Automated Transformation of GPU-Specific OpenCL Kernels Targeting Performance Portability on Multi-Core/Many-Core CPUs*

Dafei Huang[1,2], Mei Wen[1,2], Changqing Xun[1,2], Dong Chen[1,2], Xing Cai[3], Yuran Qiao[1,2], Nan Wu[2,3], and Chunyuan Zhang[1,2]

[1] Department of Computer, National University of Defense Technology
[2] State Key Laboratory of High Performance Computing, Changsha, China
[3] Simula Research Laboratory, Oslo, Norway
hdafei@acm.org

Abstract. When adapting GPU-specific OpenCL kernels to run on multi-core/many-core CPUs, coarsening the thread granularity is necessary and thus extensively used. However, locality concerns exposed in GPU-specific OpenCL code are usually inherited without analysis, which may give side-effects on the CPU performance. When executing GPU-specific kernels on CPUs, local-memory arrays no longer match well with the hardware and the associated synchronizations are costly. To solve this dilemma, we actively analyze the memory access patterns by using array-access descriptors derived from GPU-specific kernels, which can thus be adapted for CPUs by removing all the unwanted local-memory arrays together with the obsolete barrier statements. Experiments show that the automated transformation can satisfactorily improve OpenCL kernel performances on Sandy Bridge CPU and Intel's Many-Integrated-Core coprocessor.

Keywords: OpenCL, Performance portability, Multi-core/many-core CPU, Code transformation and optimization.

1 Introduction

Heterogeneous computing systems, which incorporate two or more types of compute devices, are nowadays widely available from supercomputers to smart phones. A typical combination has been CPU plus GPU accelerator, while Intel's many-integrated-core (MIC) coprocessor is an increasingly popular choice of accelerator, such as in the currently No.1 supercomputer of the world: Tianhe-2. Programming, however, can be a challenge for using the heterogeneous devices for computations. The common strategy is to program separately for each type of the compute devices. Such a device-specific approach requires extensive

* Supported by the National Nature Science Foundation of China under No. 61033008, 61272145, and 61103080; 863 Program under No. 2012AA012706.

F. Silva et al. (Eds.): Euro-Par 2014, LNCS 8632, pp. 210–221, 2014.
© Springer International Publishing Switzerland 2014

programming effort, thereby difficult with respect to code maintenance and portability. An ideal scenario is thus to have the same source code base for multiple architectures, while maintaining a good level of performance portability.

OpenCL was designed with cross-platform code portability in mind. The advantage of adopting OpenCL programming is thus that a unified source code can work on different hardware architectures. On the other hand, however, performance portability does not come for free with OpenCL. The majority of existing OpenCL programs are GPU-specific, written with a bias or consensus toward getting good performance through making use of a massive number of threads, the round-robin instruction scheduling pattern, and the GPU-specific memory hierarchy [4][14]. These GPU-specific implementations, when executed directly on CPUs with heavy-weight cores, typically cannot achieve good performance [10].

Code transformation can provide a GPU-specific OpenCL program with performance portability to multi-core/many-core CPUs. A common technique of transformation is to enforce a coarser thread granularity, using the so-called *work-item coalescing* or *serialization* [12,13]. Moreover, work-items within a work-group are a primary source of vector- and instruction-level parallelism, both of which are typically exploited by a single CPU thread. However, the prior work concerning OpenCL code transformation has largely neglected to incorporate CPU-specific performance properties, such as spatial and temporal data locality [6], or directly inherit data locality features from a GPU-specific OpenCL kernel, often resulting in poor performance on CPUs [12,13]. What's more, when handling local memory and barriers, the existing code transformations have mainly concentrated on functionality and semantics but not performance, and without relevant analysis.

We will propose in this paper a new approach to transforming GPU-specific OpenCL kernels into a high-performance form that suits multi-core/many-core CPUs. It is based on a precise analysis of memory accesses, with help of a linear array-access descriptor. The resulting code transformation can thus remove all the unnecessary arrays that are allocated in OpenCL's local memory. In addition, all the unnecessary thread synchronizations are properly removed, instead of blindly using the known technique of *loop fission*. Thereafter, a post optimizer performs CPU-specific loop-level optimizations. The automatically transformed OpenCL kernels can be effectively executed on the multi-core/many-core architecture by using POSIX threads.

2 Related Work

There are many publications that address the challenge of adapting OpenCL code for the multi-core/many-core architecture targeting performance portability using code transformation, which directly translates GPU-specific OpenCL code into another code fit for CPUs.

Previous research activities that implement OpenCL for CPU platforms vary widely in the chosen approach to coalescing work-items and capturing

SIMD parallelism. The Twin Peaks method [6] utilizes `setjmp` and `longjmp` to merge fine-grain work-items into a single OS-thread, and performs vectorization within a work-item, but does not explore inter work-item parallelism. Region serialization methods [12,13] coalesce work-items by constructing thread loops and performing loop fission to reproduce the similar functionality of inter work-item synchronizations. They rely on an auto-vectorization technology within loop iterations to exploit parallelism. Intel's implementation of OpenCL for x86, being the least explicitly disclosed or studied, directly targets SIMD instructions and efficiently exploits vector-parallelism within a work-group [7]. None of the above implementations, however, handles data locality well enough, since they just depend on if the locality exposed on the GPU-specific code is suitable for the targeting CPU, so they may result in a strided access pattern by executing one or more work-items as long as possible, instead of interleaving the accesses of the work-items that can share the elements on one cache line. Stratton et al. rely on CEAN expression to do a more advanced handling of spatial locality [14]. Seo et al. adopt another approach from a different viewpoint [11], by automatically adapting the work-group size for better performance on multi-core CPU architecture.

No existing work can properly handle the issue of unnecessary use of local memory and synchronization. The state of the art usually uses arrays in OpenCL's global memory (main memory as to CPU) to simulate the ones in local memory, while ignoring the existence of caches on CPU. As for barriers, the Twin Peaks method directly uses jump instructions to simulate the function, which results in excessive overhead and breaks the locality in kernel code. Other approaches fully depend on the technique of loop fission, which also results in overhead of loop control instructions and variable extensions.

3 A Linear Descriptor of Array Access

An accurate identification of local and global memory access patterns is the key to a high-quality transformation from GPU-specific kernels to the CPU-matching counterparts. However, previously proposed descriptors of array access patterns have been designed for the scenario of nested loops, or not accurate enough to extract dependencies between work-items in the context of parallel SPMD OpenCL kernels [3][5].

We propose a precise linear descriptor of array accesses, based on the observation that most array accesses in a GPU-specific kernel can be expressed linearly. For example, the only exception to linear array accesses that can be found in Nvidia Computing SDK and the SHOC benchmark suite consists of indirect array accesses.

For each array that is accessed in any loop within a GPU-specific OpenCL kernel, our new array-access descriptor expresses the array index as a linear subscript function of only initial variables, that is: the work-item/work-group IDs, the loop induction variable, and the input arguments to the OpenCL kernel. In addition, a set of linear constraints, i.e., equalities and inequalities, are derived

```
   __kernel void matrixMul( __global float* C, __global float* A, __global float* B,
                            __local float* As, __local float* Bs, int uiWA, int uiWB )
   {
1    int aBegin = uiWA * BLOCK_SIZE * Gid.y;
2    int aEnd   = aBegin + uiWA - 1;
3    int aStep  = BLOCK_SIZE;
4    int bBegin = BLOCK_SIZE * Gid.x;
5    int bStep  = BLOCK_SIZE * uiWB;

6    float Csub = 0.0f;
7    for (int a = aBegin, b = bBegin; a <= aEnd; a += aStep, b += bStep)
8    {   AS[Lid.x + Lid.y * BLOCK_SIZE] = A[a + uiWA * Lid.y + Lid.x];
9        BS[Lid.x + Lid.y * BLOCK_SIZE] = B[b + uiWB * Lid.y + Lid.x];
10       barrier(CLK_LOCAL_MEM_FENCE);
11       for (int k = 0; k < BLOCK_SIZE; ++k)
12         Csub += AS[k + Lid.y*BLOCK_SIZE] * BS[Lid.x + k*BLOCK_SIZE];
13       barrier(CLK_LOCAL_MEM_FENCE);  }
14   C[(Gid.y*GROUP_SIZE_Y+Lid.y)*GLOBAL_SIZE_X + (Gid.x*GROUP_SIZE_X+Lid.x)] = Csub;
    }
```

Fig. 1. The original GPU-specific kernel of matrix multiplication

from the conditions of branches and loops to accurately pinpoint the range of the array index. As an illustrating example, Figure 1 shows the OpenCL kernel implementation of matrix multiplication, $C = A \times B$, available from Nvidia GPU Computing SDK. (It should be noted that some of the variables are renamed for clarity, and `Lid` denotes the local work-item ID, whereas `Gid` denotes the global work-group ID.) Within the outer loop of the kernel function there are six different array accesses: write access to `AS` and read access to `A` on line `8`, write access to `BS` and read access to `B` on line `9`, read access to both `AS` and `BS` on line `12`. Descriptors of the array accesses to `AS` and `A` (line `8`,`12`) are listed in Figure 2, where f denotes the linear subscript function, *Constraint* denotes the set of linear constraints, and $Iter_x(x = a, b, k)$ represent the normalized loop induction variables. For read access `A[a+uiWA*Lid.y+Lid.x]`, the linear function is f_A^{read}, derived by replacing `a` with its corresponding linear expression without any intermediate variable. The $Constraints_A^{read}$ limits the ranges of the variables in f_A^{read}, by combining loop conditions and intrinsic constraints on work-group and work-item IDs.

$$\begin{cases} f_A^{read} = (uiWA \times BLOCK_SIZE \times Gid.y + BLOCK_SIZE \times Iter_a) + uiWA \times Lid.y + Lid.x \\ Constraint_A^{read} = \{Iter_a \geq 0\ ;\ \ Iter_a < uiWA/BLOCK_SIZE;\ Gid.y \geq 0;\ Gid.y < GLOBAL_SIZE; \\ \qquad\qquad\qquad Lid.x \geq 0;\ Lid.x < BLOCK_SIZE;\ Lid.y \geq 0;\ Lid.y < BLOCK_SIZE\} \end{cases}$$

$$\begin{cases} f_{AS}^{write} = Lid.x + Lid.y \times BLOCK_SIZE \\ Constraint_{AS}^{write} = \{Lid.x \geq 0;\ Lid.x < BLOCK_SIZE;\ Lid.y \geq 0;\ Lid.y < BLOCK_SIZE\} \end{cases}$$

$$\begin{cases} f_{AS}^{read} = Iter_k + Lid.y \times BLOCK_SIZE \\ Constraint_{AS}^{read} = \{Iter_k \geq 0;\ \ Iter_k < BLOCK_SIZE;\ Lid.y \geq 0;\ Lid.y < BLOCK_SIZE\} \end{cases}$$

Fig. 2. Array access descriptors of accesses to `AS` and `A` in matrix multiplication

The derivation of a linear array-access descriptor, such as shown in Figure 2, is fully automated by taking advantage of the *Static Single Assignment* in LLVM infrastructure.

4 Transforming GPU-Specific OpenCL Kernels

4.1 Analysis-Based Coalescing

Work-item coalescing (or *serialization*) aims to merge the work-items of an entire work-group into a single CPU thread. The standard technique of coalescing is to construct a nested *thread loop*, where the loop levels correspond to the dimension of a work-group, the loop induction variables match the local work-item IDs, and the loop body is the original GPU-specific kernel code. A complicating factor, however, arises with thread synchronization. The state of the art is to adopt *loop fission* wherever synchronization appears. An example can be found in Figure 3.

```
Kernel_Name(Kernel_Args...)
{
    Kernel_Body_1...
    barrier();
    Kernel_Body_2...
}
```

(a) Original kernel with barrier

```
Kernel_Name(Kernel_Args...)
{
   for(Lid.z=0; Lid.z<GROUP_SIZE_Z; Lid.z++)
     for(Lid.y=0; Lid.y<GROUP_SIZE_Y; Lid.y++)
       for(Lid.x=0; Lid.x<GROUP_SIZE_X; Lid.x++)
       { Kernel_Body_1... }
   for(Lid.z=0; Lid.z<GROUP_SIZE_Z; Lid.z++)
     for(Lid.y=0; Lid.y<GROUP_SIZE_Y; Lid.y++)
       for(Lid.x=0; Lid.x<GROUP_SIZE_X; Lid.x++)
       { Kernel_Body_2... }
}
```

(b) Coaleced kernel using thread loop and loop fission

Fig. 3. Work-item coalescing by constructing thread loops

Considering the negative effects of blindly adopting loop fission, our remedy is to adopt an accurate dependence analysis, based on the linear descriptor of array accesses from Section 3, so that unnecessary thread synchronizations are eliminated, thereby avoiding loop fission.

Another performance-critical factor, in connection with work-item coalescing, is the use of OpenCL's local memory. Local memory array emulated by a segment of the slow main memory attached to a CPU may result in performance penalty, due to unnecessary data copies and additional thread synchronizations. This performance dilemma has received insufficient attention in the state of the art of work-item coalescing. Our novel contribution is therefore to eliminate all the unnecessary local-memory arrays during coalescing. This again will be based on the precise analysis of memory access patterns.

Eliminating Unnecessary Local-Memory Arrays

The functionality of local memory usage in GPU-specific kernels can be classified into three types:

1) Buffering: To improve temporal and spatial data locality within the kernel code, newly accessed data that are to be reused are buffered in OpenCL's local memory, so that long-latency global memory accesses are replaced by faster local memory accesses.
2) Reorganization: Data are loaded from OpenCL's global memory and stored in local memory using a different pattern, which allows coalesced memory accesses and effectively avoids bank conflicts. A representative example is

the transposed matrix multiplication ($C = A \times A^T$) kernel [8], where tiles of matrix A are loaded in rows but stored into columns of a local-memory array.

3) Enabling communication and reducing computation: Intermediate results of a work-item are stored in OpenCL's local memory before another work-item uses them. This type of usage not only reduces duplicated computations among different work-items, but also enables inter work-item communication.

On the multi-core/many-core architecture, functionality No. 3 also has to use OpenCL's local memory, thus work-item coalescing should not change this usage of local memory. For functionality No. 2, although data copy overhead arises due to the data reorganization, subsequent more efficient accesses to the reorganized data may still draw overall performance benefits. Regarding functionality No. 1, however, the usage of OpenCL's local memory becomes obsolete because the same effect can be achieved by the cache hierarchy on CPUs. Therefore, such a usage of local memory should be eliminated during coalescing. This requires an automated code analysis that can distinguish between the three usage types, together with automated replacement of local-memory array accesses with the corresponding global-memory array accesses.

Loads from local-memory arrays can be translated to direct global memory loads, provided the following two conditions are both satisfied:

(1) For a pair of local array write and read, by examining their array access descriptors, if some of the variables in the write descriptor are substituted with the variables of the read descriptor, the two descriptors become identical including the subscript functions and constraints.
(2) In this local array read-write pair, the write data is from a global memory read, which can be checked by using a definition-use chain.

After replacing the local array read with its corresponding global array read. The local array write will become dead code, and can be removed by compiler afterwards. An example is the following local array read-write pair from Figure 2:

$$\begin{cases} f_{AS}^{write} = Lid.x + Lid.y \times BLOCK_SIZE \\ Constraint_{AS}^{write} = \{Lid.x \geq 0;\ Lid.x < BLOCK_SIZE; \\ \qquad\qquad\qquad\qquad Lid.y \geq 0;\ Lid.y < BLOCK_SIZE\} \end{cases} \tag{4.1}$$

$$\begin{cases} f_{AS}^{read} = Iter_k + Lid.y \times BLOCK_SIZE \\ Constraint_{AS}^{read} = \{Iter_k \geq 0;\ \ Iter_k < BLOCK_SIZE; \\ \qquad\qquad\qquad\qquad Lid.y \geq 0;\ Lid.y < BLOCK_SIZE\} \end{cases} \tag{4.2}$$

If we substitute $Lid.x$ in (4.1) with $Iter_k$ from (4.2), the two descriptors become identical, which satisfies condition (1). Moreover, the write data of (4.1) is read from global array `A` according to line `8` in Figure 1, which satisfies condition (2):

$$\begin{aligned} f_A^{read} = (&uiWA \times BLOCK_SIZE \times Gid.y + BLOCK_SIZE \\ &\times Iter_a) + uiWA \times Lid.y + Lid.x \end{aligned} \tag{4.3}$$

So a transformation from local memory load to direct global memory load is legal, by performing the substitution of $Lid.x$ with $Iter_k$ in (4.3), and using it to replace (4.2):

$$\begin{aligned} f_{AS}^{read} &= Iter_k + Lid.y \times BLOCK_SIZE \quad \Rightarrow \\ f_A^{read} &= (uiWA \times BLOCK_SIZE \times Gid.y + BLOCK_SIZE \\ &\quad \times Iter_a) + uiWA \times Lid.y + Iter_k \end{aligned} \tag{4.4}$$

However, for local arrays with the data reorganization functionality, it is legal but not performance-beneficial. So an intuitive or heuristic condition is induced here to guarantee that a local array does not have the functionality of data reorganization:

(3) Looking at the linear subscript functions of a local array write and its respective global memory read, the variable $Lid.x$ has the same coefficient in the two functions (or that $Lid.x$ does not exist).

For example, in formulas (4.1) and (4.3), $Lid.x$ has coefficient 1 in both f_{AS}^{write} and f_A^{read}, and array accesses by (4.1) and (4.2) are the only accesses to local array `AS`. By using the condition above, we can conclude that local array `AS` does not have the functionality of data reorganization. By removing all the local arrays that only have the functionality of data buffering, and replacing them with direct accesses to global arrays, we can thus ensure good performance after work-item coalescing. Lines `8,9,12` in Figure 5 (line numbers remain the same as in Figure 1) shows the codes after eliminating the unnecessary local arrays `AS` and `BS`.

Dependence Analysis and Synchronization Elimination

Synchronization elimination happens after the unnecessary local arrays, the main source of synchronizations, are removed. However, we cannot simply delete all the barriers, since these may serve other local arrays that are not removed, or the synchronizations may use global memory. To check whether a barrier can be safely eliminated, dependence analysis is needed. Here, dependence analysis is very different from the typical scenario, because it is the dependence between different work-items that we care about.

When performing dependence analysis for a certain barrier, we first divide the kernel into basic blocks (barriers are also boundaries of the basic blocks). Then we examine every pair of array accesses (one of the accesses must be a write operation and both touch the same local or global array) that are located separately in two basic blocks before and after the barrier. The process is shown in Figure 4, where rectangles with dashed edge show the partitioning of basic blocks with different control structures, and arrows show the basic blocks within which array access pairs must be examined. The left part emphasizes that the examinations are for different work-items. For each examination, we combine the two descriptors of the access pair to form a linear *Diophantine Inequation System*. If there is a solution to the inequation system where not all the three pairs of local IDs are required to be equal, actual dependence exists and the barrier cannot be removed.

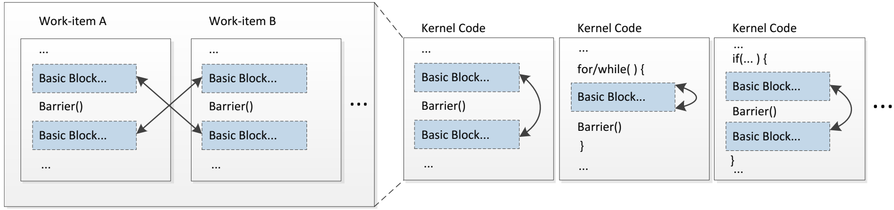

Fig. 4. An illustration of dependence analysis

$$\begin{cases} f_1 = \overrightarrow{Coe_1} \cdot \overrightarrow{Var_1}^{\mathrm{T}} + Const & \overrightarrow{Var_1} = (\ldots, Lid.z, Lid.y, Lid.x) \\ Constraint_1 \end{cases}$$
$$\begin{cases} f_2 = \overrightarrow{Coe_2} \cdot \overrightarrow{Var_2}^{\mathrm{T}} + Const & \overrightarrow{Var_2} = (\ldots, Lid.z', Lid.y', Lid.x') \\ Constraint_2 \end{cases} \quad (4.5)$$
$$\Rightarrow \begin{cases} f_1 = f_2 \\ Constraint_1 \\ Constraint_2 \end{cases}$$

Equation (4.5) shows the construction of an inequation system. The upper part shows two descriptors to be examined (Coe denotes the vector of coefficients, Var denotes the vector of variables, and $Const$ denotes a constant), and the lower part is the resultant system, generated by forcing the subscript functions to be equal while the both constraints are satisfied. Note that each local ID is no longer treated as the same variable in f_1 and f_2, so we use different names. A barrier must be reserved if the inequation system has a solution without the restriction $\{Lid.x = Lid.x'; Lid.y = Lid.y'; Lid.z = Lid.z'\}$.

By using the above dependence analysis, we can eliminate all the removable barriers in a GPU-specific kernel, and then enclose the kernel body by a thread loop. For non-removable barriers, loop fissions are inserted. Figure 5 shows the matrix multiplication kernel after coalescing, where both the barriers in the original kernel are eliminated.

```
     for (int Lid.y=0 ; Lid.y<BLOCK_SIZE; Lid.y++)
       for (int Lid.x=0 ; Lid.x<BLOCK_SIZE; Lid.x++) {
…          ...
6          float Csub = 0.0f;
7          for (int Itera=0, Iterb=0; Itera<=uiWA/BLOCK_SIZE; Itera++, Iterb++)
8,9        { //Dead Code
10           //Removed barrier(CLK_LOCAL_MEM_FENCE);
11           for(int Iterk=0; Iterk<BLOCK_SIZE; ++Iterk)
12             Csub += A[(uiWA*BLOCK_SIZE*Gid.y+BLOCK_SIZE*Itera)+uiWA*Lid.y+Iterk]
                   * B[(BLOCK_SIZE*Gid.x+BLOCK_SIZE*uiWB*Iterb)+uiWB*Iterk+Lid.x];
13           //Removed barrier(CLK_LOCAL_MEM_FENCE); }
14     C[(Gid.y*GROUP_SIZE_Y+Lid.y)*GLOBAL_SIZE_X+(Gid.x*GROUP_SIZE_X+Lid.x)] = Csub;
     }
```

Fig. 5. Code snippet of the matrix multiplication kernel after work-item coalescing

4.2 Post Optimizations

After the synchronization elimination described in Section 4.1, there are two unexploited CPU-specific performance properties of importance. The first is that inter work-item parallelism is buried, leading to insufficient utilization of the SIMD capability. The other is that loops in a coalesced code may be fused to such a degree that gives poor CPU-specific data locality. Figure 6(a) shows the unoptimized access sequences to arrays `A` and `B`, where iterative accesses to array `A` go through the whole long row, and accesses to `B` go through the whole column, resulting in successive cache misses. Furthermore, no SIMD parallelism is exploited.

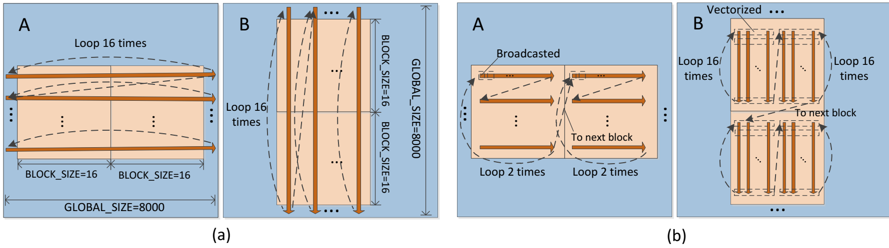

Fig. 6. Different access sequences to arrays `A` and `B`

We adopt two post optimizations of the coalesced code. They are combinations of traditional loop-level optimizations, but of vital effects on final performance.

Vectorization: The best loop level for performing vectorization should be that with induction variable `Lid.x`. This is because the coalesced memory accesses of a GPU-specific kernel often result in sequential and short-stride memory accesses across that loop level. So loop-interchange is firstly performed before ordinary vectorization so that `Lid.x`-loop becomes the innermost. The resultant effect as shown in Figure 6(b) is that, each scalar element of `A` is expanded into a vector, and each set of eight adjacent accesses to `B` is vectorized to produce a new vector. Then computational operations are fully vectorized so that the works of eight work-items are accomplished simultaneously.

Data locality re-exploitation: Our process of data locality re-exploitation has two steps, blocking of long non-thread-loops and loop interchange. As the result shown in Figure 6(b), the iterative array accesses are restricted in small blocks, so that the CPU cache can play a very good role.

The code snippet as the final output of the kernel transformation targeting the Sandy Bridge architecture can be found in Figure 7.

5 Performance Evaluation

We have implemented a fully automated tool chain that performs kernel transformation based on the Clang compiler front end and the LLVM compiler

```
for(int Itera=0, Iterb=0; Itera<=uiWA/BLOCK_SIZE; Itera++, Iterb++)
  for(int vLid.x=0; vLid.x<BLOCK_SIZE/8; vLid.x++)
    for(int Lid.y=0; Lid.y<BLOCK_SIZE; Lid.y++)
      for(int Iterk=0; Iterk<BLOCK_SIZE; ++Iterk)
        Csub[Lid.y][vLid.x]= vec_float8_add( Csub[Lid.y][vLid.x],
               vec_float8_mult(
                vec_float8_broadcast(A[(uiWA*BLOCK_SIZE*
                   Gid.y+BLOCK_SIZE*Itera)+uiWA*Lid.y+Iterk]),    //broadcast
                vec_float8_load(B+BLOCK_SIZE*Gid.x+BLOCK_SIZE*uiWB*Iterb+
                   uiWB*Iterk+vLid.x*8)                          //load
                              )           );                     //mult, add
```

Fig. 7. Final code snippet of the transformed matrix multiplication kernel

infrastructure [2]. The tool chain transforms a GPU-specific OpenCL kernel into a function, whose input arguments include the original ones from the GPU-specific kernel plus a set of work-group IDs. The vector operations are enabled by using Intel intrinsics. Each call to this function is equivalent with executing a corresponding work-group.

To run an entire OpenCL program that has both host and kernel code, the kernel transformation tool chain is integrated into an open source OpenCL implementation called FreeOCL [1], where POSIX threads are used to execute work-groups concurrently.

Experiments are carried out on two hardware platforms: (1) two Intel Xeon E5-2650 eight-core CPUs that have 16 physical cores together, as a typical multi-core CPU, (2) an Intel Xeon Phi 5110p coprocessor with 60 physical cores, as an emerging many-core CPU. The new OpenCL implementation, including our automated kernel transformation tool chain (denoted by OurOCL), is compared against the OpenCL implementation from Intel SDK for OpenCL Applications 2013, which is the official OpenCL runtime provided by Intel (denoted by IntelOCL).

Six kernels are used as the benchmarks. They cover a wide range of computational intensities and intrinsic memory localities. The first five kernels are optimized for running on GPUs so that they are well GPU-specific, where Stencil2D comes from SHOC and the remaining four kernels are from Nvidia GPU Computing SDK. The sixth kernel, NaiveMatrixMul, is the baseline matrix multiplication from [9], which is not so GPU-specific, and can show the potentiality of our method when few optimization features can be inherited.

IntelOCL is usually the most powerful commercial OpenCL runtime on Intel platforms, so we compare running the kernels via OurOCL, where kernels will be auto-transformed before execution, against running the same kernels via IntelOCL. When running the benchmarks, only the kernel execution times are recorded. Table 1 shows all the speedups of kernel executions relative to the CPU+IntelOCL configuration. The table indicates that OurOCL can improve the performance of GPU-specific kernels on multi-core CPUs by an average factor of 3.24x, not including the NaiveMatrixMul kernel. The average performance improvement of MIC+OurOCL over MIC+IntelOCL is 2.06x (3.53x/1.71x).

Table 1. Performance comparison with Intel OpenCL implementation and OpenMP

Kernel name	Scale	CPU + IntelOCL	CPU + OurOCL	CPU + OMP	MIC + IntelOCL	MIC + OurOCL	MIC + OMP
oclMatrixMul	8000 × 8000	1	3.02	0.37	1.94	3.95	3.74
oclFDTD3d	320×320×320 Radius=16 Timestep=5	1	6.02	2.20	2.22	5.88	4.13
Stencil2D	4096 × 4096 1000 iters	1	2.53	1.16	1.83	2.42	1.95
oclDCT8x8	10240 × 10240	1	3.42	2.27	1.43	4.17	4.52
oclNbody	327680	1	1.20	0.74	1.13	1.24	1.38
∗NaiveMatrixMul	8000 × 8000	1	33.48	4.10	4.55	43.76	41.43
Average (except NaiveMatrixMul)		1	3.24	1.35	1.71	3.53	3.14

IntelOCL is very good at utilizing the inter-work-group and inter-work-item parallelism by using the multiple cores and SIMD units. But its synchronization overhead is experimentally found to be somewhere between that of the region-based methods and the Twin Peaks method [14]. So the performance boost of OurOCL should be mainly attributed to the elimination of barriers and local-memory arrays, and partly the locality re-exploitation. The oclNbody kernel gets the minimum performance improvements on both platforms, because it is the most compute-intensive. The overheads induced by barriers and redundant memory copies only account for a small part of the kernel execution time. As for the two stencil computation kernels: oclFDTD3d and Stencil2D, improvements on MIC are much lower than those on CPU. This is because only a small portion of the execution time is used for computation as the two kernels are highly memory-intensive, so MIC can hardly show its superior parallel capability. The intensity of memory accesses also results in the slightly lower performances on MIC than those on CPU. On the other hand, the NaiveMatrixMul kernel obtains huge performance boosts because of both overhead removal and data locality improvement.

Performances of corresponding OpenMP implementations are also presented. The OpenMP implementations are based on the serial host implementations that can be found in every adopted benchmark, by properly adding OpenMP directives. (Execution of the OpenMP implementations on MIC uses the native mode.) We note that multi-core/many-core specific optimizations were already performed in some of the host implementations such as oclDCT8x8, and the *icc* can also automatically carry out various optimizations. Generally, improved OpenCL performances on both CPU and MIC are comparable with or even better than the OpenMP implementations. This shows that our automated code transformation can indeed greatly enhance performance portability.

6 Conclusion

To improve the performance portability of OpenCL programs from GPUs to CPUs, code transformation is widely accepted. This paper presents a novel transformation methodology for GPU-specific OpenCL kernels targeting

performance portability on multi-core/many-core CPUs, aiming at solving the potential problems induced by using local-memory arrays on CPUs, including redundant data copies and the accompanying costly synchronizations. A new array-access descriptor that can accurately uncover the array access patterns of OpenCL work-items lays the foundation of our work.

Experiments are done on Sandy Bridge CPU and Knights Corner MIC, which show that, for GPU-specific kernels, our new OpenCL implementation outperforms the powerful Intel OpenCL runtime on both platforms.

References

1. FreeOCL: multi-platform implementation of OpenCL 1.2 targeting CPUs, `https://code.google.com/p/freeocl/`
2. The LLVM compiler infrastructure, `http://llvm.org/`
3. Balasundaram, V., Kennedy, K.: A technique for summarizing data access and its use in parallelism enhancing transformations. In: SIGPLAN 1989 Conference on Programming Language Design and Implementation, Portland, USA, pp. 41–53 (1989)
4. Baskaran, M.M., Bondhugula, U., Krishnamoorthy, S., Ramanujam, J., Rountev, A., Sadayappan, P.: A compiler framework for optimization of affine loop nests for GPGPUs. In: 22nd International Conference on Supercomputing, Island of Kos, Greece, pp. 225–234 (June 2008)
5. Bastoul, C.: Code generation in the polyhedral model is easier than you think. In: 13th International Conference on Parallel Architectures and Compilation Techniques, Antibes Juan-les-Pins, France, pp. 7–16 (September 2004)
6. Gummaraju, J., Morichetti, L., Houston, M., Sander, B., Gaster, B.R., Zheng, B.: Twin peaks: A software platform for heterogeneous computing on general-purpose and graphics processors. In: 19th International Conference on Parallel Architectures and Compilation Techniques, Vienna, Austria, pp. 205–216 (September 2010)
7. Intel Corporation: Intel SDK for OpenCL Applications XE 2013 Optimization Guide (2013)
8. Nvidia: OpenCL Best Practices Guide (February 2011)
9. Nvidia: OpenCL Programming Guide for the CUDA Architecture (February 2011)
10. Pennycook, S., Hammond, S., Wright, S., Herdman, J., Miller, I., Jarvis, S.A.: An investigation of the performance portability of OpenCL. Journal of Parallel and Distributed Computing 73(11), 1439–1450 (2013)
11. Seo, S., Lee, J., Jo, G., Lee, J.: Automatic OpenCL work-group size selection for multicore CPUs. In: 22nd International Conference on Parallel Architectures and Compilation Techniques, Edinburgh, UK (September 2013)
12. Stratton, J.A., Grover, V., Marathe, J., Aarts, B., Murphy, M., Hu, Z., Hwu, W.M.W.: Efficient compilation of fine-grained SPMD threaded programs for multicore CPUs. In: 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization, Toronto, Canada, pp. 111–119 (April 2010)
13. Stratton, J.A., Stone, S.S., Hwu, W. M.W.: MCUDA: An effective implementation of CUDA kernels for multi-core CPUs. In: Amaral, J.N. (ed.) LCPC 2008. LNCS, vol. 5335, pp. 16–30. Springer, Heidelberg (2008)
14. Stratton, J.A., Kim, H.S., Jablin, T.B., Hwu, W.M.W.: Performance portability in accelerated parallel kernels. Tech. Rep. IMPACT-13-01, University of Illinois at Urbana-Champaign (May 2013)