Journal of Zhejiang University-SCIENCE C (Computers & Electronics)
ISSN 1869-1951 (Print); ISSN 1869-196X (Online)
www.zju.edu.cn/jzus; www.springerlink.com
E-mail: jzus@zju.edu.cn

Efficient fine-grained shared buffer management for multiple OpenCL devices*

Chang-qing XUN†[1,2], Dong CHEN[1,2], Qiang LAN[1,2], Chun-yuan ZHANG[1,2]

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

([2]*State Key Laboratory of High Performance Computing, National University of Defense Technology, Changsha 410073, China*)

†E-mail: xunchangqing@nudt.edu.cn

Received Apr. 2, 2013; Revision accepted Sept. 12, 2013; Crosschecked Oct. 15, 2013

Abstract: OpenCL programming provides full code portability between different hardware platforms, and can serve as a good programming candidate for heterogeneous systems, which typically consist of a host processor and several accelerators. However, to make full use of the computing capacity of such a system, programmers are requested to manage diverse OpenCL-enabled devices explicitly, including distributing the workload between different devices and managing data transfer between multiple devices. All these tedious jobs pose a huge challenge for programmers. In this paper, a distributed shared OpenCL memory (DSOM) is presented, which relieves users of having to manage data transfer explicitly, by supporting shared buffers across devices. DSOM allocates shared buffers in the system memory and treats the on-device memory as a software managed virtual cache buffer. To support fine-grained shared buffer management, we designed a kernel parser in DSOM for buffer access range analysis. A basic modified, shared, invalid cache coherency is implemented for DSOM to maintain coherency for cache buffers. In addition, we propose a novel strategy to minimize communication cost between devices by launching each necessary data transfer as early as possible. This strategy enables overlap of data transfer with kernel execution. Our experimental results show that the applicability of our method for buffer access range analysis is good, and the efficiency of DSOM is high.

Key words: Shared buffer, OpenCL, Heterogeneous programming, Fine grained
doi:10.1631/jzus.C1300078 **Document code:** A **CLC number:** TP393

1 Introduction

Nowadays, using various accelerators to achieve high performance has become mainstream. Often, a single system is equipped with different types of processing units. This poses a huge challenge in the design of a high productivity programming environment for such heterogeneous systems.

The OpenCL standard is proposed to answer this challenge. It abstracts all processing units to form an OpenCL computing device. Many vendors, such as Intel, NVIDIA, AMD, and Apple, have released OpenCL runtime implementations for their own products. These implementations simplify application development in heterogeneous systems. For example, in the past, to exploit both the multi-core CPUs and the NVIDIA GPU residing on the same computing node, programmers had to program the CPUs with OpenMP/MPI and the GPU with CUDA. Now, programmers can use OpenCL for both purposes. With compilers for CPUs, such as MCUDA (Stratton *et al.*, 2008) and PGI CUDA (Wolfe, 2010), CUDA is another alternative for programming multi-core CPUs and NVIDIA GPUs. However, OpenCL is more widespread and can deal with many more kinds of heterogeneous systems.

Systems containing a CPU device and an accelerator device are very common. To make full use of the computing capacity, users need to dispatch the

* Project supported by the National Natural Science Foundation of China (Nos. 61033008, 61272145, 60903041, and 61103080), the Research Fund for the Doctoral Program of Higher Education of China (No. 20104307110002), the Hunan Provincial Innovation Foundation for Postgraduate (No. CX2010B028), and the Fund of Innovation in Graduate School of NUDT (Nos. B100603 and B120605), China

workloads across multiple OpenCL devices. Unfortunately, the process of using multiple devices cannot be done automatically by the binding when new devices are detected because this requires active thought from the host programmer. It requires users to think about how to distribute the load of the computation across all available devices and modify their code to apply that strategy. A more tedious job that should be considered is to explicitly manage the data transfer between multiple devices for satisfying data dependencies.

In this paper, we propose distributed shared OpenCL memory (DSOM), which provides shared memory across all OpenCL devices available in a single node. Using DSOM can greatly reduce the effort involved in developing OpenCL programs across multiple devices. Results from our experiments show that the applicability of DSOM is good and the efficiency is high.

2 Background

OpenCL is an open heterogeneous programming framework managed by the nonprofit technology consortium, the Khronos Group. It abstracts that there is one host and one or more devices. The device consists of an array of computing units, with each unit being functionally independent of the others. An OpenCL program consists of a host program and one or more OpenCL kernels. Execution is controlled by the host program, which is a plain C program that calls OpenCL application programming interface (API) functions. The role of an OpenCL kernel, implemented as a C function, is to allow the host program to offload computational work to the OpenCL device(s), executed on multiple computing units and processing elements in parallel.

A work-item is the minimum instance of concurrent execution of an OpenCL kernel. The programmer specifies the number of work-items required to launch a kernel, which is referred to as an *N*-dimensional range (NDRange). The NDRange is a one-, two-, or three-dimensional index space of work-items.

An NDRange is divided into multiple smaller, equally sized work-groups. Each work-group is assigned a unique ID, which is also an *N*-dimensional array. Fig. 1 illustrates several aspects of a two-dimensional NDRange. A work-item in a work-group is identified by a unique local ID within the work-group, treating the entire work-group as an index space, called a local index space. The global ID of a work-item can be computed with its local ID, work-group ID, and work-group size.

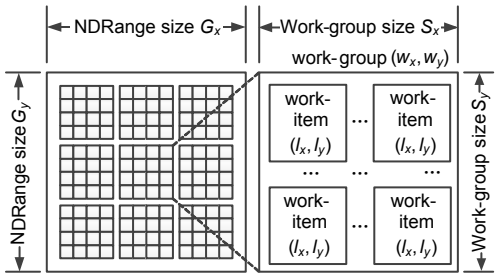

Fig. 1 The NDRange of the kernel

The host program submits memory commands (clEnqueueWriteBuffer(), clEnqueueReadBuffer()) that operate on memory objects in the device memory. The host program also submits a kernel command (clEnqueueNDRangeKernel()) that launches kernels on devices. Pointers to the memory objects are passed as arguments to a kernel that accesses the objects. A memory object in the device memory is typically a buffer object, called a buffer, for short. A buffer is valid only within a single context. Movement to and from specific devices is managed by the OpenCL runtime as necessary to satisfy data dependencies. The buffer is visible to devices on which a command queue has been created and the host has performed a read/write operation.

3 Motivation

Nowadays, systems containing multiple OpenCL devices, such as the typical CPU+GPU computing node, are widespread. Recently, many studies have shown that the capability of a CPU for processing computing-intensive tasks should be taken into account for high performance. Lee *et al.* (2010) debunked the 100x GPU vs. CPU myth by evaluating the throughput computation on the CPU and GPU. The evaluation results of SNU NPB (Seo *et al.*, 2011) are more convincing. The SNU NPB Suite is a set of NAS parallel benchmarks (NPB) implemented in C, OpenMP C, and OpenCL. The results show that the performances of OpenCL-based SNU NPB on CPU

and GPU are similar. This implies that the CPU needs to become involved in computing-intensive tasks to give full play to the performance of the whole system. CPU and GPU can both be treated as OpenCL devices, and be programmed with OpenCL C language. In addition, the system often contains more than one accelerator, such as two GPUs or a GPU and a MIC.

When using a single device, all kernel invocations are enqueued to the same command queue associated with that device. To use multiple devices, at least one command queue has to be created for each device. The command queues can either share a context or each can have its own. However, it is impossible to create a single context across devices from different vendors. This implies there is no channel to create buffers shared by them. To support concurrent execution on devices from different vendors, separate buffers, contexts, and command queues are required.

Fig. 2 provides the code fragment to demonstrate a typical usage of two devices in OpenCL. Limited by space, Fig. 2 does not show the platform queries or the creation of separate contexts, command queues, and kernels. One context per device is created here. Buffers associated with different contexts are created. In the main loop, kernels execute on two devices. Then, parts of their results are exchanged due to the data dependency between two devices. Finally, the inputs and outputs should be swapped for the next iteration. In this example, the programmers have to manage the data exchange explicitly, including controlling the accessing region of reading and writing to buffers precisely, and dealing with the synchronization. Moreover, this example code is not optimized for multiple devices, because the overlap of data transfer and kernel execution is not considered.

Although a single context and multiple buffers can be created across devices from the same vendor, the current implementations are very inefficient in shared buffer management. For example, NVIDIA OpenCL runtime can guarantee correctness only by serializing all operations on shared buffers. The evaluation of shared buffer management from NVIDIA OpenCL runtime is described in detail in Section 5.3.

As a result, in this paper we propose DSOM, which provides shared buffers for multiple OpenCL devices from single or multiple vendors, and achieves high efficiency through fine-grained shared buffer management.

```
1  BufAOld=clCreateBuffer(ContextA, ...);
2  BufANew=clCreateBuffer(ContextA, ...);
3  BufBOld=clCreateBuffer(ContextB, ...);
4  BufBNew=clCreateBuffer(ContextB, ...);
5  for (...) {
6          clSetKernelArg(KernelA, ..., BufAOld);
7          clSetKernelArg(KernelA, ..., BufANew);
8          clSetKernelArg(KernelB, ..., BufBOld);
9          clSetKernelArg(KernelB, ..., BufBNew);
10         clEnqueueKernelNDRange(..., KernelA, ...);
11         clEnqueueKernelNDRange(..., KernelB, ...);
12
13         clEnqueueReadBuffer(..., BufANew, ...);
14         clEnqueueReadBuffer(..., BufBNew, ...);
15         clBarrier();
16         clEnqueueWriteBuffer(..., BufBNew, ...);
17         clEnqueueWriteBuffer(..., BufANew, ...);
18         SWAP(BufAOld, BufANew);
19         SWAP(BufBOld, BufBNew);
20 }
```

Fig. 2 Sample code demonstrating how to use two devices

4 DSOM design and implementation

4.1 DSOM overview

DSOM is an OpenCL runtime implementation based on an open source project FreeOCL. DSOM exposes all OpenCL devices available in the system to users by invoking the corresponding vendors' runtimes (Fig. 3). This is enabled by the OpenCL installable client driver (ICD). DSOM loads all available ICDs like an ICD loader. DSOM acts as an umbrella OpenCL platform, representing the vendor platforms it finds. This umbrella platform gives us the opportunity to provide shared buffers across different vendors' devices. Users can create a single context across all available devices which are enabled by our umbrella platform, and create shared buffers associated with this context. DSOM is responsible for shared buffer management.

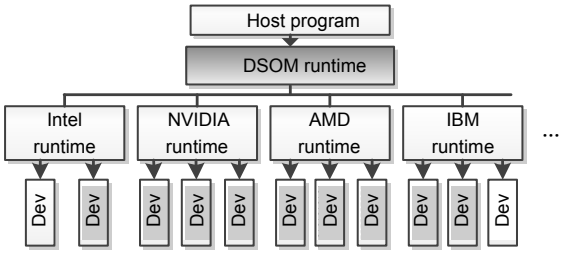

Fig. 3 Runtime structure of systems with DSOM

Fig. 4 shows a version of the code shown in Fig. 2, with DSOM. The host program creates shared

buffers associated with a single shared context, and shared buffers are passed to kernels executed on two devices as arguments. When kernels finish execution, all data exchange operations for satisfying the data dependencies for the next iteration are managed by DSOM.

```
1  BufShareOld= clCreateBuffer(ContextShare, ...);
2  BufShareNew= clCreateBuffer(ContextShare, ...);
3  for (...) {
4        clSetKernelArg(KernelA, ..., BufShareOld);
5        clSetKernelArg(KernelA, ..., BufShareNew);
6
7        clEnqueueKernelNDRange(..., KernelA, ...);
8        clEnqueueKernelNDRange(..., KernelB, ...);
9
10       SWAP(BufShareOld, BufShareNew); }
```

Fig. 4 Sample code demonstrating how to use two devices with DSOM

DSOM consists mainly of two modules: buffer manager and access range generator. The first is responsible for managing shared buffers. The second assists the buffer manager in generating buffer access ranges of kernels.

4.2 Buffer manager

4.2.1 Memory architecture

The memory architecture of DSOM is shown in Fig. 5. Shared buffers are allocated in the system memory. For each shared buffer, DSOM creates buffers in device memories on all devices available. As a result, the size of the maximal shared buffer is the size of the maximum space that can be allocated from the device which has the smallest memory of all the available devices. DSOM treats these buffers as virtual software-managed distributed caches, referred to as cache buffers in this work.

The host directly accesses shared buffers in the system memory without touching cache buffers when users invoke clEnqueueReadBuffer() or clEnqueueWriteBuffer(). Kernels executed on devices access shared buffers through cache buffers when users launch kernels by invoking clEnqueueNDRangeKernel().

Shared buffers are accessed in bulk. This differs from how the main memory is accessed by CPUs, which load and store several bytes each time. DSOM needs only to add instrumentations to every macro instruction (clEnqueueReadBuffer(), clEnqueueWriteBuffer(), and clEnqueueNDRangeKernel()) to maintain cache coherency. This enables efficient software management.

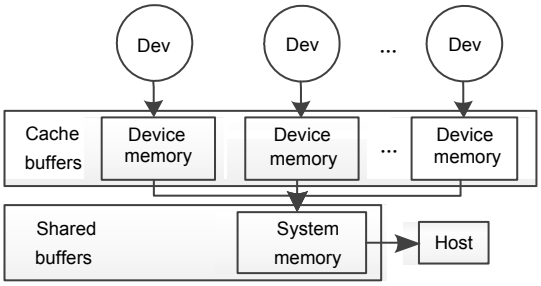

Fig. 5 Memory architecture of DSOM

A precondition of maintaining cache coherency is that DSOM has the ability to obtain the buffer access range of each macro instruction. The buffer access range of a macro instruction indicates which portion of the buffer is accessed by the macro instruction. The buffer access ranges of clEnqueueReadBuffer() and clEnqueueWriteBuffer() can easily be obtained according to the arguments passed to them. However, to obtain the buffer access ranges of kernels is nontrivial. The access range generator described in Section 4.3 solves this problem.

4.2.2 Coherency protocol

The buffer manager treats cache buffers as write-back caches, and adopts a cache coherency protocol similar to the MSI (modified, shared, invalid) protocol.

The buffer manager maintains a buffer segment table for each buffer to record the data distribution state across all the devices. Each item of this table is a buffer segment representing a portion of the corresponding buffer with the same state, described by three arguments: offset, size, and tag. Offset and size denote a segment's start address and size, respectively. Tag is a bitfield variable. The width of a tag is equal to the number of devices plus 1, which represents the host. Each bit in the tag denotes the status of the data on the specific device (the host). Value 1 means valid, and value 0 means invalid. The length of a segment can be as short as one byte, which ensures that no redundant data transfers will happen.

Fig. 6 shows the state machine of each byte in a buffer. In the system, buffer read/write requests may be invoked by the host (called host read/write, for

short) or the kernel executed on any device (called device read/write, for short). The three states are defined as follows:

1. Modified: This byte has been modified in one cache buffer, and thus is inconsistent with the system memory. The byte has to be written back to the system memory before other devices or the host can read it.

2. Shared: This byte is unmodified and exists in at least one cache buffer.

3. Invalid: This byte must be fetched from the system memory if it is to be stored in this cache buffer.

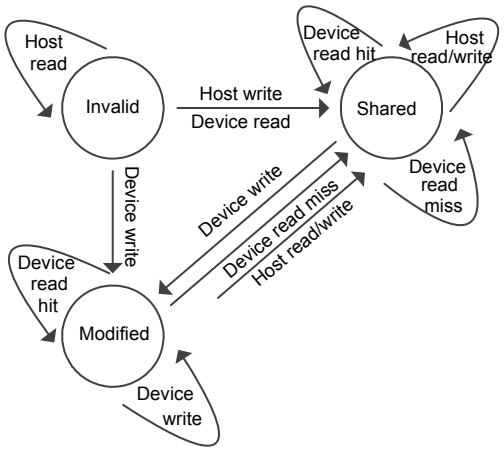

Fig. 6 Cache buffer coherency protocol

The buffer access requests from the host or devices are described by offset, size, mode (write/read), and source (host, device). The first two arguments are used to locate a data segment in a buffer as in the segment table. Mode denotes a read or write data segment in the buffer or its copy in a cache buffer, and source denotes the requester. Table 1 lists the actions (except state updating) taken by the buffer manager in response to a specific request. Buffer access requests to different segments within different states may cause different actions. After the actions are completed, the state of each segment will be updated according to the state machine (Fig. 6). A segment accessed by different requests may be split into multiple segments, and adjacent segments with the same state will be merged. Data transfer is needed only when a device miss-read or a host read/write happens. If collision occurs between two write requests to the same buffer, the runtime will execute them serially.

4.2.3 Update strategy

When a kernel finishes execution on a device, the results are modified. The update strategy relates to the writing of results back to the system memory or updating them to the cache buffers of other devices. Several common strategies are introduced in the form of an adaptive update, to improve update efficiency:

1. Batch update. Once a kernel finishes execution, the runtime writes the results back to the system memory, and updates all cache buffers. This strategy is straightforward, but may lead to a lot of redundant data transfer.

2. Lazy update. Only if data is needed by a kernel which is to be executed, does the runtime carry out the necessary data transfer to meet the needs of this kernel. The amount of data transferred is minimal. However, the data transfer operations always lag, which blocks follow-up actions. This implies that the data transfer cannot overlap with kernel executions.

3. Balance update. As a tradeoff between the batch update and the lazy update, the runtime writes results back without updating cache buffers. Only if data is needed, is the cache buffer updated.

4. Adaptive update. The runtime using the foregoing three strategies makes it hard to minimize the data transfer while launching them as early as possible. Considering that it is common to contain main loops in applications and that different iterations feature similar behaviors (including kernel execution and data transfer), we present a novel strategy called adaptive update. Adaptive update can minimize data transfer and hide latency by precise pre-data transfer.

Table 1 Actions of the buffer manager

Request	Action
Device write	Modified/shared/invalid: nothing
Device read	Modified: if read miss, write the segment back and then transfer it to the device sending request; Shared/invalid: if read miss, transfer the segment to the device sending request
Host write	Modified/shared/invalid: copy the data segment from the user space to the system memory and transfer it to all devices
Host read	Modified: write the data segment back to system memory from the device, and then copy it to the user space; Shared/invalid: copy the data segment from system memory to the user space

DSOM maintains data transfer operation histories for each kernel's buffers. All operations associated with the buffer after the kernel execution are recorded until the buffer is written by the host or another kernel. DSOM uses lazy update for each buffer initially. If a record in the operation history of a buffer occurs more than twice, DSOM will do the same operation sequence automatically once the buffer is written by the kernel the next time.

Fig. 7 shows the operation history of Stencil_Kernel's buffer newdata. WB means that the result is written back to the system memory, and UP means that DSOM updates the specific cache buffer. Histories of kernels executed on different devices are recorded separately. The operation sequences of the newdata of both kernels repeat. As a result, when the kernel next finishes, DSOM will do the same operations immediately.

```
1 StencilKernel@Device0
2   newdata:
3     1. WB(13,4), UP(1,13,4); 2. WB(13,4), UP(1,13,4); ...
4 StencilKernel@Device1
5   newdata:
6     1. WB(19,4), UP(0,19,4); 2. WB(19,4), UP(0,19,4); ...
```

Fig. 7 Operation history of StencilKernel's newdata

4.3 Access range generator

4.3.1 Theoretical basis

Our access range generator is based on a precise analysis of global array references in kernels. Array access analysis will generate the read or write access patterns to the elements or portions of arrays. It aims to obtain the knowledge of which portions or even which elements of the array are accessed by a given code segment (basic block, loop, or kernel in our case). We introduce the array access analysis in this section as the theoretical basis of our work.

Array access analysis can be largely categorized into summary or exact methods with different trade-offs of accuracy and complexity. Summary methods are approximate but can be computed quickly and economically, while exact methods are precise, but very costly in terms of computation and space storage.

Summary methods can be further divided into array sections, bounded regular sections using triplet notation, linear-constraint methods such as data access descriptors, and array region analysis. Callahan and Kennedy (1988) proposed a regular section descriptor (RSD) to describe the array access information. The RSD is a pair of the form $<A, \boldsymbol{\theta}>$, where A is the name of an array variable and $\boldsymbol{\theta}$ is a vector of different dimensional variables. Shen *et al.* (1990) used a triplet notation of lower bound, upper bound, and stride for each dimension of arrays.

Paek *et al.* (2002) designed a linear memory access descriptor (LMAD) which treats all the array accesses as one dimension. It characterizes a single index by stride, upper bound, span for each dimension, and a base offset.

Linearization is a typical kind of exact array access analysis. Triolet *et al.* (1986) were the first to use a set of linear inequalities constructed from the subscripting expressions to describe the array access region. This representation is used for dependence analysis by combining the linear inequalities associated with the references. Then, the Fourier-Motzkin elimination method (Dantzig and Curtis, 1973) is performed on the combined inequalities to determine whether there is an integer solution to see whether the dependency exists.

Pugh (1992) also used a set of linear equalities and inequalities to represent the array accesses and to treat data dependency problems as deciding whether there is an integer solution to the sets. Pugh (1992) used a more practical method named the Omega test to check the integer solution. Although both Fourier-Motzkin elimination and the Omega test have exponential complexity in the worst case, the Omega test can reduce the complexity to polynomial in more practical situations.

Considering the high processing complexity, Balasundaram and Kennedy (1989) summarized data accesses using a simplified boundary in a hyper plane form of $x_i=c$ or $x_i \pm x_j=c$. Here, x_i, x_j are coordinates of axes. This restriction makes the analyzer available to implement the region operations in polynomial complexity. It can describe the entire array precisely, a triangular region, or a single diagonal, assuming that the axes can only increase continuously. Thus, an access with a stride or an access region that is not a convex hull cannot be precisely described.

To obtain precise buffer access ranges, our buffer access range analysis is also a kind of exact array access analysis, and is similar to linearization.

4.3.2 Buffer access pattern analysis

In OpenCL C kernel programs, global array references may be contained in conditional or loop constructs, and loop variables are usually used to generate array subscripts. Moreover, work-items are executed in single program multiple data manner, so the ID of the work-item is another set of variables used to calculate array subscripts. The global ID of a work-item can be computed with its work-group ID and local ID. This means that the ID of a work-item can be represented uniquely by its work-group ID and local ID. In this study, we use a function with constraints to represent the buffer access pattern for each global array reference in kernels:

$$f(l_x,l_y,l_z,w_x,w_y,w_z,i_1,i_2,...,i_N)$$
$$\text{if}\begin{cases} c_1(l_x,l_y,l_z,w_x,w_y,w_z,i_1,i_2,...,i_N)*0, \\ c_2(l_x,l_y,l_z,w_x,w_y,w_z,i_1,i_2,...,i_N)*0, \\ \vdots \\ c_M(l_x,l_y,l_z,w_x,w_y,w_z,i_1,i_2,...,i_N)*0, \end{cases} \quad (1)$$
$$* \text{ can be } =, \neq, <, >, \leq, \text{ or } \geq,$$

where l_x, l_y, l_z represent the local IDs of work-items in the corresponding dimensions, w_x, w_y, w_z represent work-group IDs, i_1, i_2, …, i_N denote the loop variables of surrounding loops the global array has referenced, f is the subscripting function, and c_1, c_2, …, c_M are the constraints. The group size is omitted as the coefficient of group ID. The constraints are constructed from the loop bounds and conditional statements. If the constraints are not satisfied, no buffer access will occur. With this equation, the buffer access address for any iteration of loops of any work-item can be computed.

To obtain the access pattern of each global array reference, the access range generator parses the kernel when the user invokes clBuildProgram() to build the kernel. The kernel parser in the access range generator is implemented based on the LLVM compiler infrastructure (Lattner and Adve, 2004).

OpenCL C is based on C99 with limitations and extensions specialized for parallelism. The forbidden use of function pointers, recursion, bit fields, and variable-length arrays reduces the complexity of program analysis.

Our parser is facilitated by LLVM's analysis and transformation passes, which are responsible for inline function expansion, alias analysis, dead code elimination, and natural loop canonicalization. According to the information provided by LLVM's natural loop information pass, the parser obtains the loop bounds and the strides.

To obtain the symbolic subscript function and constraints, the parser symbolically executes the kernel by using KLEE (Cadar *et al.*, 2008). KLEE is a symbolic virtual machine built on top of the LLVM compiler infrastructure. The parser assigns symbols to all induction variables, local ID, group ID, global ID, local size, global size, and scalar arguments passed to the kernel. After symbolic execution, it arrives at subscript expressions in terms of these symbols for each global array reference. The symbolic path constraints of the global array reference plus the loop bounds are the constraints of the access pattern of the global array reference.

Fig. 8 shows the results of parsing StencilKernel of Stencil2D from SHOC (Danalis *et al.*, 2010). There are two buffer arguments to this kernel: data and newdata. l_x, l_y and w_x, w_y denote the local IDs and work-group IDs, respectively. ws_x, ws_y denote the numbers of work-groups. LCOLS and LROWS are macros. Line 4 describes the data access pattern of major work-items, and lines 5–12 the data access pattern of work-items on the boundary of the problem. Line 17 represents the newdata access pattern of work-items. The number of work-groups will be given when the kernel is launched. DSOM evaluates these equations to obtain the buffer access pattern of each global array reference.

4.3.3 Buffer access range generation

Access patterns are not sufficient for shared buffer management. Access ranges must be generated. The access range in DSOM is represented as a set of address segments, like offset and size. Access range generation can be performed by transforming the range of an access pattern into a set of address segments. The naive way is to traverse the domain of the subscripting function, obtain every address accessed, and then generate the access range by merging all addresses. The transversal may take a long time and the number of addresses generated may be very large, which incurs a large merging overhead.

```
1  Stencil2D/StencilKernel
2  data
3    Read:
4    LROWS*(ws_x*LCOLS+2)*w_y+LCOLS*w_x+(ws_x*LCOLS+2)*l_y+l_x+ws_x*LCOLS+3;
5    LROWS*(ws_x*LCOLS+2)*w_y+LCOLS*w_x+(ws_x*LCOLS+2)*l_y+l_x+3, l_y==0;
6    LROWS*(ws_x*LCOLS+2)*w_y+LCOLS*w_x+(ws_x*LCOLS+2)*l_y+l_x+ws_x*LCOLS+3, l_y==0;
7    LROWS*(ws_x*LCOLS+2)*w_y+LCOLS*w_x+(ws_x*LCOLS+2)*l_y+l_x+ws_x*LCOLS+2, l_x==0;
8    LROWS*(ws_x*LCOLS+2)*w_y+LCOLS*w_x+(ws_x*LCOLS+2)*l_y+l_x+ws_x*LCOLS+LCOLS+3, l_x==0;
9    LROWS*(ws_x*LCOLS+2)*w_y+LCOLS*w_x+(ws_x*LCOLS+2)*l_y+l_x, l_x==0&&l_y==0;
10 LROWS*(ws_x*LCOLS+2)*w_y+LCOLS*w_x+(ws_x*LCOLS+2)*l_y+l_x+LROWS*(ws_x*LCOLS+2), l_x==0&&l_y==0;
11 LROWS*(ws_x*LCOLS+2)*w_y+LCOLS*w_x+(ws_x*LCOLS+2)*l_y+l_x+LCOLS+1, l_x==0&&l_y==0;
12 LROWS*(ws_x*LCOLS+2)*w_y+LCOLS*w_x+(ws_x*LCOLS+2)*l_y+l_x+LROWS*(ws_x*LCOLS+2)+LCOLS+1, l_x==0&&l_y==0;
13 Write:
14 newdata
15 Read:
16 Write:
17 LROWS*(ws_x*LCOLS+2)*w_y+LCOLS*w_x+(ws_x*LCOLS+2)*l_y+l_x+ws_x*LCOLS+3;
```

Fig. 8 Buffer access patterns of StencilKernel

Designing a fast access range generation algorithm for all types of analytic functions is almost impossible. As a result, we have designed an algorithm for one of the most common situations. The algorithm works when the subscripting function is linear, the constraints are linear, and the shape of the feasible region produced by the constraints is regular.

Fig. 9 demonstrates the regular feasible regions. Assuming there are two independent variables in the constraints, the feasible regions are two-dimensional. Figs. 9a–9c show regular shapes, while Figs. 9d–9f show irregular shapes. Generally speaking, a region can be seen as regular when it is a rectangle; otherwise, it is irregular. Apparently, when the constraint is absent, the region is regular too. Access patterns complying with this condition can be represented by the affine function shown in Eq. (2):

$$f = A_1X_1 + A_2X_2 + ... + A_NX_N,\ \ X_i \in [L_i, U_i], \quad (2)$$

where f is the affine function, A_1, A_2, …, A_N are coefficients, and $[L_i, U_i]$ (i=1, 2, …, N) are domains of independent variables defined by the feasible region.

Without loss of generality, assuming coefficients are arranged in ascending order, the pseudo code of our algorithm is as shown in Fig. 10. A is the coefficient array, while L and U represent the domains. The algorithm merges continuous (overlapped) segments into one. The continuity is determined by the relationship between the coefficient and the size of the current segment. When the coefficient is less than or equal to the size of the current segment, the segments to be generated will be continuous. Otherwise, the algorithm generates a list of segments. After the access range of each global array reference is obtained, the total buffer access ranges of the kernel can be generated by merging all of them. Compared to naïve implementation, our method takes much less time.

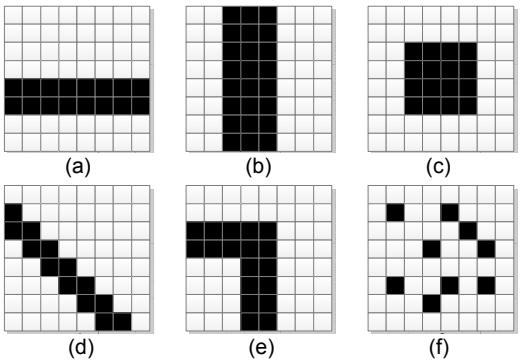

Fig. 9 Regular (a–c) and irregular (d–f) shapes of feasible regions

```
1   SegList GetSegList:
2   int A[N], L[N], U[N];
3   int Idx=0, Offset=0, Size=1;
4   while (A[Idx]<=Size && Idx<N) {
5       Offset+=L[Idx]*A[Idx];
6       Size+=(U[Idx]-L[Idx])*A[Idx];
7       Idx++; }
8   for every L[Idx],U[Idx]->L[N-1],U[N-1]
9       SegList.push(Offset+Pos,Size);
10  return SegList;
```

Fig. 10 Pseudo code of the fast buffer access range generation algorithm

Finally, taking StencilKernel as an example again, the size of the grid is 4×4, and the size of the haloed grid is 6×6. The data type is single precision. We use two devices to process the upper and lower halves of the grid, respectively. Fig. 11 shows the

buffer access ranges of kernels executed on the two devices. For simplicity, the unit used here is double word. There is overlap between the input data, which brings data dependencies between the two devices. Also, the addresses of the results are not continuous, so the buffer access range will consist of two address segments.

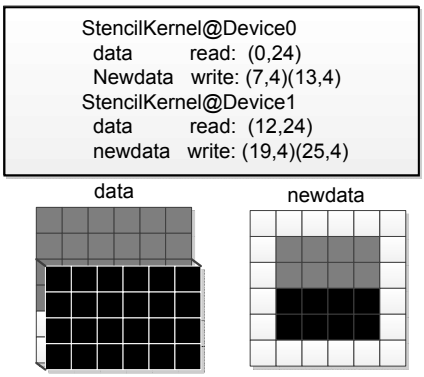

Fig. 11 Buffer access range of StencilKernel

Our access range generator cannot handle dynamic buffer accesses, also known as indirect accesses, e.g., bufa[bufb[i]]. When dynamic buffer accesses occur, the access range cannot be generated until the kernel has finished, while the buffer manager needs the buffer access range of the kernel before it can be launched. As an alternative, DSOM conservatively assumes that the buffer access range of the dynamic buffer reference is the entire buffer. For read-only buffers, this causes no error; however, for read-write or write-only buffers, when multiple devices update the buffer, DSOM needs to determine in which cache buffer the correct output resides. DSOM updates the shared buffer with the result of the comparison between the shared buffer in the system memory and the modified cache buffers in the device memories. One device typically does not modify the location that is modified by another device. Thus, for each byte of the buffer, there is only one cache buffer in which the value at the position for this byte is different from that in the shared buffer. Of course, this comparison incurs a huge overhead.

5 Evaluation

In Section 5.1 the applicability of our buffer access range generator is evaluated using a number of benchmarks. In Section 5.2 the performance of DSOM is evaluated using eight typical benchmarks. Section 5.3 compares the performance of shared buffer management using NVIDIA's OpenCL runtime.

5.1 Applicability

We used benchmarks from Parboil (Stratton *et al.*, 2012), SHOC (Danalis *et al.*, 2010), NVIDIA SDK, and SNU NPB (Seo *et al.*, 2011) to evaluate the applicability of our access range generator. Note that implementations of the same program in different benchmark suits may differ, such as the BFS from Parboil and SHOC.

Table 2 lists the experimental results. Our fast access range generation algorithm is fit for more than 85% of the chosen benchmarks. This shows good applicability of our method.

5.2 Performance

We evaluated the performance of DSOM by using a heterogeneous system consisting of two Intel Xeon E5620 quad-core CPUs, one NVIDIA Tesla C2050 GPU, one Intel MIC, plus a 16 GB DDR2 main memory. The operating system was Red Hat Enterprise Linux 5. The accelerators communicate with the CPUs via a PCI-E Gen. 2 16× bus that uses point-to-point serial links. The associated data transfer rate was 8 GB/s (full duplex).

The benchmarks used are described in Table 3. DSOM was evaluated with two kinds of hardware configurations: A, E5620+C2050; B, C2050+MIC. A includes systems consisting of CPUs and an accelerator. B includes systems with multiple accelerators. Under configuration A, CPUs can share system memory with the host, leading to less data transfer. All benchmarks use two devices in data parallel manner, and the load is balanced according to the performance ratio between the devices. It is available by first profiling applications on each device.

Figs. 12a and 12b show the amount of data transfer completed by DSOM using different update strategies, which are normalized to the amount of data using batch update. The data transfer of the adaptive update is close to that of the lazy update in most cases. Note that the data transfer of NBODY using four kinds of update strategies is the same, because all data must be transferred between devices to satisfy data dependency. Even though the amounts are the same,

Table 2 Benchmarks for applicability evaluation

Benchmark	Total	N_1	N_2	N_3*
Parboil				
SAD	18	8	0	10 (0, 10)
SGEMM	4	4	0	0 (0, 0)
STENCIL	8	0	8	0 (0, 0)
TPACF	11	5	5	1 (0, 1)
SPMV	7	0	2	5 (5, 0)
CUTCP	6	5	0	1 (1, 0)
BFS	7	0	1	6 (6, 0)
HISTO	33	19	14	0
LBM	40	38	2	0
SHOC				
MD	3	2	0	1 (1, 0)
REDUCTION	5	4	1	0
SCAN	6	1	5	0
SGEMMN	28	26	2	0
SORT	7	1	5	1 (1, 0)
SPMV	3	0	0	3 (3, 0)
STENCIL2D	12	2	10	0
FFT	5	4	1	0
BFS	52	3	0	49 (49, 0)
NVIDIA				
MATVECTMUL	18	11	7	0
BLACKSCHOLES	5	5	0	0
VECTORADD	3	0	3	0
DOTPRODUCT	9	0	9	0
DCT8x8	4	0	4	0
FDTD3D	11	5	6	0
HISTOGRAM	8	0	8	0
NBODY	10	10	0	0
MEDIANFILTER	7	0	7	0
SOBELFILTER	7	0	7	0
SNU NPB				
EP	3	0	3	0
CG	81	51	17	13 (13, 0)
MG	40	35	5	0
IS	27	18	6	3 (3, 0)
FT	19	14	5	0
BT	62	62	0	0
LU	1433	1418	15	0
SP	1939	1935	4	0

N_1 is the number of array references for which the subscripting functions are linear and no constraint exists; N_2 is the number of array references for which our fast access range generation algorithm can apply, minus N_1; N_3 is the number that our method can apply minus N_1 and N_2. * The two numbers in brackets represent the numbers of indirect buffer accesses and nonlinear subscripting functions, respectively. The average percentages of N_1 and N_2 are 40% and 45%, respectively

the performance may differ due to the different times when the data transfers are launched. There is a significant reduction for STENCIL computing based benchmarks, such as STENCIL2D and FDTD3D, because the amount of data around the border is much less than that of the whole grid.

Figs. 13a and 13b give the performances of benchmarks on DSOM using different strategies under configurations A and B, normalized to the performance of using single C2050. Obviously, the performance of batch and balance update degrades sharply due to much redundant data transfer, which is more than 50× in the worst case. Although the data transfer of lazy update is always the lowest, its performance is not the best. For benchmarks containing numerous iterations such as NBODY, STENCIL2D, FDTD3D, and CG, the lazy update suffers from a large performance loss due to the absence of overlap of data transfer and kernel execution. Performance of adaptive update is close to those of hand-coded versions for most cases under configurations A and B. The runtime overheads of SPMV and CG decrease a little, because our fast access range generation algorithm cannot be applied to a lot of their global array references. However, the runtime overhead is acceptable.

5.3 Shared buffer from NVIDIA

For comparison, we evaluated the shared buffer management from NVIDIA OpenCL runtime. Two Tesla C2050 GPUs were used, and the experimental setup is described below.

We created a 512 MB shared buffer associated with a shared context. Two command queues were created associated with two devices, respectively. The first and last bytes of the shared buffer were written by submitting commands to two command queues, respectively. Execution times of the two commands measured by using event objects were close to zero. It cost 0.36 s for the host to finish the two clEnqueueWriteBuffer() operations. The result of another experiment showed that writing and reading 512 MB on a single device cost 0.22 s and 0.14 s respectively, the sum of which is 0.36 s. This experiment indicates that after the first device writes the shared buffer, NVIDIA OpenCL runtime writes all data from the first device back to the system memory and updates all data to the second device before the second device writes the buffer (Fig. 14).

Table 3 Benchmarks for performance evaluation

Benchmark	Source	Number of kernel executions	Description
EP	SNU NPB	1	Class C, 8 589 934 592 random variates
SAD	Parboil	1	4×4, search range 33×33, input image 1920×1080
SPMV	SHOC	1	Matrix *A*, *B* 8192×8192
NBODY	NVIDIA	100	32 768 bodies, 100 iterations
STENCIL2D	SHOC	1000	Size 4 (4096×4096:16×16), 1000 iterations
FDTD3D	NVIDIA	1000	376×376×376, radius 4, 1000 time-steps
FT	SNU NPB	8	256×256×256, double
CG	SNU NPB	10 600	Class D
EP	SNU NPB	1	Class C, 8 589 934 592 random variates

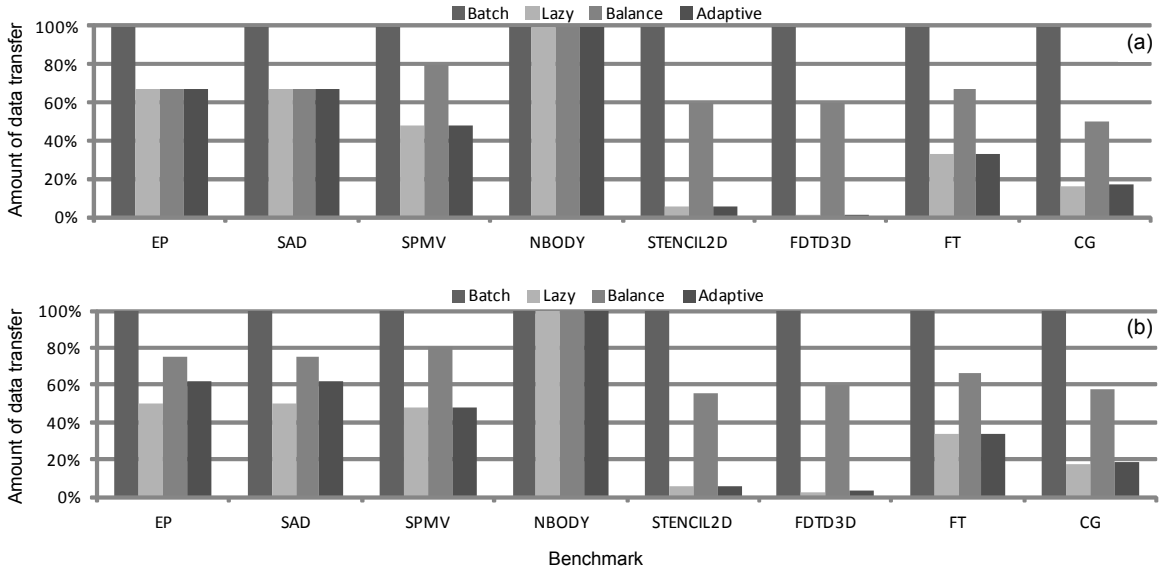

Fig. 12 The data transfer under configuration A (a) and configuration B (b)

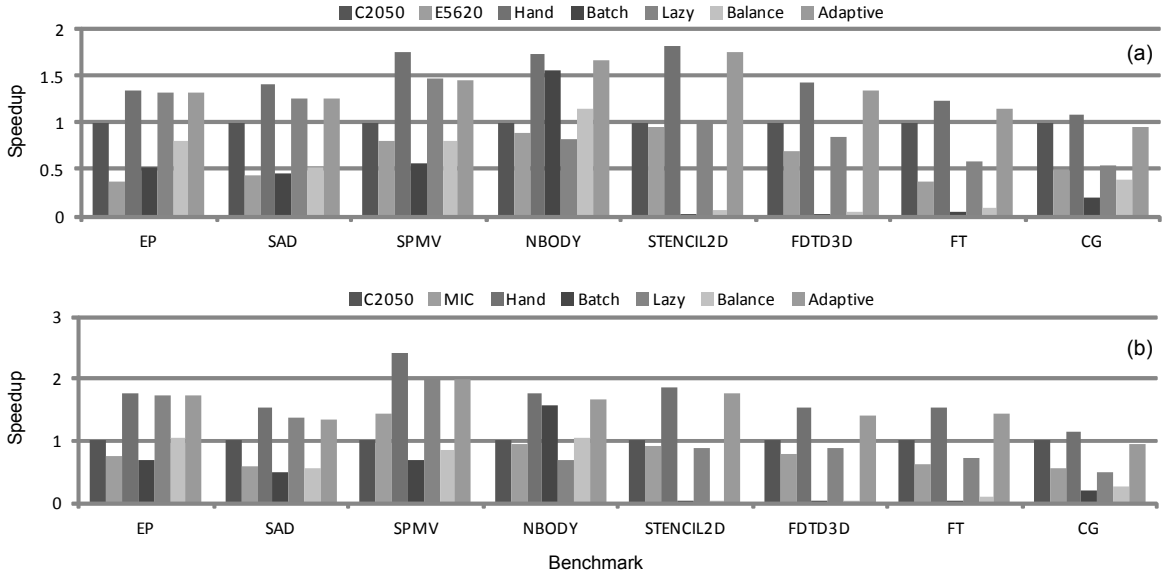

Fig. 13 Performance of DSOM under configuration A (a) and configuration B (b)

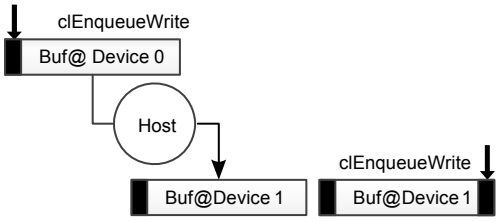

Fig. 14 Shared buffer management from NVIDIA runtime

NVIDIA OpenCL runtime serializes commands manipulating shared buffers and copies the whole buffer between devices. Apparently, even the performance of DSOM using the batch update strategy is much better than that of NVIDIA OpenCL runtime. Taking the experiments shown in Fig. 14 as an example, DSOM would only write back and update the first byte and write back the last byte at the end instead of transferring 512 MB of data.

6 Related work

Many shared memory systems have been built on physically distributed memory. Hardware distributed shared memory (DSM) systems consist of logic for coherency protocol maintenance and shared data access detection. Most hardware DSMs implement the write invalid protocol, locate data by dictionary, and manage shared data in cache line granularity (Delp *et al.*, 1988; Frank *et al.*, 1993). There are hardware DSMs relying on software too. Software in these systems usually is responsible for supporting the coherency protocol (Agarwal *et al.*, 1995) or virtual coherency protocol (Wilson *et al.*, 1993).

Software DSM systems can be implemented as compiler extensions or runtime systems. DSM systems based on compilers usually add new semantics to programming language for declaring shared data structure. Compilers generate necessary synchronization and coherency codes for each shared data access (Bal and Tanenbaum, 1988).

Runtime DSMs provide users with APIs for registering shared data structure. Software runtime systems usually use memory protection hardware for shared data structure access detection, and are responsible for maintaining coherency and synchronization. Runtime systems can be part of the operating system (Dasgupta *et al.*, 1991), or a user level library (Bershad *et al.*, 1993). The first arrangement allows the operating system to manage system resources better, but needs more implementation effort. A user library must use the system call to interact with the memory protection hardware.

There have been studies about how to support shared memory between CPUs and GPUs. This kind of shared memory can help programmers avoid explicit data transfer management. JCUDA (Yan *et al.*, 2009) introduces a semi-automatic mechanism. CGCM (Jablin *et al.*, 2011), DyManD (Jablin *et al.*, 2012), and GMAC (Gelado *et al.*, 2010) provide automatic systems. Pai *et al.* (2012) proposed a compiler-assisted fast and efficient runtime supporting shared memory based on previous work. All these systems manage shared buffers in coarse granularity. They treat any buffer accesses from the CPU or GPU as accesses to the whole buffer. For example, even when the CPU writes only a byte, the runtime will write the whole buffer back to the system memory.

Recently, Kim *et al.* (2011) proposed how to achieve a single computing device image in OpenCL for multiple GPUs by managing a virtual shared memory. They proposed a sample based buffer access range analysis method. It abstracts the subscripting functions of array accesses as affine functions of work-item ID and loop variables. However, the sample conditions of their method are very limited. Seven of eleven benchmarks used in their study for evaluation contain array references their method cannot handle. In our experiment described in Section 5.1, only the A type of array references meets the sample conditions. Sampling also involves a significant runtime overhead.

7 Conclusions

In this paper, we have presented distributed shared OpenCL memory (DSOM) to support shared memory across multiple OpenCL devices. DSOM can help users avoid tedious data transfer management between devices. DSOM allocates shared buffers in the system memory and treats the on-device memory as a software-managed virtual cache buffer. To support fine-grained shared buffer management, we designed a kernel parser in DSOM for buffer access range analysis. A basic modified, shared, invalid

cache coherency was implemented for DSOM to maintain coherency for cache buffers. Also, we proposed a novel strategy to minimize data transfer between devices, which launches each necessary data transfer as early as possible. This strategy enables the overlap of data transfer with kernel execution.

Although DSOM is also implemented as a software runtime, it is actually an implementation for OpenCL which, unlike previous systems, does not involve new APIs. This makes DSOM friendlier to users. DSOM uses a compiler-assisted runtime method to obtain the buffer access range. Most of the work is done by static analysis, which greatly reduces the runtime overhead.

Our method has three limitations. The first and major one is that our access range generator cannot handle dynamic buffer accesses, also known as indirect accesses, e.g., bufa[bufb[i]]. The second is that the condition of our fast access range generation algorithm may not be satisfied. This forces us to obtain the access range by the naive method, which incurs a large overhead. The third limitation is that the size of the largest shared buffer cannot exceed the maximum memory space of the device which has the minimal memory space among all the available devices.

References

Agarwal, A., Bianchini, R., Chaiken, D., Johnson, K.L., Kranz, D., Kubiatowicz, J., Lim, B.H., Mackenzie, K., Yeung, D., 1995. The MIT Alewife Machine: Architecture and Performance. Proc. 22nd Annual Int. Symp. on Computer Architecture, p.2-13. [doi:10.1145/223982.223985]

Bal, H.E., Tanenbaum, A.S., 1988. Distributed Programming with Shared Data. Proc. Int. Conf. on Computer Languages, p.82-91. [doi:10.1109/ICCL.1988.13046]

Balasundaram, V., Kennedy, K., 1989. A Technique for Summarizing Data Access and Its Use in Parallelism Enhancing Transformations. Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation, p.41-53. [doi:10.1145/73141.74822]

Bershad, B.N., Zekauskas, M.J., Sawdon, W.A., 1993. The Midway Distributed Shared Memory System. Compcon Spring, Digest of Papers, p.528-537. [doi:10.1109/CMPCON.1993.289730]

Cadar, C., Dunbar, D., Engler, D., 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. Proc. 8th USENIX Conf. on Operating Systems Design and Implementation, p.209-224.

Callahan, D., Kennedy, K., 1988. Analysis of interprocedural side effects in a parallel programming environment. *J. Parall. Distr. Comput.*, **5**(5):517-550. [doi:10.1016/0743-7315(88)90011-1]

Danalis, A., Marin, G., McCurdy, C., Meredith, J.S., Roth, P.C., Spafford, K., Tipparaju, V., Vetter, J.S., 2010. The Scalable Heterogeneous Computing (SHOC) Benchmark Suite. Proc. 3rd Workshop on General-Purpose Computation on Graphics Processing Units, p.63-74. [doi:10.1145/1735 688.1735702]

Dantzig, G.B., Curtis, E.B., 1973. Fourier-Motzkin elimination and its dual. *J. Comb. Theory A*, **14**(3):288-297.

Dasgupta, P., LeBlanc, R.J.Jr., Ahamad, M., Ramachandran, U., 1991. The clouds distributed operating system. *Computer*, **24**(11):34-44. [doi:10.1109/2.116849]

Delp, G., Sethi, A., Farber, D., 1988. An Analysis of Memnet—an Experiment in High-Speed Shared-Memory Local Networking. Symp. Proc. on Communications architectures and protocols, p.165-174. [doi:10.1145/52324.52342]

Frank, S., Burkhardt, H., Rothnie, J., 1993. The KSR 1: Bridging the Gap Between Shared Memory and MPPs. Compcon Spring, Digest of Papers, p.285-294. [doi:10.1109/CMPCON.1993.289682]

Gelado, I., Stone, J.E., Cabezas, J., Patel, S., Navarro, N., Hwu, W.W., 2010. An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems. Proc. 15th ASPLOS on Architectural Support for Programming Languages and Operating Systems, p.347-358. [doi:10.1145/1736020.1736059]

Jablin, T.B., Prabhu, P., Jablin, J.A., Johnson, N.P., Beard, S.R., August, D.I., 2011. Automatic CPU-GPU Communication Management and Optimization. Proc. 32nd ACM SIGPLAN Conf. on Programming Language Design and Implementation, p.142-151. [doi:10.1145/1993498.1993516]

Jablin, T.B., Jablin, J.A., Prabhu, P., Liu, F., August, D.I., 2012. Dynamically Managed Data for CPU-GPU Architectures. Proc. 10th Int. Symp. on Code Generation and Optimization, p.165-174. [doi:10.1145/2259016.2259038]

Kim, J., Kim, H., Lee, J.H., Lee, J., 2011. Achieving a Single Compute Device Image in OpenCL for Multiple GPUs. Proc. 16th ACM Symp. on Principles and Practice of Parallel Programming, p.277-288. [doi:10.1145/1941553.1941591]

Lattner, C., Adve, V., 2004. LLVM: a Compilation Framework for Lifelong Program Analysis & Transformation. Proc. Int. Symp. on Code Generation and Optimization: Feedback-Directed and Runtime Optimization, p.75-87.

Lee, V.W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A.D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., *et al.*, 2010. Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. *SIGARCH Comput. Archit. News*, **38**(3):451-460. [doi:10.1145/1816038.1816021]

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]

Pai, S., Govindarajan, R., Thazhuthaveetil, M.J., 2012. Fast and Efficient Automatic Memory Management for GPUs

Using Compiler-Assisted Runtime Coherence Scheme. Proc. 21st Int. Conf. on Parallel Architectures and Compilation Techniques, p.33-42. [doi:10.1145/2370816.2370824]

Pugh, W., 1992. A practical algorithm for exact array dependence analysis. *ACM Commun.*, **35**(8):102-114. [doi:10.1145/135226.135233]

Seo, S., Jo, G., Lee, J., 2011. Performance Characterization of the NAS Parallel Benchmarks in OpenCL. IEEE Int. Symp. on Workload Characterization, p.137-148. [doi:10.1109/IISWC.2011.6114174]

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

Stratton, J.A., Stone, S.S., Hwu, W.W., 2008. Languages and Compilers for Parallel Computing. Springer-Verlag Berlin Heidelberg, p.16-30. [doi:10.1007/978-3-540-89740-8]

Stratton, J.A., Rodrigues, C., Sung, R., Obeid, N., Chang, L.W., Anssari, N., Liu, D., Hwu, W.W., 2012. Parboil: a Revised Benchmark Suite for Scientific and Commercial Throughput Computing. IMPACT Technical Report No. IMPACT-12-01, Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign, Champaign, Illinois, USA.

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

Wilson, A.W.Jr., LaRowe, R.P.Jr., Teller, M.J., 1993. Hardware Assist for Distributed Shared Memory. Proc. 13th Int. Conf. on Distributed Computing Systems, p.246-255. [doi:10.1109/ICDCS.1993.287702]

Wolfe, M., 2010. Implementing the PGI Accelerator Model. Proc. 3rd Workshop on General-Purpose Computation on Graphics Processing Units, p.43-50. [doi:10.1145/1735688.1735697]

Yan, Y., Grossman, M., Sarkar, V., 2009. JCUDA: a Programmer-Friendly Interface for Accelerating Java Programs with CUDA. Proc. 15th Int. Euro-Par Conf. on Parallel Processing, p.887-899. [doi:10.1007/978-3-642-03869-3-82]