OpenCL – An effective programming model for data parallel computations at the Cell Broadband Engine

Jens Breitbart and Claudia Fohry
Research Group Programming Languages / Methodologies
Universität Kassel, Germany
Email:{*jbreitbart, fohry*}*@uni-kassel.de*

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 the Einstein@Home application, our translated OpenCL version achieves almost the same speed as a native CBE version.**

Another main contribution of the paper is a proposal for an additional memory level in OpenCL, called static local memory. With little programming expense, it can lead to significant speedups such as factor seven for reduction. Finally, we studied 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.

I. 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, e. g. GPUs, FPGAs and the Cell Broadband Engine (CBE), has entered the market and provides for 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) [1] was suggested recently as a standard for programming both multicore CPUs and accelerators. The standard is currently supported by NVIDIA in a developer version of their GPU driver, and by AMDs Stream SDK for x86 CPUs and AMD GPUs. Apple ships an OpenCL implementation with OS X 10.6. Furthermore, IBM released a first technical preview of OpenCL for the Power/VMX and CBE systems which, however, 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 goal.

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. II and III.

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. A device comprises multiple compute units, composed of multiple processing elements each. Computation is organized into work-groups that contain work-items to be run in parallel. Work items are mapped to processing elements, and work-groups are mapped to compute units. Memory is organized as a hierarchy, with a major bottleneck between local memories of compute units and global memory. To speed up memory accesses, techniques such as switching between multiple work-items mapped to the same processing element, 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 CBE is quite obvious: We identify the host with the PPE, and each compute unit with an SPE. The SPEs sequentially process the work-items that OpenCL runs in parallel on the processing elements. 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. III. While a compiler will be clearly needed for practical use, this experimental study relies on a manual implementation of

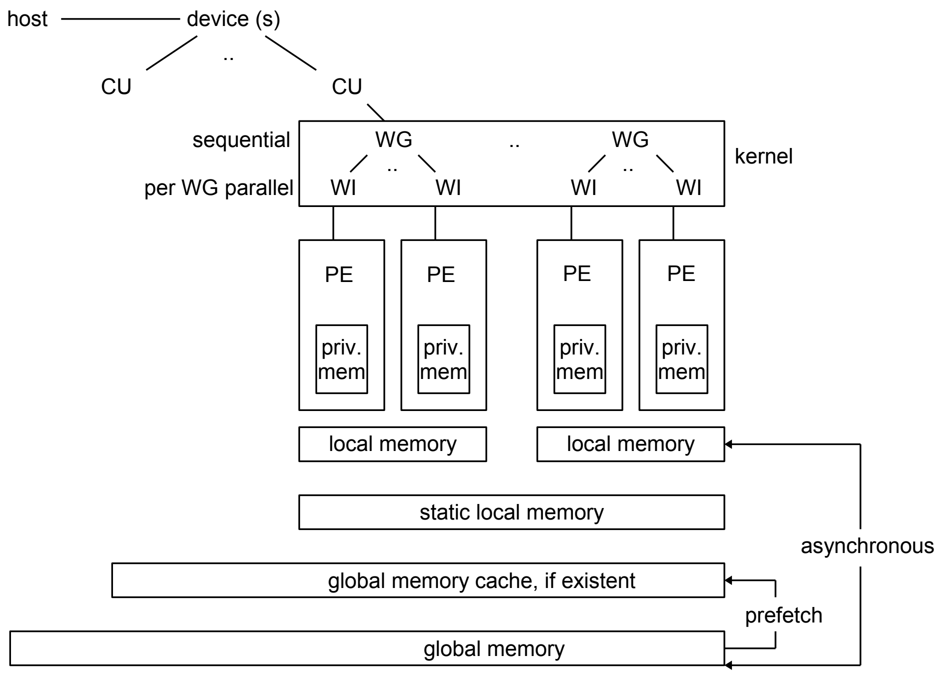

Figure 1. Extended OpenCL model

the mapping with C macros.

The question as to whether the mapping leads 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 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 attached to optimization.

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 the programs appear to be at the OpenCL level, the faster they 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, all major performance optimizations applicable on the CBE should be possible to express at OpenCL level. While the Einstein@Home result suggests this to be basically true, we found a performance optimization that can not yet be expressed: *static local memory*. In fact, this optimization is not restricted to CBE, and we suggest to extend OpenCL accordingly. In OpenCL, local memory belongs to a work-group, but occasionally data are reused between work groups. If they run on the same compute unit, 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 for static local memory, and provide two examples for the usefulness of this easy-to-use technique: constant tables and reduction. For reduction, speedups up to a factor of seven are achieved.

A forth contribution of this paper is experiments with the mapping scheme. In addition to the scheme described above, we studied a heterogeneous variant in which the PPE takes the role of both the host and an additional compute unit. This variant lead to additional speedups.

The paper is organized as follows. First, Sect. II gives an overview of the OpenCL standard and Sect. III introduces the CBE. Next, Sect. IV describes our mapping from OpenCL to the CBE, including remarks on implementation. Experiments with different optimization levels in OpenCL and their performance impact on the CBE are reported on in Sect. V. Section VI introduces the Einstein@Home application and compares implementations with and without OpenCL. Next, Sect. VII is devoted to the static local memory extension, and Sect. VIII to the heterogeneous mapping scheme. The paper finishes with an overview of related work

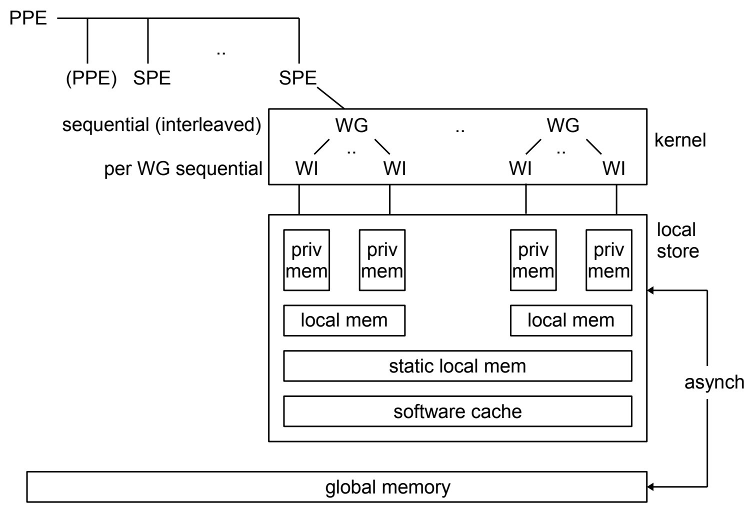

Figure 2. Mapping of extended OpenCL model to CBE

and conclusions, in Sects. IX and X, respectively.

II. OpenCL

OpenCL is a framework for parallel computing consisting of a language, API, libraries and a runtime system. It was originally designed by Apple, and then turned over to the Khronos Group. The first version of the standard was released in May 2009.

As depicted in Fig. 1, OpenCL is based on a platform model that divides a system into one *host* and one or several *devices* (e.g. GPUs). The devices act as co-processors to the host. They are subdivided into multiple so-called *compute units* (CUs), which are again subdivided into one or multiple *processing elements* (PEs). Note that the static local memory component is not part of the standard, but an extension proposed in this paper.

An OpenCL 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 in parallel on different data. Each instance is called a *work-item* (WI). While all work-items run the same kernel, they may perform different instructions at a time. Work-items can be arranged in so-called *work-groups* (WGs). OpenCL defines indexing schemes by which a work-item can be uniquely identified through either a global ID, or a work-group ID together with a local ID.

The work-groups are assigned to CUs, where the work-items of each group are run in parallel on the PEs. Typically, multiple work-groups are assigned to the same CU, and multiple work-items 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 work-groups or work-items, respectively). Synchronization of work-items is possible within a work-group only, and takes the form of a barrier. OpenCL also supports a fence operation to provide for memory consistency within a work-group.

This model is almost identical to NVIDIAs GPU programming model CUDA, except that CUDA uses different names. In CUDA, work-items are called threads, and work-groups are called threadblocks.

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 work-items and the host in reads/writes.
- *Constant memory* is similar to global memory, except that work-items may only read from this memory.
- *Local memory* is read/write memory local to a work-group, and is shared by all work-items of this group.
- *Private memory* is local to each work-item.

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 a pointer points to.

As a kernel can neither access host main memory nor dynamically allocate global and constant memory, all memory management must be done by the host. The OpenCL API provides functions to allocate linear memory blocks in

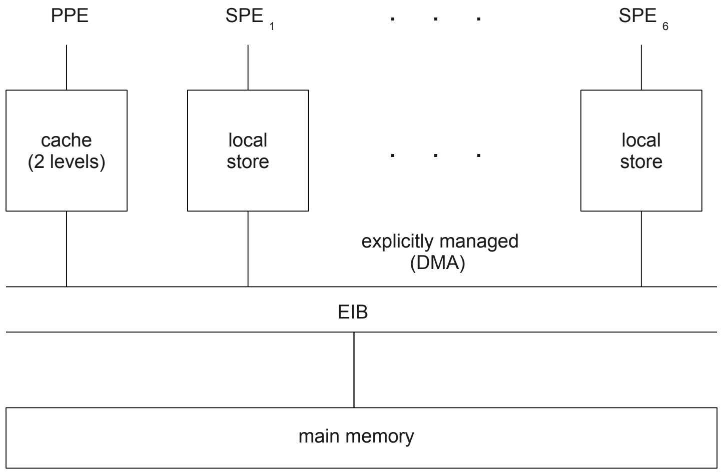

Figure 3. Cell Broadband Engine overview

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 work-items. 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 access, 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)

III. Cell Broadband Engine Overview

The CBE (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. 3, it is a heterogeneous CPU that 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.

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 access.

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 explicitly by 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.

IV. 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

Table I
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 II
RUNNING TIME IN SECONDS FOR MULTIPLYING TWO $2048 * 2048$ MATRIXES

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

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 [2]. The bindings were chosen since our programs partially rely on the CuPP library [3], a framework of easier-to-use interfaces for reoccuring tasks.

Our implementation is illustrated in Fig. 2. It maps the host to the PPE, and each CU to an SPE. A variant in which the PPE additionally takes the role of another CU is discussed in Sect. VIII. We experimented with one OpenCL device only. In the base variant, which is referred to except in Sect. VIII, work-groups are statically mapped to SPEs, i.e., each SPE processes about the same number of work-groups. The groups are processed sequentially, as are the work-items in the groups. When a barrier or memory fence is encountered, all work-items are run sequentally 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 a macro called `START`/`END` must be called. These macros start and end loops that are used to sequentially execute the work items. A barrier is implemented as a joined `END` and `START` macro, which first end the loops and than restart the loops, so it is guaranteed that all calculations for all work items prior the barrier are executed before the barrier or memory fence.

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 work-group, and variables in private memory are stored once for each work-item.

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 work-groups.

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

V. 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 work-item corresponds to the calculation of one element of C. Obviously, all work-items can be executed in parallel. An overview of the different versions is given in Table. I.

Table II 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 (a current high-end card with a theoretical peak performance of about 1 TFlop).

Version α is a naive matrix multiplication, in which 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 $C[0][j]$ $(j = 0 \dots n)$, but as work-items 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 II.

Version β uses a well-known blockwise matrix multiplication algorithm, which is described for instance in [4] 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 cor-

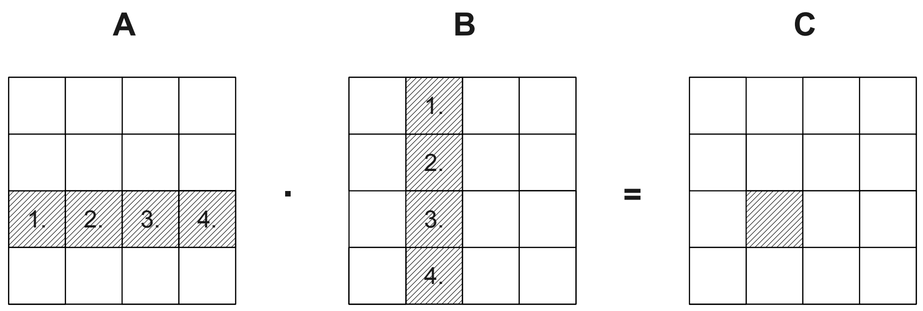

Figure 4. Blocked matrix multiplication overview

responding 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 work-group, and separating submatrix multiplications by a barrier. Version β is much faster than α at both the OpenCL and CBE levels, since data can be stored in global memory cache or software cache, respectively. Our experimental results show that the best performance is achieved with a work group 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 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 [5] 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 macros-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 work-groups are processed at a time. When one of them is waiting for global memory data, it is suspended and another work-group takes over. Only four work groups can be in progress, as otherwise their memory footprint (e.g. private memory data of all work-items) 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 work-group size. In all versions prior to ϵ, it was 128^2, but now it is 64^2. Having more work-groups increases the overhead for memory transfers, as it increases the numer of DMA transfers, and reduces the performance of the DMA engine. For this reason, version ϵ is slower than version δ. To isolate the effect, we re-run version δ with work-group size 64^2. In this case, version δ required 2.7 seconds, and is thus slower than version ϵ.

Overlapping memory transfers and calculations lead 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 work-groups and does not allow for 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 work-items 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 comprises this type of parallelism as well. Therefore, a compiler for the CBE can often bundle instructions of four work items to generate SIMD instructions.

Table III
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

Version ζ uses assembler code written by Hackenberg [6] that exploits the SIMD units to improve the performance of the calculations. The large improvement shown in Table II suggests that mapping parallel work-items to SIMD units is worthwhile. We leave this optimization for future work, as we do with the cache prefetch instruction.

VI. 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 [8]. It fits the data parallel model of OpenCL quite well, as there are ”groups” of independent calculations whose results must be reduced [9]. We reimplemented the CUDA version in OpenCL, mapping calculation groups to work-groups, and individual calculations to work-items.

Since the size of calculation groups varies, but OpenCL requires work-group sizes to be constant throughout a kernel call, the maximum size is computed prior to each call and work-groups with less calculations have idle work-items. 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 III 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.

VII. AN IMPROVED MEMORY SYSTEM

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: use of a lookup table in fast memory to calculate sinus and cosinus 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 work-group, and thus the table must be reinitialized for each work-group.

To prevent this problem, we propose a new level of memory that we call *static local memory*. This memory is local not to a work-group but to a CU, and can be accessed by all work-items scheduled there. Static local memory must be initialized by the first work-group, 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. V. Static local memory further speeds up the Einstein@Home application, as can be seen in the third line of Table III.

Besides constant tables, static local memory has more applications. We investigated reduction, which is an important parallel programming pattern. In our implementation, one work-item is used for every data item to be reduced. Each work-group 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 work-group of each CU accesses global memory. Table IV shows performance numbers for adding 2^{25} values. On six SPEs, a speedup of 7 is achieved. For the high importance of reductions, we consider this improvement worth adding a new memory level to OpenCL, especially as it is easy to use.

VIII. HETEROGENEOUS COMPUTING USING THE PPE

This section investigates 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 memory 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 work-groups to SPEs, using a similar scheme as the guided work distribution of OpenMP [10]: First large chunks of work are assigned, and then this size is decreased. As can be seen in the last line of Table III, inclusion of the PPE has a similar effect as adding an SPE. The small gain achieved on

Table IV
RUNNING TIME IN SECONDS FOR A REDUCTION OF 2^{25} ELEMENTS WITH AND WITHOUT STATIC LOCAL MEMORY

Hardware	PPE	GTX	Number of SPEs					
			1	2	3	4	5	6
local memory	10.1	1.2	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

six SPEs is due to the overall running time being dominated by the sequential fraction of the program. Dynamic scheduling induces little overhead: static scheduling optimized for Einstein@Home improved the performance by only 0.7% on six SPEs.

IX. Related Work

Our implementation resembles the one presented by IBM [5], 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 supports portability of OpenCL programs to different architectures.

Another project called *OpenCL PS3* [11] has 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 [12] 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.

X. 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 an extension to OpenCL, static local memory. This additional memory level is easy to use and brings about performance gains such as a factor of 7 for reduction. We expect static local memory to be beneficial beyond CBE for all hardware architectures that have local memory, e.g. GPUs. Investigation of other architectures as well as other algorithms that may profit from static local memory is left for future research.

References

[1] "The OpenCL Specification. Version 1, revision 43," May 2009. [Online]. Available: \url{http://www.khronos.org/registry/cl/specs/opencl-1.0.43.pdf}

[2] "Khronos OpenCL API registry," http://www.khronos.org/registry/cl/. [Online]. Available: \url{http://www.khronos.org/registry/cl/}

[3] J. Breitbart, "CuPP - a framework for easy CUDA integration," in *IEEE Int. Parallel and Distributed Processing Symposium*, 2009, available in Digital Library.

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

[5] "OpenCL development kit for linux on power," http://www.alphaworks.ibm.com/tech/opencl/. [Online]. Available: \url{http://www.alphaworks.ibm.com/tech/opencl/}

[6] D. Hackenberg, "Fast Matrix Multiplication on Cell (SMP) Systems," http://www.tu-dresden.de/zih/cell/matmul. [Online]. Available: \url{http://www.tu-dresden.de/zih/cell/matmul}

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

[8] J. Breitbart and G. Khanna, "An exploration of CUDA and CBEA for Einstein@Home," in *Int. Conf. on Parallel Processing and Applied Mathematics (PPAM)*. Springer LNCS, 2009, to appear.

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

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

[11] "OpenCL PS3," http://sites.google.com/site/openclps3/. [Online]. Available: \url{http://sites.google.com/site/openclps3/}

[12] G. Diamos, A. Kerr, and M. Kesavan, "Translating GPU binaries to tiered SIMD architectures with Ocelot," Georgia Institute of Technology, School of Electrical and Computer Engineering, Tech. Rep., 2009.