Frontiers of Information Technology & Electronic Engineering
www.zju.edu.cn/jzus; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

Improving performance portability for GPU-specific OpenCL kernels on multi-core/many-core CPUs by analysis-based transformations*#

Mei WEN[1,2], Da-fei HUANG‡[1,2], Chang-qing XUN[1,2], Dong CHEN[1,2]

([1]*School of Computer, National University of Defense Technology, Changsha 410073, China*)

([2]*National Key Laboratory of Parallel and Distributed Processing, Changsha 410073, China*)

E-mail: meiwen@nudt.edu.cn; huangdafei1012@163.com; {xunchangqing, chendong}@nudt.edu.cn

Received Jan. 30, 2015; Revision accepted June 30, 2015; Crosschecked Oct. 19, 2015

Abstract: OpenCL is an open heterogeneous programming framework. Although OpenCL programs are functionally portable, they do not provide performance portability, so code transformation often plays an irreplaceable role. When adapting GPU-specific OpenCL kernels to run on multi-core/many-core CPUs, coarsening the thread granularity is necessary and thus has been 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. Typically, the use of OpenCL's local memory on multi-core/many-core CPUs may lead to an opposite performance effect, because 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 using array-access descriptors derived from GPU-specific kernels, which can thus be adapted for CPUs by (1) removing all the unwanted local-memory arrays together with the obsolete barrier statements and (2) optimizing the coalesced kernel code with vectorization and locality re-exploitation. Moreover, we have developed an automated tool chain that makes this transformation of GPU-specific OpenCL kernels into a CPU-friendly form, which is accompanied with a scheduler that forms a new OpenCL runtime. Experiments show that the automated transformation can improve OpenCL kernel performance on a multi-core CPU by an average factor of 3.24. Satisfactory performance improvements are also achieved on Intel's many-integrated-core coprocessor. The resultant performance on both architectures is better than or comparable with the corresponding OpenMP performance.

Key words: OpenCL, Performance portability, Multi-core/many-core CPU, Analysis-based transformation
doi:10.1631/FITEE.1500032 **Document code:** A **CLC number:** TP312

1 Introduction

Heterogeneous computing systems, which incorporate two or more types of computing 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, according to the TOP500 list released in Nov. 2014 (TOP500.org, 2014). Programming, however, can be a challenge for using the heterogeneous devices for computation. The common strategy is to program separately for each type of computing device. Take, for instance, a CPU-GPU hybrid system. The mainstream

‡ Corresponding author

* Project supported by the National Natural Science Foundation of China (No. 61272145) and the National High-Tech R&D Program (863) of China (No. 2012AA012706)
# A preliminary version of this paper was presented at Euro-Par 2014 Conference, Porto, Aug. 25, 2014

ORCID: Da-fei HUANG, http://orcid.org/0000-0001-6617-7608

approach uses CUDA programming for the GPUs, whereas OpenMP or POSIX threads are used for the CPUs. Such a device-specific approach requires extensive programming effort, and is 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 (Munshi, 2011) was designed with cross-platform code portability in mind. The advantage of adopting OpenCL programming is that a unified source code can work on different hardware architectures. On the other hand, performance portability does not come for free with OpenCL. Generally, performance portability refers to the performance levels that an application tuned for a particular architecture can achieve on other platforms. In this paper, we focus on improving a specific kind of performance portability, namely the performance portability of GPU-specific OpenCL kernels on multi-core/many-core CPUs. To evaluate the resultant improvement, the performance of the GPU-specific kernel under the vendor-provided OpenCL runtime is taken as the baseline, since it is the original kernel performance that indicates the performance portability implemented by the vendor.

The majority of existing OpenCL programs are GPU-specific, written with a bias or consensus toward obtaining good performance through making use of a massive number of threads, the round-robin instruction scheduling pattern, and the GPU-specific memory hierarchy (Baskaran *et al.*, 2008; Stratton *et al.*, 2013). These GPU-specific implementations, when executed directly on CPUs with heavy-weight cores, typically cannot achieve good performance (Rul *et al.*, 2010; Dong *et al.*, 2012; Pennycook *et al.*, 2013). For example, for GPU-specific OpenCL kernels written with consensus from the OpenCL community and GPU vendors (Stratton *et al.*, 2013), creating and running thousands of threads simultaneously to explore inter-workgroup and inter-workitem (intra-workgroup) parallelism would be unrealistic on a commodity multi-core CPU system. Another major problem is the distinct memory hierarchy of the OpenCL platform model, such as local-memory which makes it hard to natively map the memory hierarchy to other architectures.

Code transformation can give performance portability from a GPU-specific OpenCL program to multi-core/many-core CPUs. A common technique of transformation is to enforce a coarser thread granularity, or the so-called work-item coalescing (Lee *et al.*, 2010) or serialization (Stratton *et al.*, 2008; 2010), which means that the OpenCL work-items in one work-group are combined into a single, sequential CPU thread. Moreover, work-items within a work-group are a primary source of vector- and instruction-level parallelism, both of which can be effectively exploited by a single CPU thread. However, the prior work concerning OpenCL code transformation has largely neglected the incorporation of CPU-specific performance properties, such as spatial and temporal data locality (Gummaraju *et al.*, 2010), or directly inherits data locality features from a GPU-specific OpenCL kernel, often resulting in poor performance on CPUs (Stratton *et al.*, 2008; 2010). Moreover, when handling local memory and barriers, the existing automated code transformations have concentrated mainly on functionality and semantics but not performance, and without relevant analysis.

Our idea is to exploit CPU-specific performance properties, not fully depending on the original GPU-specific performance concerns. We propose 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 the 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, by making use of the parallelism properties and data locality, which can be extracted from the GPU-specific kernel. More specifically, we have developed a fully automated source-to-source code transformation tool chain. The input is GPU-specific OpenCL kernel code, and the output is an optimized function, which is of good data locality, effectively vectorized, and free of unnecessary local-memory usage and thread synchronization. An accompanying scheduler is also developed to execute the transformed OpenCL kernel on the multi-core/many-core architecture, by effectively using POSIX threads. Finally, a new OpenCL runtime implementation is

presented, with our automated kernel transformation tool chain as the key part.

Our contributions over existing work include:

1. A novel work-item coalescing methodology including removal of unnecessary local-memory arrays and synchronizations. The methodology is based on a linear array-access descriptor that accurately uncovers the actual array access patterns. The costs due to data replication and synchronization are greatly reduced, while easing subsequent CPU-specific optimizations.

2. A CPU-targeting post optimizer for the coalesced code, which incorporates loop-level optimization techniques, by extracting information of parallelism and locality embedded in the original GPU-specific kernel. The optimizer considers the architectural details of multi-cores/many-cores to enable a satisfactory performance boost on both CPUs and MICs.

3. A complete tool chain that automatically transforms GPU-specific kernels into an optimized form, which can effectively run on the multi-core/many-core architecture.

2 Related work

There are many publications that address the challenge of adapting OpenCL code for the multi-core/many-core architecture and targeting performance portability. These can be classified into two categories: code transformation and auto-tuning. The work presented in this paper falls in the first category, which directly translates GPU-specific OpenCL code into another code fit for CPUs.

Previous research that implements OpenCL for CPU platforms varies widely in the chosen approach to coalescing work-items and capturing SIMD parallelism. The Twin Peaks method (Gummaraju *et al.*, 2010) uses '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 (Stratton *et al.*, 2008; 2010) 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 (Intel Corporation, 2013b). None of the above implementations, however, handle data locality well enough, 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.* (2013) relied on CEAN expression to do a more advanced handling of spatial locality.

With regard to the local memory and accompanying synchronization, the state-of-the-art methods usually use 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 the 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 the kernel code. Other approaches above fully depend on the technique of loop fission, which also results in overhead of loop control instructions and variable expansions. The issue of ineffective use of local memory and synchronization has been studied recently. In our previous paper (Huang *et al.*, 2014), we proposed our work for the first time and emphasized the work-item coalescing method, but only slightly touched on the optimizations after coalescing. As an extension, this paper demonstrates our whole transformation tool chain including post optimizations and with a supporting scheduler in more detail, and provides a more systematic view of our work. Similar to our work, Fang *et al.* (2014a) presented a method to remove local memory usage automatically and effectively. However, their method was designed for use combined with auto-tuning, so the performance impact of local memory elimination was not considered in the transformation process. They also provided a performance impact indicator for local memory usage (Fang *et al.*, 2014b), but had not yet integrated it into their code transformer.

Auto-tuning (Du *et al.*, 2012; Pennycook *et al.*, 2013) is another widely used methodology. A set of performance-critical parameters is first identified in the code, and the best performance is achieved by tuning these parameters into the best combination. Compared with code transformation, some shortcomings are unavoidable: the manual coding work

is heavy, and the original OpenCL code should be rewritten or reconstructed to expose all parameters that possibly affect performance. Auto-tuning is also time-consuming and relatively non-robust. Some important parameters for one architecture may have no effect on another architecture, but auto-tuning may not realize this by itself, which results in wasting a lot of time. Pennycook *et al.* (2013) presented an architecture-independent method for developing single-source implementations targeting acceptable performance (instead of high performance) on different architectures.

The work of Phothilimthana *et al.* (2013) aimed at portable performance on heterogeneous architectures by combining the above two methodologies. They introduced and extended the PetaBricks language and its compiler. They have also automated the OpenCL code generation for multiple devices, while their code optimization depends mainly on an auto-tuner.

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, most of the previously proposed descriptors of array access patterns have been designed for the scenario of nested loops, providing useful information for optimizations such as automatic parallelization and privatization. Moreover, existing descriptors often use approximation, and are thus not accurate enough to extract dependencies between work-items in the context of parallel SPMD OpenCL kernels. Examples include triplet notation (Shen *et al.*, 1990) and linear memory access descriptor (Paek *et al.*, 2002). These existing descriptors record some basic features of array accesses, such as induction variables, upper and lower bounds of each induction variable, and stride between two accessing elements. Linear equalities and inequalities have been used to form linear constraint systems for describing array accesses, such as region in Triolet *et al.* (1986), data access descriptor in Balasundaram and Kennedy (1989), and the state-of-the-art polyhedral model in Bastoul (2004), but none of the above targets parallel programs. Among existing descriptors for data-parallel programs, the descriptor presented by Jang *et al.* (2011) models memory access patterns in a loop nest as a translation guidance targeting data-parallel architectures, which is a memory access matrix plus offset vector. Based on their work, Fang *et al.* (2014b) proposed a descriptor for OpenCL kernels. With that descriptor, they developed a performance impact indicator for local memory usage, which takes the descriptor and platform architecture as input and outputs the estimated performance impact if local memory is used.

In this paper, we propose a precise yet more flexible 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 (Danalis *et al.*, 2010) consists of indirect array accesses; i.e., array access indexes are elements of another array. In other words, although such a descriptor is limited to describing linear and direct memory accesses, it can cover the vast majority of GPU-specific OpenCL programs.

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 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 from the conditions of branches and loops to accurately pinpoint the range of the array index. As an example, Fig. 1 shows the OpenCL kernel function implementation of matrix multiplication, $\boldsymbol{C} = \boldsymbol{A} \times \boldsymbol{B}$, available from Nvidia GPU computing SDK (Here, 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 $\boldsymbol{AS}$ and read access to $\boldsymbol{A}$ on line 8, write access to $\boldsymbol{BS}$ and read access to $\boldsymbol{B}$ on line 9, read access to both $\boldsymbol{AS}$ and $\boldsymbol{BS}$ on line 12. Descriptors of the array accesses to $\boldsymbol{AS}$ and $\boldsymbol{A}$ are listed in Fig. 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. Descriptors of the array accesses to $\boldsymbol{BS}$ and $\boldsymbol{B}$ are very similar and are not listed in Fig. 2.

The derivation of a linear array-access descriptor, as shown in Fig. 2, is fully automated as a part

```
  __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 Nvidia GPU computing SDK

of the tool chain to be presented in Section 5.

4 Transforming GPU-specific OpenCL kernels

With the linear descriptor of array accesses at hand, transformation of GPU-specific OpenCL kernels will be carried out in two stages: analysis-based work-item coalescing and hardware-adaptive post optimization. The overall objective is to secure good performance of the transformed OpenCL kernels on the multi-core/many-core architecture.

4.1 Analysis-based coalescing

Recall that work-item coalescing (or serialization) aims to enforce a coarser thread granularity, by merging 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 with work-item coalescing, however, arises with thread synchronization in the GPU-specific kernel. The state-of-the-art methods are to adopt loop fission to handle thread synchronization in connection with coalescing. An example of thread loop construction can be found in Fig. 3, which shows that an original GPU-specific kernel that has a barrier statement is transformed into two nested loops due to loop fission. The negative effects of loop fission are additional control statements, and possible variable expansions to prevent variables from being overwritten before their use in subsequent thread loops. Our remedy for this performance problem is to adopt an accurate dependence analysis, based on the linear descriptor of array accesses, so that unnecessary thread synchronizations can be eliminated, thereby avoiding loop fission. The details will be discussed in Section 4.1.2.

Another performance-critical factor, in connection with work-item coalescing, is the use of OpenCL's local memory. It is very common that GPU-specific kernels use arrays that are allocated in the local memory, for the purpose of good performance when executed on GPUs. This GPU-specific strategy is well motivated because OpenCL's local memory is directly mapped to a GPU's on-chip fast memory. However, CPUs and MICs do not provide such a hardware support for OpenCL's local memory, which is in this case emulated by a segment of the slow main memory attached to a CPU or MIC. Blind usage of local-memory arrays on the multi-core/many-core architecture will thus result in a performance penalty, due to unnecessary data copies and additional thread synchronizations. This

$$
\begin{cases}
f_{\boldsymbol{A}}^{\mathrm{read}} = (\mathrm{uiWA} \times \mathrm{BLOCK\_SIZE} \times \mathrm{Gid}.y + \mathrm{BLOCK\_SIZE} \times \mathrm{Iter}_a) + \mathrm{uiWA} \times \mathrm{Lid}.y + \mathrm{Lid}.x, \\
\mathrm{Constraint}_{\boldsymbol{A}}^{\mathrm{read}} = \{\mathrm{Iter}_a \geq 0\ ;\ \ \mathrm{Iter}_a < \mathrm{uiWA/BLOCK\_SIZE};\ \mathrm{Gid}.y \geq 0;\ \mathrm{Gid}.y < \mathrm{GLOBAL\_SIZE}; \\
\qquad\qquad \mathrm{Lid}.x \geq 0;\ \mathrm{Lid}.x < \mathrm{BLOCK\_SIZE};\ \mathrm{Lid}.y \geq 0;\ \mathrm{Lid}.y < \mathrm{BLOCK\_SIZE}\},
\end{cases}
$$

$$
\begin{cases}
f_{\boldsymbol{AS}}^{\mathrm{write}} = \mathrm{Lid}.x + \mathrm{Lid}.y \times \mathrm{BLOCK\_SIZE}, \\
\mathrm{Constraint}_{\boldsymbol{AS}}^{\mathrm{write}} = \{\mathrm{Lid}.x \geq 0;\ \mathrm{Lid}.x < \mathrm{BLOCK\_SIZE};\ \mathrm{Lid}.y \geq 0;\ \mathrm{Lid}.y < \mathrm{BLOCK\_SIZE}\},
\end{cases}
$$

$$
\begin{cases}
f_{\boldsymbol{AS}}^{\mathrm{read}} = \mathrm{Iter}_k + \mathrm{Lid}.y \times \mathrm{BLOCK\_SIZE}, \\
\mathrm{Constraint}_{\boldsymbol{AS}}^{\mathrm{read}} = \{\mathrm{Iter}_k \geq 0;\ \ \mathrm{Iter}_k < \mathrm{BLOCK\_SIZE};\ \mathrm{Lid}.y \geq 0;\ \mathrm{Lid}.y < \mathrm{BLOCK\_SIZE}\}.
\end{cases}
$$

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

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

(b)
Kernel_Name(Kernel_Args...)
{
  for(Lid.z=0; Lid.z<GROUP_SIZE_Z; Lid.z++)
    for(Lid.y=0d; 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...
      }
}
```

Fig. 3 Work-item coalescing by constructing thread loops: (a) original kernel with barrier; (b) coalesced kernel using thread loop and loop fission

performance dilemma, which is induced by local-memory arrays, has not received sufficient attention so far in work on work-item coalescing. Our novel contribution is therefore to eliminate all the unnecessary local-memory arrays during coalescing automatically. This again will be based on a precise analysis of memory access patterns, enabled by the linear array-access descriptor from Section 3.

4.1.1 Eliminating unnecessary local-memory arrays

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

Type 1 (buffering): to improve temporal and spatial data locality within the kernel code, recently accessed data to be reused is buffered in OpenCL's local memory, so that long-latency global memory accesses can be replaced by faster local memory accesses.

Type 2 (reorganization): data is 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 when the data is later read from local memory. A representative example is the transposed matrix multiplication ($\boldsymbol{C} = \boldsymbol{A} \times \boldsymbol{A}^{\mathrm{T}}$) kernel (Nvidia Corporation, 2011a), where tiles of matrix $\boldsymbol{A}$ are loaded in rows but stored into columns of a local-memory array.

Type 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.

Type 4 (avoiding register spilling): if work-items have a mass of private data that exceeds the capacity of private memory (registers), private data will spill into low-speed off-chip memory. This type of usage treats local memory as an extension of private memory to store private data.

On the multi-core/many-core architecture, functionality Type 3 also has to use OpenCL's local memory, and thus work-item coalescing should not change this usage of local memory. Local memory with functionality Type 4 is also irreplaceable, because here local memory is used to store intermediate results and there is no other memory space declared to store the spilled data. For functionality Type 2, although OpenCL's local memory is emulated only by a segment of the main memory, and that 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 Type 1, however, the usage of OpenCL's local memory becomes obsolete because the same effect can be achieved by the cache hierarchy on CPUs (including MIC). Redundant memory copies are a plain waste. Therefore, such a usage of local memory should be eliminated during coalescing. This requires an automated code analysis that can distinguish between the four usage types, together with automated replacement of local-memory array accesses with the corresponding global-memory array accesses.

As long as a local-memory array does not have the functionality Type 3 or 4, but has functionality Type 1 or 2 or both, the sequence of accesses to this array must match the procedures below, which are also suggested in Nvidia Corporation (2011b):

1. Loading data from global memory and storing the data into the local memory array.

2. Possible synchronization with all the work-items in a work-group, so that each work-item can safely read the data that has been stored by different work-items.

3. Reading data from the local-memory array and using the data.

4. Possibly an additional synchronization if these procedures are iterative, so that data in the

local-memory array cannot be overwritten before usage.

For example, as shown in the code example of Fig. 1, two local-memory arrays, $\boldsymbol{AS}$ and $\boldsymbol{BS}$, are used for the purpose of buffering data shared within a work group. These two arrays are used exactly as in the above procedures. The values of $\boldsymbol{AS}$ and $\boldsymbol{BS}$ correspond directly to segments of the global-memory arrays $\boldsymbol{A}$ and $\boldsymbol{B}$, as shown in lines 8 and 9. We can also see in lines 10 and 13 the associated thread synchronizations.

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

Condition 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.

Condition 2: In this local array read-write pair, the write data is from a global memory read. This condition can be checked by using a definition-use chain (Steven, 1997), and the local array write and the respective global array read are usually in the same statement.

Loads from local-memory arrays with the functionality Type 3 or 4 do not satisfy the above two conditions because their elements are not read from global memory at any time.

Code transformation can be carried out as follows. For a local array read-write pair that satisfies the above two conditions, we can replace the local array read with its corresponding global array read. The local array write will become dead code, and can be removed by the compiler afterwards. An example is the following local array read-write pair from Fig. 2:

$$\begin{cases} f_{\boldsymbol{AS}}^{\text{write}} = \text{Lid}.x + \text{Lid}.y \times \text{BLOCK\_SIZE}, \\ \text{Constraint}_{\boldsymbol{AS}}^{\text{write}} = \\ \{\text{Lid}.x \geq 0;\ \text{Lid}.x < \text{BLOCK\_SIZE};\ \text{Lid}.y \geq 0; \\ \ \text{Lid}.y < \text{BLOCK\_SIZE}\}, \end{cases} \tag{1}$$

$$\begin{cases} f_{\boldsymbol{AS}}^{\text{read}} = \text{Iter}_k + \text{Lid}.y \times \text{BLOCK\_SIZE}, \\ \text{Constraint}_{\boldsymbol{AS}}^{\text{read}} = \\ \{\text{Iter}_k \geq 0;\ \text{Iter}_k < \text{BLOCK\_SIZE};\ \text{Lid}.y \geq 0; \\ \ \text{Lid}.y < \text{BLOCK\_SIZE}\}. \end{cases} \tag{2}$$

If we substitute Lid.x in Eq. (1) with Iter$_k$ from Eq. (2), the two descriptors become identical, which satisfies Condition 1. Moreover, the write data of Eq. (1) is read from global array $\boldsymbol{A}$ according to line 8 in Fig. 1, which satisfies Condition 2:

$$f_{\boldsymbol{A}}^{\text{read}} = (\text{uiWA} \times \text{BLOCK\_SIZE} \times \text{Gid}.y + \text{BLOCK\_SIZE} \times \text{Iter}_a) + \text{uiWA} \times \text{Lid}.y + \text{Lid}.x. \tag{3}$$

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 Eq. (3) and using it to replace Eq. (2):

$$\begin{aligned} & f_{\boldsymbol{AS}}^{\text{read}} = \text{Iter}_k + \text{Lid}.y \times \text{BLOCK\_SIZE} \\ & \Rightarrow \\ & f_{\boldsymbol{A}}^{\text{read}} = (\text{uiWA} \times \text{BLOCK\_SIZE} \times \text{Gid}.y + \\ & \text{BLOCK\_SIZE} \times \text{Iter}_a) + \text{uiWA} \times \text{Lid}.y + \text{Iter}_k. \end{aligned} \tag{4}$$

Such a transformation can be applied to local arrays not having the communication functionality nor containing spilled data. 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: looking at the linear subscript functions of a local array write and its respective global memory read, if variable Lid.x has the same coefficient in the two functions (or that Lid.x does not exist; i.e., the coefficient is 0), this local array does not have the functionality of data reorganization.

For example, in Eqs. (1) and (3), Lid.x has coefficient 1 in both $f_{\boldsymbol{AS}}^{\text{write}}$ and $f_{\boldsymbol{A}}^{\text{read}}$, and array accesses by Eqs. (1) and (2) are the only accesses to local array $\boldsymbol{AS}$. By using the condition above, we can conclude that local array $\boldsymbol{AS}$ does not have the functionality of data reorganization (only when a local array has no reorganization functionality, will its removal potentially benefit the resulting performance on multi-core/many-core CPUs). By removing all the local arrays that have only the functionality of data buffering, and replacing them with direct accesses to global arrays, we can thus ensure good performance after work-item coalescing. Fig. 4 shows the code snippet of the matrix multiplication kernel after eliminating the unnecessary local arrays $\boldsymbol{AS}$ and $\boldsymbol{BS}$, which both have the functionality of data buffering (line numbers in Fig. 4 are the same as in Fig. 1).

```
8   //Dead Code
9   //Dead Code
10  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  barrier(CLK_LOCAL_MEM_FENCE);
```

Fig. 4 Code snippet of the matrix multiplication kernel after eliminating unnecessary local memory usage

4.1.2 Dependence analysis and synchronization elimination

If different work-items in a work group access the same memory location, thread synchronization is typically needed, without which the result of kernel execution can be unpredictable. Barrier statements in GPU-specific kernels often lie very close to the local memory accesses, because local memory is shared within a work-group. However, the existence of a barrier does not necessarily mean that there is actual dependence (RAW, WAR, or WAW) between the different work-items, which is the root need for synchronization. For example, as pointed out in Section 4.1.1, only local memory usage of the communication functionality (Type 3) means that one work-item cannot proceed without another work-item's intermediate result. For local memory usages only with the other three functionalities, barriers are induced but without actual dependence between work-items. If each work-item directly accesses global memory for its own needed data, barriers are not necessary. That is, barriers induced by the reorganization and buffering functionalities of local memory can be eliminated.

The effect of synchronizations has two aspects. If loop fission is used to replace barriers during thread loop construction, additional costs such as control statements and variable expansion will be introduced. However, barriers also have the ability to change the execution flow of a work-group, which sometimes provides better temporal and spatial locality. Our strategy of handling synchronizations is: first we let work-item coalescing eliminate the synchronizations while neglecting their possible benefits to locality; then we re-exploit the locality that fits for the multi-core/many-core architecture through post optimizations, which will be presented in Section 4.2.

Synchronization elimination happens after the unnecessary local arrays 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 eliminated safely, 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 (dependences within a work-item are naturally preserved by sequential execution of statements and have nothing to do with synchronizations). The basic idea of our dependence analysis is that, if two work-items have accesses (one of the accesses must be a write operation) to the same local or global array, and the target access regions of the array overlap, then there is actual dependence between the two work-items.

When performing dependence analysis for a certain barrier, we first divide the kernel into basic blocks, each referring to a maximal group of statements such that one statement in the group is executed if and only if every other statement is executed, except that 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. In the upper part of Fig. 5, rectangles with dashed edges 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 lower part of Fig. 5 emphasizes that the examinations are for different work-items. When performing an examination, we

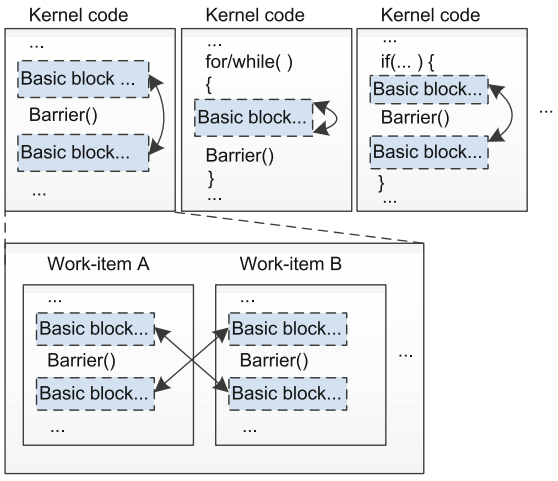

Fig. 5 An example of dependence analysis

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.

Eq. (5) shows the construction of an inequation system, where **Coe** denotes the vector of coefficients, **Var** the vector of variables, and Const a constant. 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 $\{\mathrm{Lid}.x = \mathrm{Lid}.x'; \mathrm{Lid}.y = \mathrm{Lid}.y'; \mathrm{Lid}.z = \mathrm{Lid}.z'\}$.

$$
\begin{cases} f_1 = \mathbf{Coe}_1 \cdot {\mathbf{Var}_1}^{\mathrm{T}} + \mathrm{Const}, \\ \mathrm{Constraint}_1 \quad \mathbf{Var}_1 = (..., \mathrm{Lid}.z, \mathrm{Lid}.y, \mathrm{Lid}.x), \end{cases}
$$

$$
\begin{cases} f_2 = \mathbf{Coe}_2 \cdot {\mathbf{Var}_2}^{\mathrm{T}} + \mathrm{Const}, \\ \mathrm{Constraint}_2 \quad \mathbf{Var}_2 = (..., \mathrm{Lid}.z', \mathrm{Lid}.y', \mathrm{Lid}.x'), \end{cases}
$$

$$
\Rightarrow \begin{cases} f_1 = f_2, \\ \mathrm{Constraint}_1, \\ \mathrm{Constraint}_2. \end{cases} \tag{5}
$$

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 so that those barriers become implicit, thus completing the work-item coalescing process. Fig. 6 shows the matrix multiplication kernel after coalescing, where both the barriers in the original kernel are eliminated. As shown, there are no more direct costs of synchronizations.

However, the execution flow of the entire work-group adopted by the original GPU-specific kernel is changed. Fig. 7a shows the original access sequence to global arrays $\boldsymbol{A}$ and $\boldsymbol{B}$, where each short row segment of matrix $\boldsymbol{A}$ is accessed 16 times serially, and rows of each matrix block in $\boldsymbol{B}$ are accessed in turn for 16 iterations in a coalesced manner (the threads in a half warp access a row simultaneously). Fig. 7b shows that of the coalesced code, where matrices are no longer blocked and iterative accesses to array $\boldsymbol{A}$ go through the whole long row, and accesses to $\boldsymbol{B}$ go through the whole column, resulting in successive cache misses. Furthermore, no SIMD parallelism is exploited. The coalesced code therefore needs some post optimizations that aim at specifically re-exploiting data locality and parallelism for the target CPU architecture, but not trying to recover the original locality in the GPU-specific implementation. This process of post optimization will be the subject of Section 4.2, with the aim of achieving the result shown in Fig. 7c for this particular example. In Fig. 7c, each scalar element of $\boldsymbol{A}$ is expanded into a vector, and each set of eight adjacent accesses to $\boldsymbol{B}$ is vectorized to produce a new vector. Then computational operations are fully vectorized. In addition, the iterative array accesses are restricted in small blocks, so that the CPU cache can play a very good role.

```
  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       //Dead Code
9       //Dead Code
10      //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      //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. 6 Code snippet of the matrix multiplication kernel after work-item coalescing

4.2 Architecture-adaptive post optimizations

After work-item coalescing is done as described in Section 4.1, the thread granularity becomes coarser so that a work-group can be well executed as a CPU thread. However, there are still two unexploited CPU-specific performance properties of importance. The first is that inter work-item parallelism may be buried, leading to insufficient utilization of the SIMD capability of a multi-core/many-core CPU. The other is that loops in a coalesced code may be fused to such a degree that results in poor CPU-specific data locality. In this study, we adopt two post optimizations of the coalesced code: vectorization and locality re-exploitation. Both optimizations will try to retain GPU-specific performance advantages and, more importantly, insert changes for CPU-specific performance improvement.

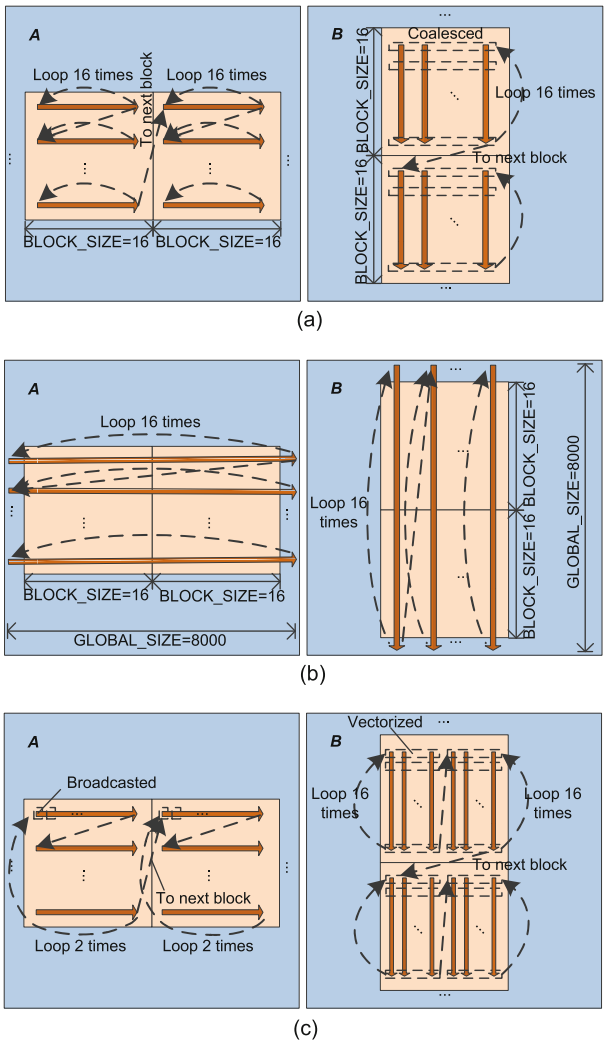

Fig. 7 Different access sequences to arrays A and B in connection with a matrix multiplication $C = A \times B$: (a) access sequences in the original GPU-specific kernel; (b) access sequences after work-item coalescing; (c) access sequences after post-optimizations

4.2.1 Vectorization

For the coalesced code, compilers such as 'icc' can automatically perform loop vectorization to use the SIMD capability of each core in a commodity multi-core/many-core CPU. However, compilers can vectorize only the innermost level of a loop nest (Intel Corporation, 2012), which may be un-vectorizable or not the most appropriate loop level to vectorize. We will therefore not rely on compiler vectorization, but explicitly insert vector instructions instead. Such a post optimization relies on the parallelism information existing in the GPU-specific kernel implementation, while addressing the specifics of multi-core/many-core CPUs.

On a GPU, the best access pattern for the global memory is sequential, unit-stride, and aligned (Nvidia Corporation, 2011a); i.e., the kth work-item accesses the kth word in an aligned global memory segment. This access pattern will result in coalesced memory accesses, which can best utilize a GPU's global memory bandwidth. The local memory accesses of a GPU-specific kernel usually also use the sequential and unit-stride (not necessarily aligned) pattern, for the purpose of avoiding bank conflicts. Another access pattern for the local memory is to let a half-warp of 16 contiguous work-items access the same local memory location.

Since OpenCL's global and local memory both reside in the main memory attached to a multi-core/many-core CPU, the best loop level for performing vectorization should be that with induction variable Lid.x. This is because sequential and unit-stride memory accesses across iterations can be transformed to vector loads and stores, whereas accesses across iterations to the same location can be transformed to vector-set/broadcast operations. The remaining arithmetic operations can be vectorized using the general loop vectorization method smoothly, since there is no loop dependence in the Lid.x loop after work-item coalescing.

Our vectorization-based optimization that can be applied to the coalesced code has the following steps:

1. If there are loops in the original GPU-specific kernel, we perform loop distribution so that the non-thread-loops become inner loops of the nested thread loop. This step may induce variable expansion and more control statements. Compared with the significant performance boost by vectorization, however, the extra costs are negligible.

2. For every loop nest, we perform loop interchange to move the Lid.x loop to the innermost level of the whole loop nest. This step is profitable due to vectorization and is always legal because there is no dependence across the thread loop levels.

3. Considering the SIMD width of the target CPU architecture, we perform loop blocking for the Lid.x loop, so that the induction variable of the blocked Lid.x loop (denoted as the vLid.x loop) can be incremented by SIMD-width/element-width. We then vectorize the innermost loop. For the Sandy Bridge architecture where the SIMD width is 256 bits, the loop increment is 8, when the kernel

operates on single-precision float-point numbers. As for the MIC Knights Corner architecture where the SIMD width is 512 bits, the loop increment is 16. Moreover, since fused multiply-add (FMA) is supported on Knights Corner, additions and multiplications in the same statements are fused.

The code snippet of the coalesced matrix multiplication kernel after vectorization is shown in Fig. 8, where a target CPU of 256-bit SIMD width is assumed and vLid.x is normalized. The prefix 'vec' stands for a vector data type or vector operation. For all the three loop nests after blocking Lid.x loops, the innermost loops are fully vectorized and eliminated.

```
for(int vLid.x=0; vLid.x<BLOCK_SIZE/8; vLid.x++)
  for(int Lid.y=0; Lid.y<BLOCK_SIZE; Lid.y++)
    Csub[Lid.y][vLid.x] = vec_float8(0.0f);

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 Itera=0, Iterb=0; Itera<=uiWA/BLOCK_SIZE;
        Itera++, Iterb++)
      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

for(int vLid.x=0; vLid.x<BLOCK_SIZE/8; vLid.x++)
  for(int Lid.y=0; Lid.y<BLOCK_SIZE; Lid.y++)
    vec_float8_store(C+(Gid.y*GROUP_SIZE_Y+Lid.y)
                     *GLOBAL_SIZE_X+Gid.x*GROUP_SIZE_X
                     +vLid.x*8, Csub[Lid.y][vLid.x]);
```

Fig. 8 Code snippet of the matrix multiplication kernel after vectorization

4.2.2 Data locality re-exploitation

After work-item coalescing and vectorization, the transformed kernel code has parallelism that is well suited for the multi-core/many-core architecture. However, data locality remains to be improved. Most GPU-specific kernels have data-locality considerations such as blocking long-trip loops, but in some kernels data locality is not well exploited. Take the 'naive matrix multiplication kernel', which is presented in Nvidia Corporation (2011b) as a baseline, as an example. This naive kernel has the same functionality as that of the kernel we discussed above, but it has no locality optimizations, and is thus used by Nvidia to show the usage of local memory.

Our data locality re-exploitation uses the original locality considerations for GPUs, while trying to fit the locality for the target multi-core/many-core architecture, so that GPU-specific kernels with poor original data locality can also become efficient on a multi-core/many-core CPU. Classical loop-level optimizations such as loop interchange, loop blocking, and improvement of register usage are implemented (with possible modifications), and architectural characteristics are also taken into account. Our process of data locality re-exploitation has the following three steps:

1. Blocking of non-thread-loops. The main idea is: if a loop whose induction variable appears in the contiguous dimension of a multi-dimensional array, but not in any other dimension, then blocking the loop is usually profitable (Allen and Kennedy, 2002). For every non-thread-loop, we block the loop if the following two conditions are both satisfied:

(a) The descriptor of one array access in the loop body contains the induction variable of the non-thread-loop, and the coefficient of the induction variable is small enough (A conservative threshold 2 is adopted, since a larger coefficient will result in access pattern with larger stride, which may limit the effectiveness of loop blocking).

(b) For the array access in (a), we convert the range of elements accessed during one iteration into bytes. This memory coverage range per iteration should be large enough (the threshold is set empirically to 1/8 of the L1 cache size, which is 4 KB for both Sandy Bridge and Knights Corner, since a loop with a narrow memory access range can also fit into the cache without the need of blocking).

Blocking will transform the non-thread-loop into two nested loops, and the trip count of the inner loop is set to the work-group size in the lowest dimension, i.e., the range of Lid.x. Although a typical loop blocking should include strip mining and loop interchange, only strip mining is used here and loop interchange will be considered for all the loops together in the next step. As shown in Fig. 8, since the Iter_k loop and Iter_a/Iter_b loop of the middle loop nest are actually two nested loops after blocking, no loop satisfies the blocking conditions. However, the

k loop in the naive matrix multiplication kernel will be automatically detected and blocked.

2. Loop interchange. The heuristic loop interchange algorithm proposed by Allen and Kennedy (2002) is adopted. For loop nest $\{L_1, L_2, \ldots, L_n\}$, a heuristic function is set up to estimate the number of cache misses incurred by each array access (including vector loads and stores). For each loop L_i, supposing it is positioned innermost, the number of misses of the whole nest (called innermost memory cost) is estimated as $C_M(L_i)$. Then the loops are re-arranged from inner to outer in descending order of their C_M values.

However, the above algorithm assumes infinite trip count of every loop. Since the thread loops and inner blocked loops usually have small trip counts, which embodies temporal data locality, the algorithm will not work effectively. So, we introduce a modification when calculating the innermost memory cost of a loop with a small trip count (threshold at the work-group size in the lowest dimension), by not multiplying with the trip counts of the outer loops. There are sometimes more than one advantageous interchange. For the middle loop nest of the code snippet in Fig. 8, the loop with $\mathrm{Iter}_a/\mathrm{Iter}_b$ should be placed outermost, and the Iter_k loop should be placed innermost, but the positions of the vLid.x and Lid.y loops are exchangeable.

3. Interchange selection for vector register reuse. Vector registers in CPU cores are scarce, 16 per core on Sandy Bridge and 32 on Knights Corner. The degree of vector register reuse can greatly influence the performance. Improving vector register reuse is actually equal to improving temporal data locality. The heuristic function used to calculate $C_M(L_i)$ is still adopted here, but with modifications such that it can quantify the temporal locality. Every vector operation is assumed as one 'memory access', the cache size is set to the number of vector registers, and the length of a cache line is set to 1. Then we use the modified heuristic function to estimate the number of 'cache misses' for each possible loop interchange identified in Step 2. The loop interchange with the fewest estimated 'cache misses' is adopted. In the middle loop nest of Fig. 8, if there are 16 256-bit vector registers, placing the vLid.x loop outside the Lid.y loop will achieve better temporal locality.

The code snippet of the vectorized matrix multiplication kernel after the entire process of locality re-exploitation can be found in Fig. 9 (showing only the middle loop nest), which is also the final output of the kernel transformation targeting the Sandy Bridge architecture. The final access sequences to arrays $\boldsymbol{A}$ and $\boldsymbol{B}$ are as in Fig. 7c, which shows that the original coalesced accesses in the GPU-specific kernel are transformed into vector accesses, and the data locality is greatly improved.

```
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. 9 Final code snippet of the matrix multiplication kernel after an automated transformation targeting Sandy Bridge

Fig. 10 shows the output of kernel transformation targeting Knights Corner, where the positions of the vLid.x and Lid.y loops are exchangeable. The differences between Figs. 9 and 10 are due to the different SIMD widths, the availability of FMA, and the number of vector registers.

5 A fully automated tool chain and a supporting scheduler

To automate the entire code transformation, which includes deriving array-access descriptors, removing unnecessary local-memory arrays and thread synchronizations, constructing thread loops, and post optimizations via vectorization and locality re-exploitation, we have implemented a fully automated tool chain that performs source-to-source kernel transformation based on the Clang (LLVM Team and others, 2012) compiler front end and the LLVM compiler infrastructure (Lattner and Adve, 2005). The automated tool chain transforms a GPU-specific OpenCL kernel into a linkable function, whose input arguments include the original ones to the GPU-specific kernel plus a set of work-group IDs. The

```
for(int vLid.x=0; vLid.x<BLOCK_SIZE/16; vLid.x++)
  for(int Lid.y=0; Lid.y<BLOCK_SIZE; Lid.y++)
    Csub[Lid.y][vLid.x] = vec_float16(0.0f);

for(int Itera=0, Iterb=0; Itera<=uiWA/BLOCK_SIZE;
    Itera++, Iterb++)
  for(int vLid.x=0; vLid.x<BLOCK_SIZE/16; vLid.x++)
  //vLid.x loop and Lid.y loop are exchangeable
  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_float16_FMA(
         vec_float16_broadcast(A[(uiWA*BLOCK_SIZE
                 *Gid.y+BLOCK_SIZE*Itera)
                 +uiWA*Lid.y+Iterk]), //broadcast
         vec_float16_load(B+BLOCK_SIZE*Gid.x
                +BLOCK_SIZE*uiWB*Iterb+uiWB*Iterk
                +vLid.x*16]),        //load
         Csub[Lid.y][vLid.x]);        //FMA

for(int vLid.x=0; vLid.x<BLOCK_SIZE/16; vLid.x++)
  for(int Lid.y=0; Lid.y<BLOCK_SIZE; Lid.y++)
    vec_float16_store(C+(Gid.y*GROUP_SIZE_Y+Lid.y)
                          *GLOBAL_SIZE_X+Gid.x*GROUP_SIZE_X
                          +vLid.x*16, Csub[Lid.y][vLid.x]);
```

Fig. 10 Final code snippet of the matrix multiplication kernel after an automated transformation targeting Knights Corner

vector operations are enabled by using Intel intrinsics (Intel Corporation, 2013a). Each call to the function is equivalent to executing a corresponding work-group; that is to say, the granularity for scheduling is a single work-group.

The transformed OpenCL kernel code, however, cannot be directly executed by any standard OpenCL runtime. To assist kernel execution, a scheduler is needed to call the translated function in a similar way as a standard OpenCL runtime schedules work-groups. The main idea is to statically assign groups of contiguous work-groups to the CPU cores as evenly as possible, which means the difference in assigned work-group counts of CPU cores is less than one. Such an approach transforms the inter work-group parallelism to thread-level parallelism.

With our new scheduler, the process of kernel execution is as follows:

1. The host thread divides the work-groups into equal sets, and assigns each work-group set contiguously to a logic core on the target multi-core/many-core platform (A physical core is assumed as two logical cores on Sandy Bridge, and four logical cores on Knights Corner except that the first available physical core is neglected).

2. For every logic core, the host thread creates a POSIX thread as a worker thread, and sets the affinity of each worker thread to the corresponding core.

3. Each worker thread calls the translated function iteratively with designated work-group IDs.

4. The host thread waits for the joining of the worker threads.

Fig. 11 shows the entire process where a GPU-specific kernel is transformed by our fully automated tool chain and then executed with help of the new scheduler.

To run an entire OpenCL program that has both host and kernel code, the kernel transformation tool chain and scheduler support are integrated into an open source OpenCL implementation called FreeOCL (Freeocl, 2012). The modules for code generation and kernel scheduling in FreeOCL are replaced by our transformation tool chain and scheduler. The other modules that implement OpenCL host APIs are left unchanged. When compiling the translated kernel code, Intel C++ compiler v13.0.0 has been used for both CPU and MIC. Compiler flags '-O3, -xHost' have been adopted for Sandy Bridge CPUs. As for Knights Corner coprocessors, offload pragmas are added into the transformed kernel code before compilation, so the whole process of kernel execution runs on MIC, where the chosen compiler optimization option is '-O3'. Note that since we use the offload mode to execute the transformed kernels, the last core of the coprocessor is reserved and unavailable for computation. OpenCL programs with GPU-specific kernels can thus run efficiently with our OpenCL runtime on multi-core/many-core platforms.

6 Performance evaluation

Numerical experiments and time measurements 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, and (2) an Intel Xeon Phi 5110p coprocessor with 60 physical cores, as an emerging many-core CPU. The Xeon CPUs also act as host where the operating system is Red Hat Enterprise Linux 6.2 with kernel version 2.6.32-220. The new OpenCL implementation, including our automated kernel transformation tool chain and execution support (denoted by OurOCL), is compared against the OpenCL implementation

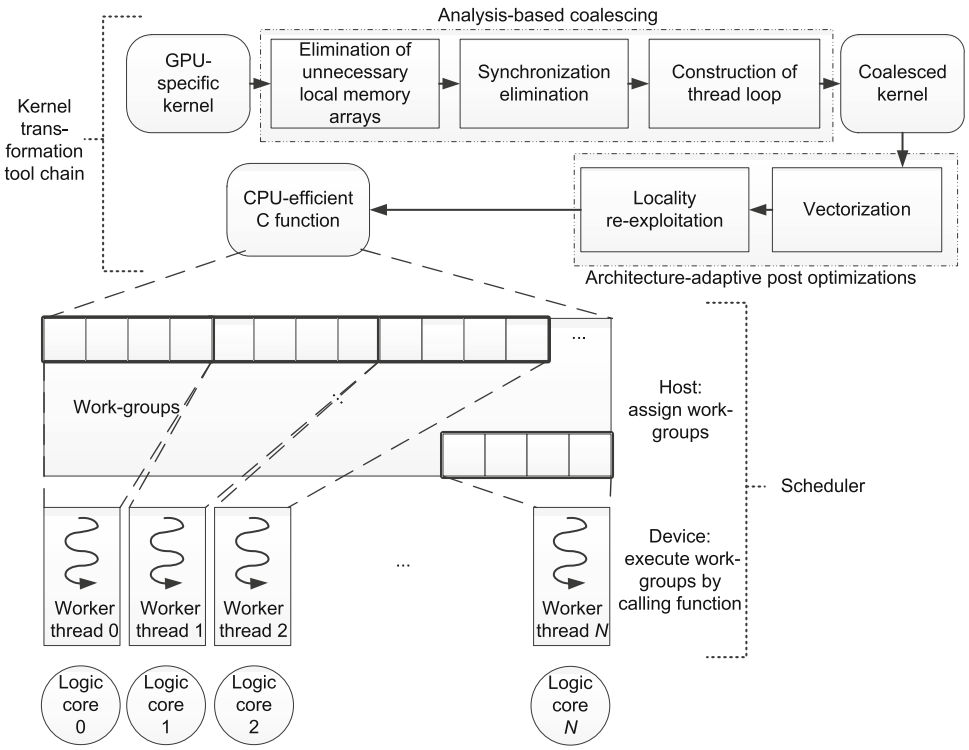

Fig. 11 Transformation and execution of a GPU-specific kernel

from Intel SDK for OpenCL Applications 2013, which is the official OpenCL runtime provided by Intel (denoted by IntelOCL).

Six GPU-specific kernels are used as the benchmarks (Table 1). The first five GPU-specific kernels are already optimized for running on GPUs, 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 Nvidia Corporation (2011b) as stated in Section 4.2.2, and its data locality is not well exploited.

As IntelOCL is usually the most powerful commercial OpenCL runtime on Intel platforms, the performance of a kernel under IntelOCL is commonly the best performance achievable with only vendor provided optimizations, and can indicate the 'official' performance portability. So, the improvement of performance portability can be evaluated by comparing it with the performance under IntelOCL. We compare running the GPU-specific 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 and with these the relative performance ratios are calculated. In Table 2, all the performance ratios are normalized by the performance of CPU+IntelOCL, and absolute kernel execution times are also provided in brackets. The table shows that OurOCL can improve the performance of GPU-specific kernels on multi-core CPUs by an average factor of 3.24×, not including the NaiveMatrixMul kernel. The average performance improvement of MIC+OurOCL over MIC+IntelOCL is 2.00×.

IntelOCL is very good at utilizing the inter-work-group and inter-work-item parallelism by using the multiple cores and SIMD units. However, its synchronization overhead is experimentally found to be somewhere between that of the region-based methods and the Twin Peaks method (Stratton *et al.*, 2013). So, the performance boost of OurOCL should be attributed mainly to the elimination of barriers and local-memory arrays, and partly to the locality re-exploitation. The oclNbody kernel achieves the minimum performance improvements on both platforms, because it is the most compute-intensive. The overheads induced by barriers and redundant memory copies account for only a small part of the kernel execution time. As for the two stencil computation

Table 1 Six benchmarks used for performance evaluation

Kernel name	Scale	Local work size	Description
oclMatrixMul	8000 × 8000	16 × 16	Matrix multiplication with blocking
oclFDTD3d	320 × 320 × 320, Radius=16, Timestep=5	32 × 32	Finite differences time domain progression, 3D stencil calculation
Stencil2D	4096 × 4096, 1000 iterations	16 × 16	Standard 2D 9-point stencil calculation
oclDCT8x8	10 240×10 240	32 × 2	Discrete cosine transform (DCT) for 8 × 8 block
oclNbody	327 680	256	Gravitational simulation of 327 680 bodies
NaiveMatrixMul	8000 × 8000	16 × 16	Matrix multiplication without blocking or local memory usage

Table 2 Performance comparison with Intel OpenCL implementation and OpenMP

Kernel name	Performance ratio (execution time)					
	CPU+IntelOCL	CPU+OurOCL	CPU+OMP	MIC+IntelOCL	MIC+OurOCL	MIC+OMP
oclMatrixMul	1 (23.30 s)	3.02 (7.71 s)	0.37 (62.98 s)	1.94 (12.03 s)	3.93 (5.93 s)	3.74 (6.23 s)
oclFDTD3d	1 (0.80 s)	6.15 (0.13 s)	2.16 (0.37 s)	2.22 (0.36 s)	5.71 (0.14 s)	4.21 (0.19 s)
Stencil2D	1 (20.65 s)	2.53 (8.16 s)	1.16 (17.80 s)	1.83 (11.26 s)	2.42 (8.53 s)	1.95 (10.59 s)
oclDCT8x8	1 (75.96 ms)	3.42 (22.20 ms)	2.27 (33.46 ms)	1.43 (53.22 ms)	4.18 (18.19 ms)	4.52 (16.81 ms)
oclNbody	1 (10.63 s)	1.20 (8.82 s)	0.74 (14.36 s)	1.13 (9.44 s)	1.24 (8.59 s)	1.38 (7.70 s)
NaiveMatrixMul	1 (258.16 s)	33.44 (7.72 s)	4.10 (62.98 s)	4.55 (56.73 s)	43.76 (5.90 s)	41.44 (6.23 s)

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.

To show the extent of performance improvement achieved by OurOCL, performances of corresponding OpenMP implementations are also presented in Table 2. The OpenMP implementations are based on the serial host implementations that are used to verify the correctness of kernel execution and can be found in every adopted benchmark, by properly adding OpenMP directives (execution of the OpenMP implementations on MIC uses the native mode). This kind of implementation does not mean no other optimizations are conducted. We note that multi-core/many-core specific optimizations have already been performed in some of the host implementations, especially oclDCT8x8, and the icc can also automatically carry out various optimizations more freely without the constraints from OpenCL semantics. What is more, we note that the compiler optimizations for MIC are more aggressive and efficient than those for CPU. As a result, the performance of oclDCT8x8 under MIC+OMP is slightly better than the performance under MIC+OurOCL. Although the OpenMP version of oclNbody is not as optimized as oclDCT8x8, it reports a better performance compared to MIC+OurOCL. The reason is still under exploration, and we assume that it is because oclNbody has a relatively simple array access pattern and the scheduling strategy of MIC is also more effective than the static scheduler in OurOCL. Generally, improved OpenCL performances with OurOCL on both CPU and MIC are comparable with or even better than the OpenMP implementations. This shows that our automated code transformation with scheduler support can alleviate the negative factors in GPU-specific kernels and enhance the performance to match the moderately optimized CPU-specific implementations.

Fig. 12 shows the performance improvement that is due to each step of architecture-adaptive post optimizations on both CPU and MIC. Relative performances compared to the relevant original GPU-specific kernels are reported in the figure. Note that only oclMatrixMul and NaiveMatrixMul under Sandy Bridge are provided with a performance improvement at the last step, because only those two have multiple candidate interchange schemes

after performing loop interchange. For the others having a unique loop interchange scheme, the last step is omitted. From Fig. 12, we can conclude that, for the coalesced kernel codes that suffer from heavy losses in locality and lack of parallelism after coalescing (oclMatrixMul, oclFDTD3d, oclNbody), our post optimizations will aggressively extract parallelism and re-exploit locality. For those coalesced codes of closer or even better performance compared to the original GPU-specific kernels (Stencil2D, oclDCT8x8), because the benefits of synchronization and local memory elimination are comparable with the potential losses in locality and parallelism, the optimizations can further improve the performance.

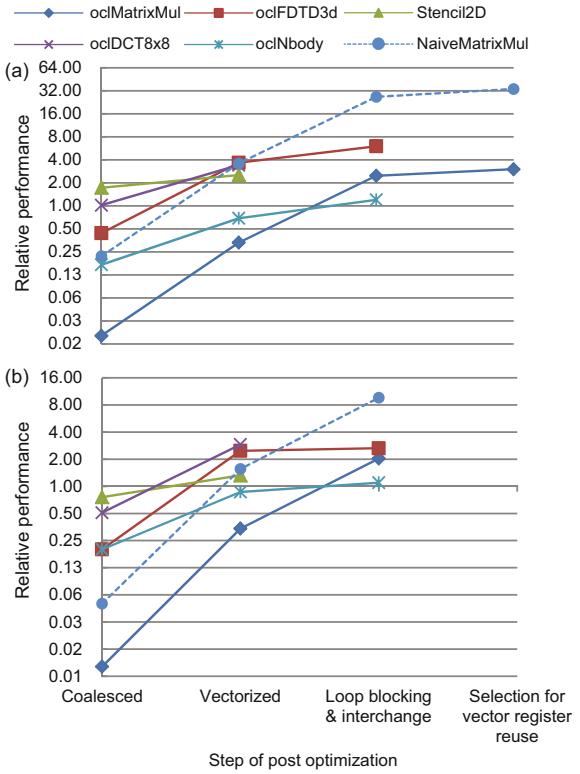

Fig. 12 Performance improvement compared to the relevant original GPU-specific kernels due to each step within post optimization: (a) CPU; (b) MIC

According to the figure, the vectorization for oclMatrixMul and oclFDTD3d can greatly improve the performance, and some improvements even exceed the SIMD length. The reason is that loop distribution and interchange are the first two steps of the vectorization, which may also benefit the locality, so an additional performance boost is achieved. However, the vectorization for oclNbody results in relatively low improvement, because the speeds and positions of 'bodies' are arranged as short vectors, similar to the array of structure pattern, which baffles efficient vectorization between work-items. The improvements of NaiveMatrixMul are almost the same as those of oclMatrixMul, since the coalesced codes of the two kernels are of the same execution flow. Loop blocking is applicable to NaiveMatrixMul, and after loop blocking the kernel becomes exactly the same as oclMatrixMul.

7 Conclusions

To improve the performance portability of OpenCL programs from GPUs to CPUs, code transformation has been 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 two particular problems that have not been well addressed by existing transformation methods. One is the potential side-effects induced by using local-memory arrays on CPUs, including redundant data copies and the accompanying costly synchronizations; the other is the possibility of poor data locality caused by neglecting or blindly inheriting locality embedded in the original GPU-specific kernels.

A new array-access descriptor that can accurately uncover the array access patterns of OpenCL work-items is the foundation of our work. With the help of descriptors derived from GPU-specific kernels, our novel work-item coalescing method can remove all the harmful local-memory arrays together with the obsolete barriers. Post optimizations for coalesced kernel code include vectorization and data locality re-exploitation, which not only extract parallelism and locality in the original GPU-specific kernels, but also consider the architectural details of the target platform. These transforming procedures are implemented as a fully automated tool chain, accompanied with a scheduler that forms a new OpenCL runtime.

Experiments are done on Sandy Bridge CPU and Knights Corner MIC, two leading representatives of the multi-core/many-core architecture. Measurements show that, for GPU-specific kernels, our new OpenCL implementation outperforms the powerful Intel OpenCL runtime on both platforms. Im-

proved OpenCL performances of our runtime are comparable with or even better than the performances of corresponding OpenMP implementations. This proves the effectiveness of our code transformation methodology.

Our proposed approach also exposes some limitations. First, the linear descriptor of array access can support only affine access patterns and cannot be applied to indirect array accesses. Although some application-specific techniques are proposed, it remains an open problem in generic code transformation research. Second, since this study focuses on the transformation techniques and uses only a simple static scheduling strategy, more effective schedule methods for CPU architecture can be developed to further boost the overall performance. Third, the performance impact of local memory is a complicated problem (Fang *et al.*, 2014b), and our analysis approach is limited to common usages of local memory. We will look further into this limitation and improve the applicability of our work. Last but not least, the transformation is restricted to the scope within a single work-group, so some performance factors regarding multiple cores such as the usage of shared cache (L3 cache of Sandy Bridge and L2 cache of Knights Corner) remain unexplored. We plan to study inter-work-group optimization in future work.

References

Allen, R., Kennedy, K., 2002. Optimizing Compilers for Modern Architectures: a Dependence-Based Approach. Morgan Kaufmann, San Francisco.

Balasundaram, V., Kennedy, K., 1989. A technique for summarizing data access and its use in parallelism enhancing transformations. *ACM SIGPLAN Not.*, **24**(7):41-53. [doi:10.1145/74818.74822]

Baskaran, M.M., Bondhugula, U., Krishnamoorthy, S., *et al.*, 2008. A compiler framework for optimization of affine loop nests for GPGPUs. Proc. 22nd Annual Int. Conf. on Supercomputing, p.225-234. [doi:10.1145/1375527.1375562]

Bastoul, C., 2004. Code generation in the polyhedral model is easier than you think. Proc. 13th Int. Conf. on Parallel Architectures and Compilation Techniques, p.7-16. [doi:10.1109/PACT.2004.1342537]

Danalis, A., Marin, G., McCurdy, C., *et al.*, 2010. The scalable heterogeneous computing (SHOC) benchmark suite. Proc. 3rd Workshop on General-Purpose Computation on Graphics Processing Units, p.63-74. [doi:10.1145/1735688.1735702]

Dong, H., Ghosh, D., Zafar, F., *et al.*, 2012. Cross-platform OpenCL code and performance portability for CPU and GPU architectures investigated with a climate and weather physics model. Proc. 41st Int. Conf. on Parallel Processing Workshops, p.126-134. [doi:10.1109/ICPPW.2012.19]

Du, P., Weber, R., Luszczek, P., *et al.*, 2012. From CUDA to OpenCL: towards a performance-portable solution for multi-platform GPU programming. *Parall. Comput.*, **38**(8):391-407. [doi:10.1016/j.parco.2011.10.002]

Fang, J., Sips, H., Jaaskelainen, P., *et al.*, 2014a. Grover: looking for performance improvement by disabling local memory usage in OpenCL kernels. Proc. 43rd Int. Conf. on Parallel Processing, p.162-171. [doi:10.1109/ICPP.2014.25]

Fang, J., Sips, H., Varbanescu, A.L., 2014b. Aristotle: a performance impact indicator for the OpenCL kernels using local memory. *Sci. Progr.*, **22**(3):239-257. [doi:10.3233/SPR-140390]

Freeocl, 2012. FreeOCL: multi-platform implementation of OpenCL 1.2 targeting CPUs. Available from https://code.google.com/p/freeocl [Accessed on Apr. 13, 2014].

Gummaraju, J., Morichetti, L., Houston, M., *et al.*, 2010. Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors. Proc. 19th Int. Conf. on Parallel Architectures and Compilation Techniques, p.205-216. [doi:10.1145/1854273.1854302]

Huang, D., Wen, M., Xun, C., *et al.*, 2014. Automated transformation of GPU-specific OpenCL kernels targeting performance portability on multi-core/many-core CPUs. Proc. Euro-Par, p.210-221. [doi:10.1007/978-3-319-09873-9_18]

Intel Corporation, 2012. A Guide to Vectorization with Intel C++ Compilers.

Intel Corporation, 2013a. Intel C++ Intrinsic Reference. Available from https://software.intel.com/sites/default/files/a6/22/18072-347603.pdf [Accessed on Feb. 9, 2014]

Intel Corporation, 2013b. Intel SDK for OpenCL Applications XE 2013 Optimization Guide. Available from http://software.intel.com/en-us/vcsource/tools/opencl-sdk-xe/ [Accessed on Feb. 9, 2014]

Jang, B., Schaa, D., Mistry, P., *et al.*, 2011. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *IEEE Trans. Parall. Distr. Syst.*, **22**(1):105-118. [doi:10.1109/TPDS.2010.107]

Lattner, C., Adve, V., 2005. The LLVM compiler framework and infrastructure tutorial. *In*: Eigenmann, R., Li, Z.Y., Midkiff, S.P. (Eds.), Languages and Compilers for High Performance Computing. Springer, p.15-16.

Lee, J., Kim, J., Seo, S., *et al.*, 2010. An OpenCL framework for heterogeneous multicores with local memory. Proc. 19th Int. Conf. on Parallel Architectures and Compilation Techniques, p.193-204. [doi:10.1145/1854273.1854301]

LLVM Team and others, 2012. Clang: a C language family frontend for LLVM. Available from http://clang.llvm.org/ [Accessed on Apr. 13, 2014].

Munshi, A., 2011. The OpenCL specification. Available from http://www.khronos.org/opencl [Accessed on Apr. 12, 2014]

Nvidia Corporation, 2011a. OpenCL Best Practices Guide. Available from https://hpc.oit.uci.edu/nvidia-doc/sdk-cuda-doc/OpenCL/doc/OpenCL_Best_Practices_Guide.pdf [Accessed on Feb. 10, 2014].

Nvidia Corporation, 2011b. OpenCL Programming Guide for the CUDA Architecture. Available from https://hpc.oit.uci.edu/nvidia-doc/sdk-cuda-doc/OpenCL/doc/OpenCL_Programming_Guide.pdf [Accessed on Feb. 10, 2014].

Paek, Y., Hoeflinger, J., Padua, D., 2002. Efficient and precise array access analysis. *ACM Trans. Progr. Lang. Syst.*, **24**(1):65-109. [doi:10.1145/509705.509708]

Pennycook, S.J., Hammond, S.D., Wright, S.A., *et al.*, 2013. An investigation of the performance portability of OpenCL. *J. Parall. Distr. Comput.*, **73**(11):1439-1450. [doi:10.1016/j.jpdc.2012.07.005]

Phothilimthana, P.M., Ansel, J., Ragan-Kelley, J., *et al.*, 2013. Portable performance on heterogeneous architectures. Proc. 18th Int. Conf. on Architechtural Support for Programming Languages and Operating Systems, p.431-444. [doi:10.1145/2451116.2451162]

Rul, S., Vandierendonck, H., D'Haene, J., *et al.*, 2010. An experimental study on performance portability of OpenCL kernels. Symp. on Application Accelerators in High Performance Computing. Available from https://biblio.ugent.be/publication/1016024

Shen, Z., Li, Z., Yew, P., 1990. An empirical study of Fortran programs for parallelizing compilers. *IEEE Trans. Parall. Distr. Syst.*, **1**(3):356-364. [doi:10.1109/71.80162]

Steven, S.M., 1997. Advanced Compiler Design and Implementation. Morgan Kaufmann, San Francisco.

Stratton, J.A., Stone, S.S., Hwu, W.M.W., 2008. MCUDA: an effective implementation of CUDA kernels for multicore CPUs. Proc. 21st Int. Workshop on Languages and Compilers for Parallel Computing, p.16-30. [doi:10.1007/978-3-540-89740-8_2]

Stratton, J.A., Grover, V., Marathe, J., *et al.*, 2010. Efficient compilation of fine-grained SPMD threaded programs for multicore CPUs. Proc. 8th Annual IEEE/ACM Int. Symp. on Code Generation and Optimization, p.111-119. [doi:10.1145/1772954.1772971]

Stratton, J.A., Kim, H., Jablin, T.B., *et al.*, 2013. Performance portability in accelerated parallel kernels. Technical Report No. IMPACT-13-01, Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, IL.

TOP500.org, 2014. TOP500 lists: November 2014. Available from http://top500.org/lists/2014/11/ [Accessed on Nov. 29, 2014].

Triolet, R., Irigoin, F., Feautrier, P., 1986. Direct parallelization of call statements. *ACM SIGPLAN Not.*, **21**(7):176-185. [doi:10.1145/13310.13329]