be guaranteed that all WGs are active at the same time). Thus, OpenCL only supports barrier synchronization of WIs within a WG, as well as a fence operation to achieve memory consistency within a WG and between different WGs.

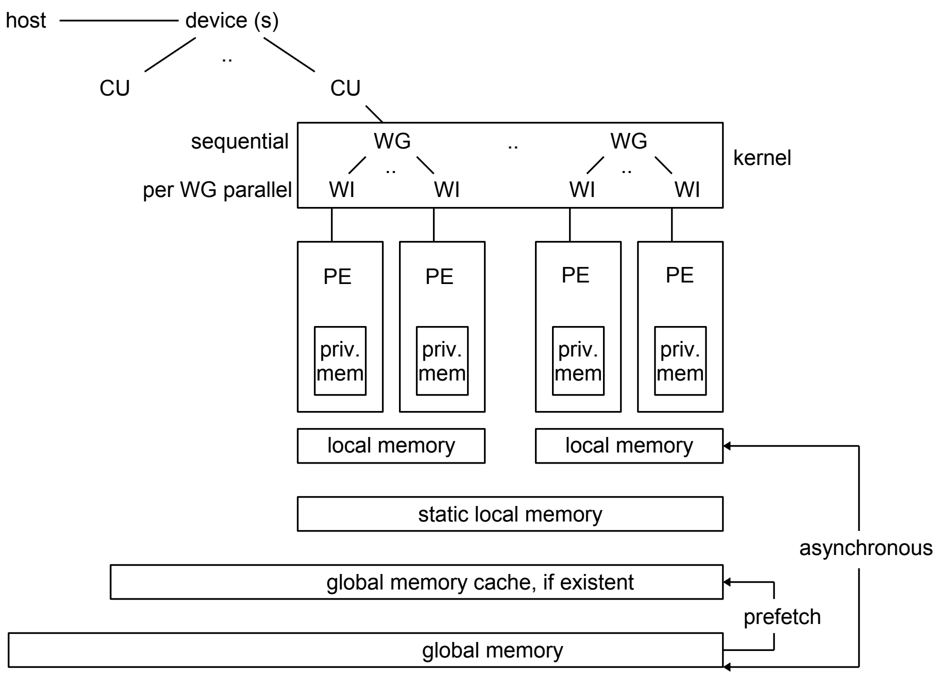

Figure 1: Extended OpenCL model

OpenCL also defines a programming language for writing kernels, which is an extension of C. Kernels are executed within their own memory domain and may not directly access host main memory. Kernel memory is divided into four distinct regions:

- *Global memory*, a kind of "device main memory", can be accessed by all WIs and the host in reads/writes.
- *Constant memory* is similar to global memory, except being read-only by the device.
- *Local memory* is read/write memory local to a WG, and is shared by all WIs of this group.
- *Private memory* is local to each WI.

The mapping of memory regions to the hardware is implementation-defined, but typically local memory is faster than global memory. The OpenCL programming language defines type qualifiers to specify in which memory region a variable is stored or to which region a pointer points to.

As a kernel can neither access host main memory nor dynamically allocate memory, all memory management must be done by the host. The OpenCL API provides functions to allocate linear memory blocks in global or constant memory, as well as to copy data to or from these blocks. In most cases, the host copies all input data to the kernel memory domain prior to a kernel launch, and the result back afterwards.

Memory access latency is arguably the most important issue on modern processor architectures. On NVIDIA GPUs, for example, accessing global memory costs up to 100 times as much as executing a compute instruction for 32 WIs. Due to this high discrepancy, program performance is often dominated by memory access time, a problem referred to as the *memory wall.* The problem is somewhat reduced if the hardware caches global memory data on the device, which OpenCL allows but does not require. Moreover, OpenCL supports several techniques for hiding memory latency:

- use of excess parallelism as described above
- asynchronous memory accesses, i.e., explicitly overlapping calculations and memory transfers

- prefetching based on programmer hints as to which global memory data are accessed next and should be cached (if cache exists)

3 Overview of GPU and CBE architectures

There is a close correspondence between OpenCL and GPU architectures, which are many-core chips with hundreds of slow in-order cores organized in tiles. GPU architectures support fast communication and synchronization within a tile, but communication between different tiles must go through slow off-chip memory. On NVIDIA hardware, tiles are called streaming multiprocessors and correspond to the CUs of OpenCL, whereas the cores correspond to PEs. NVIDIA's GPU programming model CUDA is almost identical to the OpenCL model, except for different names. In CUDA, WIs are called threads, and WGs are called threadblocks. To hide memory latency, PEs are oversaturated with threads. The number of threads depends on factors such as the number of registers available [5].

For our experiments, we used a GeForce GTX 280, which has 30 CUs with 8 PEs each and a theoretical peak performance of about 1 TFlop single precision.

The implementation of OpenCL on GPUs is obvious. Global memory is mapped to off-chip graphics memory, and has an access time of hundreds of cycles. On the latest hardware generation, this memory is cached, however the hardware we used does not provide a cache. Local memory is mapped to on-chip memory called shared memory, and private memory is mapped to registers.

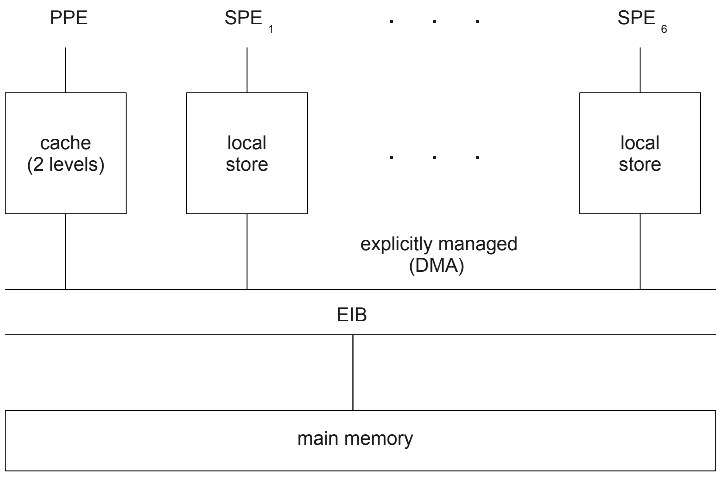

Figure 2: Cell Broadband Engine overview

The Cell Broadband Engine (and its corresponding architecture CBEA) is a completely redesigned processor that was originally intended to be used in gaming consoles, but is now used in areas such as high-performance computing as well. As depicted in Fig. 2, the CBE is heterogeneous and provides two kinds of cores on the same chip: one PPE and up to eight SPEs. The PPE is a relatively slow in-order CPU based on the Power4 architecture, whereas most of the processing power is provided by the SPEs. These are SIMD cores with 256 KB of fast on-chip memory called *local store*. All compute elements are connected through a high-speed interconnect bus (EIB). The original CBE with 3.2 GHz had a peak performance of over 200 GFLOPS for single-precision, and 15 GFLOPS for double-precision arithmetic. The current CBEA release, called PowerXCell, improves double-precision performance to 100 GFLOPS. We used a Playstation 3 with an original CBE with six SPEs for the work presented in the following sections.

The CBE is typically programmed in C, processor-specific functionalities are accessed through function calls. At assembler level, the architecture also supports SIMD instructions, which refer to 128 bits, i.e. four words.

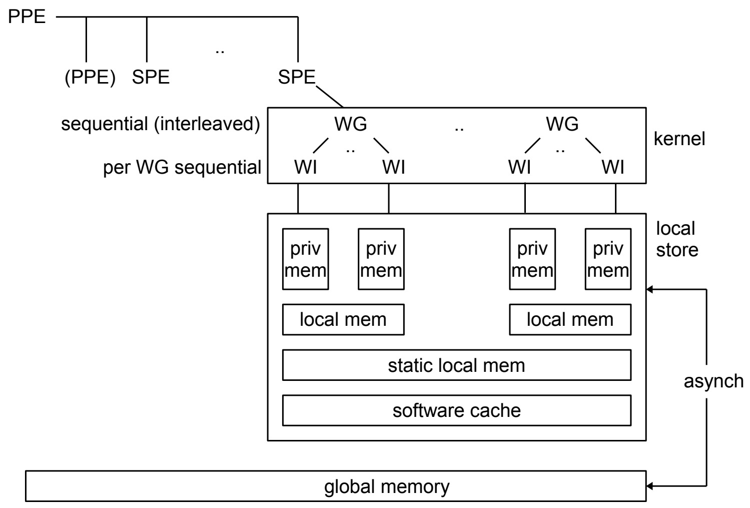

Figure 3: Mapping of extended OpenCL model to CBE

The CBE tackles the memory wall in a different way from other current processors. To bring more compute elements onto the chip, it omits caches for the SPEs, and instead focuses on interleaving computation and memory accesses. From this and other techniques, the CBE achieves a “real world” application performance of nearly 100% of its theoretical peak performance [8].

A software challenge introduced by the CBE design is the need for explicit management of memory transfers by the programmer. PPE and SPEs are equipped with a DMA engine. While the PPE can access main memory directly, an SPE only has direct access to its own local store. Between main memory and local store, data must be moved by explicit DMA transfers, and therefore data must be partitioned to fit the limited size of this store. As an alternative, some current compilers support *software caching*, i.e., part of the local store is maintained as a conventional cache managed automatically. Software caching comes at the price of not being able to overlap calculations and memory transfers.

4 OpenCL implementation for the CBEA

Both OpenCL and CBE programs are based on C, extended by keywords and/or function calls. Ideally, an implementation would automatically compile an OpenCL program into a C program with function calls for CBE. For the purposes of this experimental study, we chose a simpler approach. We use a semi-automatic translation scheme based on C macros, which requires slight changes of the kernel syntax. Moreover, at the host side, function calls resemble the inofficial OpenCL C++ bindings available on the OpenCL homepage [11]. The bindings were chosen since our programs partially rely on the CuPP library [2], a framework of easier-to-use interfaces for reoccurring tasks.

Our implementation is illustrated in Fig. 3. It maps the host to the PPE, and each CU to an SPE. A heterogeneous variant, in which the PPE additionally takes the role of another CU, is discussed at the end of this section. We experimented with one OpenCL device only. In the base variant, which is referred to except where otherwise noted, WGs are statically mapped to SPEs, i.e., each SPE processes about the same number of WGs. The groups are processed sequentially, as are the WIs in the groups. When a barrier or memory fence is encountered, all WIs are run sequentially up to this point, and then the program continues behind it.

This behavior is implemented by our macro-based translation scheme. At the beginning/end of a kernel function, macros called `START`/`END` must be called. They start and end loops that are used to sequentially execute the WIs. A barrier or memory fence is implemented as a joined `END` and `START` macro, which first ends and then restarts the loops.

Table 1: Overview of program versions

α	naive matrix multiplication
β	blocked matrix multiplication
γ	blocked matrix multiplication with preloading
δ	blocked matrix multiplication with preloading by `vloadn`, bypassing cache
ϵ	like δ, but with asynchronous memory transfers
ζ	assembler version using SIMD units

Table 2: Running time in seconds for multiplying two $2048 * 2048$ matrices

Hardware	PPE	GTX	6 SPEs					
Version			α	β	γ	δ	ϵ	ζ
Time (s)	1682	0.9	767	37.3	15.9	2.3	2.5	0.2

Global and constant memories are implemented as software-cached main memory, with the only difference that constant memory is declared as `const`. Local and private memories are both mapped to the local store of the SPEs. Variables in local memory are stored once for each WG, and variables in private memory are stored once for each WI.

The implementation conforms to the relaxed memory consistency model of OpenCL, which requires local memory to be consistent at a barrier or fence only, and does not require global memory to be consistent across WGs.

As an aside, our mapping easily allows for atomic operations, which are an optional extension of the OpenCL standard. Local memory accesses are always atomic since at no point multiple WIs of the same group are executed. Global memory atomics correspond to the atomic operations provided by IBMs Cell Broadband Engine SDK.

In addition to the base variant described above, we studied a heterogeneous variant of our mapping scheme, in which the PPE takes the role of both the host and a CU to increase the available processing power. The architecture of the PPE differs from that of an SPE in that it has no local store and instead uses a traditional cache hierarchy. Therefore, local and private memories are mapped to main memory. As we were not able to capture the performance difference to an SPE by a constant factor, we dynamically scheduled WGs to SPEs, using a similar scheme as the guided work distribution of OpenMP [16]: First large chunks of work are assigned, and then this size is decreased. As will be shown in Sect. 6, inclusion of the PPE had a similar effect as adding an SPE.

5 Evaluation of the memory system

This section investigates to which degree performance optimizations at the OpenCL level pay off on the architecture. For their importance, emphasis is given to optimizations of memory accesses. The following evaluation is based on the well-known matrix multiplication example: Given square matrices A, B, C of size n^2, calculate $A * B = C$. The following program versions have in common that a WI corresponds to the calculation of one element of C. Obviously, all WIs can run in parallel. An overview of the different versions is given in Table. 1.

Table 2 shows running times of the program versions, all performed at single precision. It also includes two reference times: naive matrix multiplication on the PPE (corresponding to version α described below), and optimized matrix multiplication as provided by NVIDIA for CUDA at a GeForce GTX 280.

Version α is a naive matrix multiplication, i.e., each matrix element $C[i][j] = \sum_k A[i][k] * B[k][j]$ is calculated independently from other $C[i][j]$. The algorithm is well-known to have low memory locality, since successive accesses to elements of A, B are too far away to keep data in cache. For instance, $A[0][0]$ is required for the calculations of $C[0][j]$ ($j = 0 \ldots n$), but as WIs are executed one after another, $A[0][0]$ needs to be reloaded for each j. The low memory locality is both visible at OpenCL level and reflected in the measured running time in Table 2.

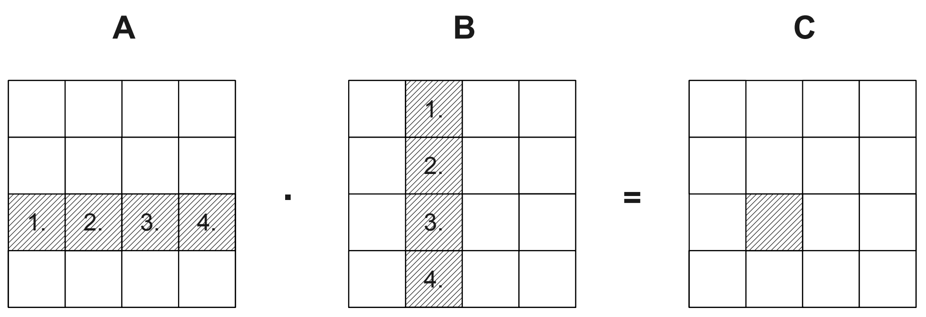

Figure 4: Blocked matrix multiplication overview

Version β uses a well-known blockwise matrix multiplication algorithm, which is described for instance in [12] for CUDA. The algorithm exploits memory locality by partitioning A, B, C into square submatrices. Calculation of each subblock C_{sub} of C proceeds in a coordinated way (see Fig. 4). First, a subblock of A is multiplied with the corresponding subblock of B, i.e., $C[i][j] = \sum_k A[i][k] * B[k][j]$ is calculated for all $C[i][j]$ from C_{sub} for only a range of k. Then, computation proceeds with the next subblocks of A, B, and so on. The scheme exhibits memory locality as successive accesses to the same elements of A, B are concentrated in time. In OpenCL, the scheme is implemented by mapping calculation of each C_{sub} to a WG, and separating submatrix multiplications by a barrier. Version β is much faster than version α at both the OpenCL and CBE levels, since data can be stored in global memory cache or software cache, respectively. In our experiments, the best performance is achieved with a WG size of $128 * 128$.

Despite the usefulness of caching, in version β interference of cache lines cannot be completely precluded, and cache maintenance induces some overhead. Therefore, we developed version γ, which uses local memory to explicitly store data of subblocks. In this version, subblocks of A, B are explicitly loaded to local memory prior to each submatrix multiplication by normal (cached) global memory accesses. Again, the optimization pays off for both OpenCL and CBE.

Version δ resembles version γ, but deploys a special function of OpenCL, called `vloadn()` to load chunks of data from global memory to local memory. We implemented it with a DMA transfer and a barrier waiting for its completion. Unlike version γ, it bypasses the cache, leading to a large performance increase. This result indicates that the performance penalty of global memory cache is significant and that on the CBE – even though global memory is cached – explicit management of local memory may pay off.

None of the versions explained before tries to hide memory latency. The purpose of version ϵ is studying the performance impact of overlapping memory transfers and calculations. This type of parallelism can be expressed explicitly in OpenCL, with asynchronous memory transfers. These, however, are hard to program and are hardly seen on other architectures. Relying on this feature would most likely result in poor performance on other architectures, for example reference [14] suggests not to use it for Power/VMX CPUs. Since OpenCL is an abstract model that requires translation, the same type of parallelism can alternatively be exploited by the compiler (or macro-based translation scheme), which makes programming easier. For our matrix multiplication example, we chose this option, based on version δ.

In our implementation, up to four WGs are processed at a time. When one of them is waiting for global memory data, it is suspended and another WG takes over. Only four WGs can be in progress, as otherwise their memory footprint (e.g. private memory data of all WIs) would be too large. When the forth group starts accessing memory, the first suspended group is reactivated, and waits for completion of its memory transfer before continuing. Because of the large memory footprint, we additionally had to change WG size. In all versions prior to ϵ, it was 128^2, but now it is 64^2. Having more WGs increases the overhead for memory transfers, as it increases the number of DMA transfers, and reduces the performance of the DMA engine. For this reason, version ϵ is

Table 3: Running time in seconds for a short Einstein@Home testcase

	Number of SPEs					
	1	2	3	4	5	6
Native	172	111	91	81	76	73
Local memory	189	122	99	88	83	79
Static local memory	169	111	90	83	78	75
Static local memory with PPE	116	95	85	79	75	72

slower than version δ. To isolate the effect, we re-run version δ with WG size 64^2. In this case, version δ required 2.7 seconds, and was thus slower than version ϵ.

Overlapping memory transfers and calculations leads to only small speedups for two reasons: First, the amount of local memory required is rather high, which allows for only a small number of concurrently processed WGs and does not permit a significant amount of latency hiding. Second, as version ζ below indicates, the program is compute-bound and not memory-bound.

An OpenCL feature not exploited by our mapping is parallelism among WIs of a group. While the CBE does not support the same level of multiprocessor parallelism as found in GPUs, its SIMD units may still be deployed. OpenCL programmers typically strive for SIMD parallelism, since CUDA deploys this type of parallelism as well. Therefore, a compiler for the CBE can often bundle instructions of four WIs to generate SIMD instructions. Version ζ uses assembler code written by Hackenberg [8] that exploits the SIMD units to improve the performance of the calculations. The large improvement shown in Table 2 suggests that mapping parallel WIs to SIMD units is worthwhile. We leave this optimization for future work, as we do with the cache prefetch instruction.

6 Performance comparison to native application

This section reports on our experiments with the client side of the Einstein@Home distributed computing project [7]. The application issues a brute force search for gravitational waves in a large data set collected by detectors all over the world. Most of the calculations are double precision. The application has already been implemented previously by one of the authors for both CUDA and the CBE [4]. It fits the data parallel model of OpenCL quite well, as there are "groups" of independent calculations whose results must be reduced [1]. We reimplemented the CUDA version in OpenCL, mapping calculation groups to WGs, and individual calculations to WIs.

Since the size of calculation groups varies, but OpenCL requires WG sizes to be constant throughout a kernel call, the maximum size is computed prior to each call and WGs with less calculations have idle WIs. Input data are not moved to local memory, since data accesses cannot be planned in advance. Instead, all reads correspond to cached global memory accesses. The reduction is implemented with local memory atomic operations.

The first two lines of Table 3 compare the performance of our OpenCL-based implementation to an optimized version of the native program mentioned above. The native version is slightly faster than the OpenCL version, as we cannot express all optimizations of the original client with OpenCL. This problem is detailed in the next section. Nevertheless, the outcome suggests that the use of OpenCL hardly hurts the performance as compared to a native CBE implementation.

With the Einstein@Home application, we moreover evaluated the heterogeneous mapping scheme described in Sect. 4. As a basis for comparison, we used the improved OpenCL variant that will be explained in Sect. 7. As the last two lines of Table 3 show, inclusion of the PPE has a similar effect as adding an SPE. The small gain achieved on six SPEs is due to the overall running time being dominated by the sequential fraction of the program. The dynamic scheduling scheme used in the heterogeneous variant induces little overhead: static scheduling optimized for Einstein@Home improved the performance by only 0.7% on six SPEs.

Table 4: Running time in seconds for a reduction of 2^{25} elements with and without static local memory

Hardware	PPE	Number of SPEs					
		1	2	3	4	5	6
Local memory	10.1	21.9	8.0	3.6	2.6	2.0	1.8
Static local memory		17.5	5.3	1.0	0.4	0.28	0.25

7 Static local memory

Comparing the native and OpenCL Einstein@Home program versions, we observed that one optimization from the native version was not possible to appropriately express in OpenCL, namely use of a lookup table in fast memory to calculate sine and cosine values faster. In the native version, the table is hold in local store and initialized once for every SPE. In OpenCL, local memory belongs to a WG, and thus the table must be reinitialized for each WG.

To prevent this problem, we propose a new level of memory that we call *static local memory*. This memory is local not to a WG but to a CU, and can be accessed by all WIs scheduled there. Static local memory must be initialized by the first WG, and the final result must be taken out by the last. Therefore the programmer must be able to identify these groups. Accesses to static local memory are faster than accesses to global memory even if the data are cached, since cache maintenance is expensive as has been discussed in Sect. 5. Static local memory speeds up the Einstein@Home application, as can be seen in the third (vs. second) line of Table 3.

Besides constant tables and other forms of reusing values after having been computed once, static local memory is useful for combining values computed by different WGs. This usage pattern is manifested by reduction, which is an important building block of many parallel programs. We experimentally investigated a variant in which one WI is used for every data item to be reduced. Each WG first combines its data in a local memory location. Then, the standard OpenCL version writes the results directly to a global memory location by an atomic operation (e.g. add). The static local memory version, in contrast, adds the results in a static local memory location, and only the last WG of each CU accesses global memory. Table 4 shows performance numbers for adding 2^{25} values. On six SPEs, a speedup of seven is achieved. For the high importance of reductions, we consider this improvement alone worth adding a new memory level to OpenCL, especially as it is easy to use. Moreover, the extension is useful for GPUs, as will be discussed in Sect. 9.

Beyond the given examples, reuse of data within a CU is useful for data decomposition-based algorithms. Here, WGs working on neighbored data may interchange boundary values through static local memory. As a simple example, we consider all-parallel-prefix-sums, also called scan, where the total sum of the ith block of array elements must be communicated to the $i+1$-st block. To make static local memory usable for scan and other data decomposition-based algorithms, the WGs that interchange data must be co-located on the same CU, i.e., the user must be able to control the mapping of WGs to CUs. Unfortunately, OpenCL does not support that, and so in Sect. 8 we suggest an extension called static work-groups.

8 Static work-groups

A WG can be considered as a unit that runs a certain number of WIs and thus accomplishes a certain task. Static WGs were first introduced by one of the authors in [3], and are a kind of container to which these tasks can be mapped. A static WG is assigned to a CU, where it runs the tasks in a user-defined sequential order without preemption. Thus, static WGs are similar in concept to CPU or SPE threads. We propose static WGs as an extension to OpenCL that serves a five-fold purpose:

- It allows a user to specify which WGs should run on the same CU, thus making the static local memory concept from Sect. 7 usable for data decomposition-based applications.

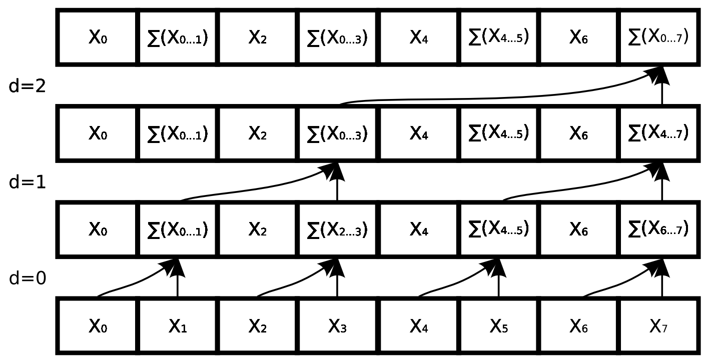

Figure 5: Up-sweep phase

- It allows a user to specify ordering constraints among the WGs mapped to the same static WG.
- It may improve load balancing among CUs (see below).
- It enables the implementation of a global barrier or other forms of synchronization between WGs running on different CUs.
- It can be used to implement static local memory on GPUs, as will be explained in Sect. 9.

Static WGs are primarily a programming pattern. A user programs a static WG by writing a single kernel, therein coding the computations of the original WIs as well as their order of execution and mapping to a particular static WG. This way, different scheduling schemes can be expressed. Static scheduling, for example, can be implemented with a loop, whose indices depend on the (static) WG index. Dynamic scheduling may be implemented with an atomically incremented counter. An appropriate scheduling scheme may balance the load among CUs, whereas the OpenCL mapping is only efficient if all tasks are about the same size.

Beyond being a programming pattern, static WGs require support from OpenCL. To make full use of the concept, a user must know the number of static WGs that can be run on the device at the same time. No matter whether there are one or several static WGs per CU, it is important that there must not be WGs in the external queue and there should be enough static WGs to keep all CUs busy. Only if all static WGs are active at the same time, a global barrier and other forms of synchronization between different WGs can be implemented.

Thus, we suggest to extend OpenCL by a query function to determine the number of static WGs. Since OpenCL does not currently provide such a function, our implementation uses hardware-dependent code to determine the number, as a work-around.

In the following, we demonstrate the usefulness of static WGs with the example of scan, which is an important parallel building block used in a wide variety of algorithms. Scan takes as input an array $[x_0, x_1, ..., x_{n-1}]$, a binary associative operator $+$, and the identity I, and returns $[I, x_0, (x_0 + x_1), ..., (x_0 + x_1 + ...x_{n-2})]$.

For our OpenCL implementation, we took the CUDA implementation by Harris et al. [9] as a basis. It subdivides the whole array into blocks and uses WGs to perform a local scan in every block. The local scan is split into two phases: first the up-sweep phase and afterwards the down-sweep phase. The array access pattern in both phases is based on a balanced tree. During the up-sweep phase, the tree is traversed from the leaves to the root computing partial sums. In the down-sweep phase, the tree is traversed from the root to the leaves, using the partial sums to compute the missing elements. Figures 5 and 6 show the algorithm in detail. After the up-sweep phase, the block sums of every WG are written to an auxiliary array (SUM). Then, in a second kernel, SUM is scanned, and in a third kernel, the results of the scan are used to update the original array, so it contains the final results. We identify two major performance problems:

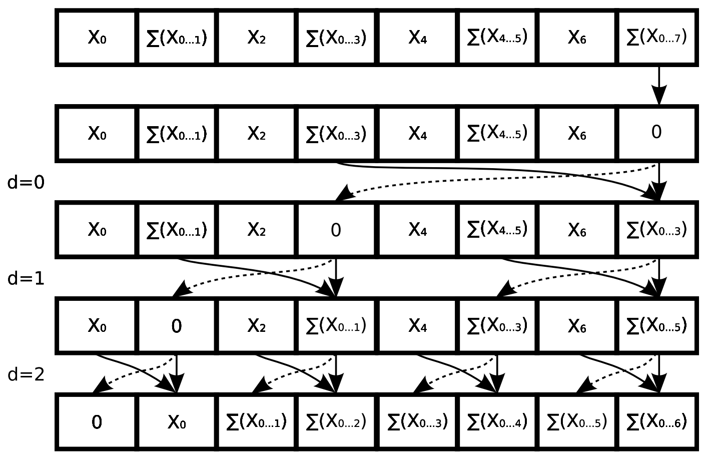

Figure 6: Down-sweep phase

Table 5: Running time in seconds for a scan of 2^{23} elements

	Number of SPEs					
	1	2	3	4	5	6
OpenCL scan	2.62	1.04	0.69	0.52	0.42	0.35
V1	2.18	0.94	0.63	0.47	0.38	0.32
Scan$^+$	1.80	0.76	0.51	0.38	0.31	0.26

1. The algorithm requires three kernel calls in the best case. If more than 1024 WGs are used, the algorithm that scans the SUM array must either be recursively applied, or a sequential scan must be performed. The first choice complicates implementation by a great deal, and every kernel call causes overhead on the CPU. The second choice will not perform well for hardware with multiple PEs in a CU.

2. Except $d = 0$ in the up-sweep phase and $d = maxLevel$ in the down-sweep phase, only a fraction of WIs are active.

On the CBE, we implemented the algorithm as described above, even though there is currently no benefit in using multiple WIs (since we do not utilize the SIMD units of the hardware). For this reason and for simplicity, we implemented the second (sequential) option for scanning the SUM array.

We suggest a solution for both problems outlined above, by using static WGs. Figure 7 shows our solution to the first problem. As you can see, the whole computation is done with one kernel call. We subdivide the input array into as many blocks as we use static WGs, and have every static WGs scan its part. Therefore, it performs the tasks of multiple of the former WGs in sequence, passing each task's final sum on to the next task through a (static) local memory location.

After the scan is complete, every static WG writes its result out to a SUM array. Note that the number of static WGs does not depend on the input size of the array, which reduces the size of the SUM array. The SUM array is scanned by the static WG that finishes its part of the input array last. It is identified by atomically decrementing a shared counter in global memory. The start value of the counter is set by the PPE to $k + 1$, where k denotes the number of static WGs used. The static WG that decrements the counter to 1 performs the scan, and then reduces the value again. When the counter reaches 0, all but the first static WG continue to update the final array. We refer to this version, which solves problem 1, simply as *V1*. Table 5 shows its performance in comparison to our initial implementation and a not-yet discussed final version solving both problems. Note that V1 outperforms our initial version.

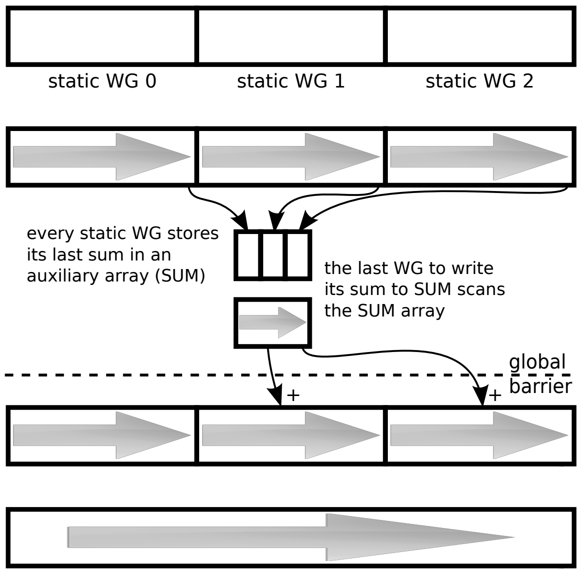
static WG 0
static WG 1
static WG 2
every static WG stores its last sum in an auxiliary array (SUM)
the last WG to write its sum to SUM scans the SUM array
global barrier
+
+

Figure 7: Global overview of V1 and Scan$^+$.

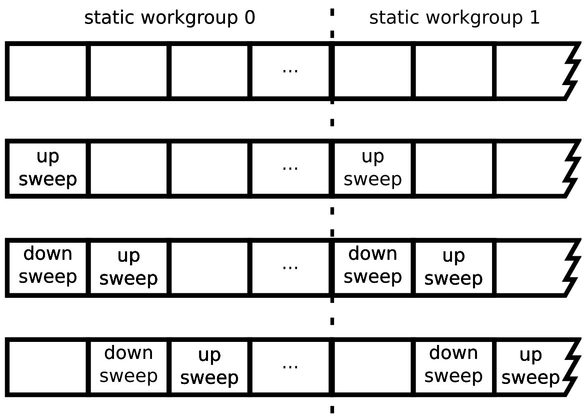

Figure 8: Overlapping of scan phases.

Problem 2 is addressed by executing both phases interleaved. We refer to this version as Scan$^+$. Figure 8 shows the interleaving in detail. The approach leads to a higher PE utilization in every step of the up-sweep and down-sweep phases. In the first step ($d = 0$, see Figs. 5 and 6), one WI has to execute an instruction of both the up- and down-sweep phases.

Compared to a native OpenCL implementation performing Harris' algorithm, the overall performance was increased by about 45% (1 SPE) to 35% (6 SPEs) on the CBE. The rather small performance increase results from the fact that we currently do not use the SIMD units, so the increased WI parallelism gives no benefit.

9 Impact of the extensions for GPUs

Since we cannot alter the existing OpenCL system from NVIDIA, we were not able to directly implement static local memory on GPUs. Instead, we based our implementation on static WGs, i.e., we deployed the programming pattern described in Sect. 8 to map a block of WGs to a static WG, and let local memory variables of the static WG take the role of static local memory variables of the original WGs. This approach has several drawbacks:

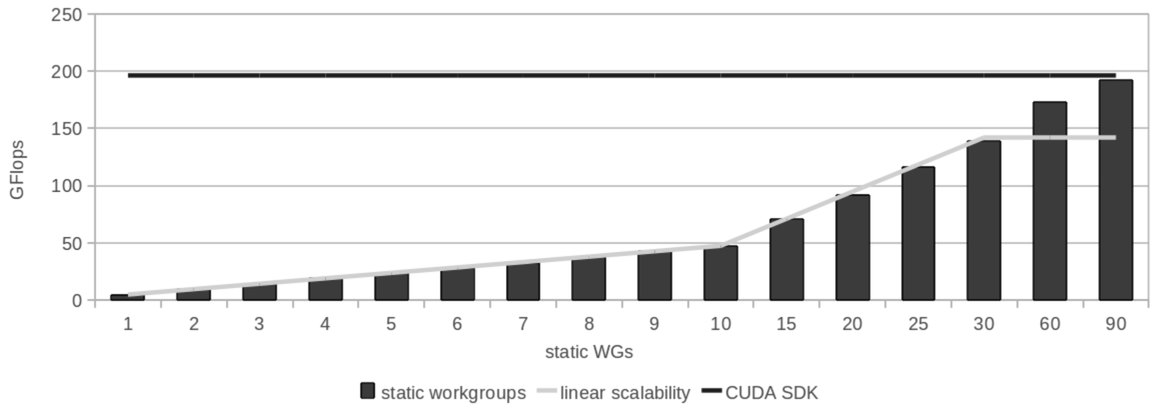

Figure 9: GTX 280 Matrix multiplication performance (matrix sizes 4800*7200*4800)

Table 6: Performance in GB/s for a reduction of N elements on a GTX 280

N	2^{17}	2^{19}	2^{21}	2^{22}	2^{25}
Static local memory	10.1	33.1	75.5	112.5	126.5
Local memory	2.2	8.7	34.0	109.1	105.3
Two kernels	2.1	8.5	33.0	106.4	104.7

- OpenCL lacks support for static WGs, as explained before.
- Static WGs increase programming complexity, as even static scheduling requires an additional loop.
- More complex kernels may increase the register pressure, which can decrease the overall performance.

Thus, direct support of OpenCL for static local memory is desirable and will be discussed in Sect. 10. To estimate the overhead of static WGs, we re-run the blocked matrix multiplication algorithm (version β) of Sect. 5, which is also used in NVIDIA's CUDA SDK sample. As Figure 9 shows, the static WG version scales linearly up to the 30 CUs, and beyond that further profits from oversaturation. The overall performance is slightly below that of the SDK implementation, due to the overhead of static WGs, but this overhead is rather small.

Using the implementation described, we evaluated the performance of reduction on a GTX 280. As Table 6 shows, static local memory improves the performance on this GPU, as well. The algorithms in the first two lines are the same as described in Sect. 7. The version with two kernels launches a second kernel to reduce the results of all WGs inside a single WG, i.e., the second kernel utilizes one CU only.

Figure 10 depicts the GPU results for the scan algorithms from Sect. 8. It can be seen that Scan$^+$outperforms Apple's OpenCL implementation of Harris' scan algorithm in almost all cases, indicating that static WGs are useful on the GPU, as well. All three versions use shared-memory padding to prevent bank conflicts, and unrolled loops to increase performance. Our implementations are mostly limited by register pressure. If hardware would provide us with more registers, we could e. g. implement global memory prefetching as it has been suggested by [5] to further increase performance.

10 Implications to the OpenCL specification

Adding static local memory to OpenCL requires modifications of both the programming language and the runtime system. One way to modify the programming language is introducing a `static`

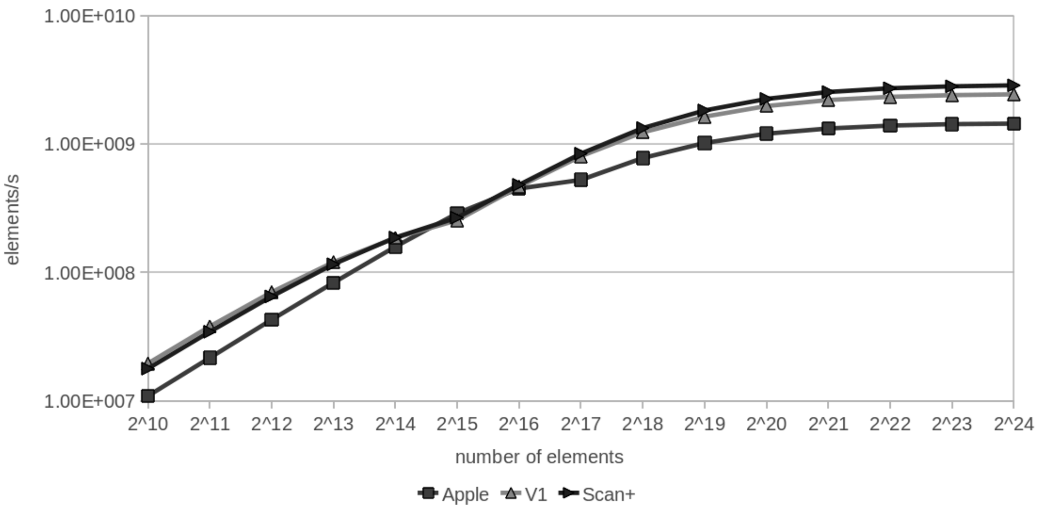

Figure 10: GTX 280 scan performance

qualifier that can be added to variables marked as `__local`, including support of pointers. To use these variables, all built-in functions, e. g. atomic and vector load/store functions, must be modified. Furthermore, support of dynamic memory allocation requires to modify the runtime system and in particular the function `clSetKernelArg()`, so that it is able to not only allocate local memory, but also static local memory. This could for instance be achieved by adding a new parameter to this function that specifies whether the requested memory is static or not. These modifications should be rather simple if WG scheduling and control of local memory are implemented in software. If they are implemented in hardware, it will probably be necessary to implement a form of static WGs, in a similar way as explained for the GPU implementation in Sect. 9.

Support of static WGs only requires to add a query function for the number of WGs that can be run in parallel with a given kernel, plus the guarantee that the WGs will not be interrupted.

11 Related work

Our CBE implementation resembles the one presented by IBM [14], but has some advantages. The IBM implementation is available for both Power/VMX and CBE, and requires a different programming style for the two systems. For example, on the CBE the use of asynchronous memory transfers is highly recommended, but should be avoided for Power/VMX. Our results suggest that, even on the CBE, asynchronous memory transfers are not necessarily required for performance, which improves portability of OpenCL programs to different architectures.

Another project called *OpenCL PS3* [15] follows the goal to make an OpenCL implementation available at the PS3, but at the time of this writing there is not yet a release available. Gpuocelot [6] is a project that uses the CUDA assembler language PTX as a platform to generate code for different hardware architectures. At present, only CBEA is supported. There is no performance evaluation available for this approach. Our implementation of the scan algorithm is based on Harris et al.'s work [9] and gains from their optimizations. Xiao et al. [17] studied fast barrier synchronization between threadblocks in CUDA. Khanna [10] experimented with the performance and ease of use of the SPE software cache.

12 Conclusion

In this paper, we have demonstrated that OpenCL is an effective programming model for the CBE. We provided an implementation that maps the host to the PPE and each CU to an SPE, as well as a variant in which the PPE takes part in the computation as well. Consideration was restricted to data parallel programs and the data parallel programming model of OpenCL.

We evaluated the implementation with two applications: matrix multiplication, and the Einstein@Home client. With the former, we showed that a higher degree of OpenCL optimization leads to faster CBE programs, i.e., OpenCL correctly reflects major performance factors of the architecture. For the latter, an OpenCL program translated to CBE was shown to be only slightly slower than a native CBE program developed under similar conditions. Some architectural features such as the use of SIMD units and prefetching have not yet been studied, and are left for future research.

As another main contribution, we proposed two extensions to OpenCL. First, static local memory is an additional memory level that is easy to use and brings about performance gains such as a factor of seven for reduction on the CBE. It is beneficial beyond CBE, as has been shown for a GTX 280 GPU. Second, static work-groups enable manual work distribution and global synchronization with OpenCL. We demonstrated the usefulness of static WGs with a faster and simpler scan algorithm, for both CBE and the GTX 280 GPU.

References

[1] Jens Breitbart. Case studies on GPU usage and data structure design. Master's thesis, University of Kassel, 2008.

[2] Jens Breitbart. CuPP - a framework for easy CUDA integration. In *IEEE Int. Parallel and Distributed Processing Symposium*, 2009. Available in Digital Library.

[3] Jens Breitbart. Static GPU threads and an improved scan algorithm. In *Euro-Par 2010 Workshop Proceedings*, 2010. To appear.

[4] Jens Breitbart and Gaurav Khanna. An exploration of CUDA and CBEA for Einstein@Home. In *PPAM (1)*, pages 486–495, 2009.

[5] David Kirk and Wen-mei Hwu. *Programming Massively Parallel Processors: A Hands-on Approach.* Morgan Kaufmann, 1st edition, February 2010.

[6] Gregory Diamos, Andrew Kerr, and Mukil Kesavan. Translating GPU binaries to tiered SIMD architectures with Ocelot. Technical report, Georgia Institute of Technology, School of Electrical and Computer Engineering, 2009.

[7] Einstein@Home Homepage. `http://einstein.phys.uwm.edu`.

[8] Daniel Hackenberg. Fast matrix multiplication on Cell (SMP) systems. `http://www.tu-dresden.de/zih/cell/matmul`.

[9] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Parallel prefix sum (scan) with CUDA. In Hubert Nguyen, editor, *GPU Gems 3*, chapter 39, pages 851–876. Addison Wesley, August 2007.

[10] Gaurav Khanna. The CBE hardware accelerator for numerical relativity: A simple approach. *International Journal of Modeling, Simulation, and Scientific Computing (IJMSSC)*, Nov 2009.

[11] Khronos OpenCL API registry. `http://www.khronos.org/registry/cl/`.

[12] NVIDIA Corporation. NVIDIA CUDA compute unified device architecture programming guide. Version 2.3, 2009.

[13] The OpenCL Specification. Version 1, revision 43, May 2009.

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

[15] OpenCL PS3. `http://sites.google.com/site/openclps3/`.

[16] OpenMP Application Program Interface. Version 3.0, May 2008. `http://www.openmp.org/mp-documents/spec30.pdf`.

[17] Shucai Xiao and Wu-chun Feng. Inter-block GPU communication via fast barrier synchronization. In *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Atlanta, Georgia, USA, April 2010.