ABSTRACT

Although Graphics Processing Unit (GPU) is expected to be a practical high performance computing platform, current programming frameworks such as CUDA and OpenCL require large programming cost. Therefore, we are developing a new framework MESI-CUDA providing shared variables to hide low-level data management in CUDA. However, handling dynamic data structures is difficult in current MESI-CUDA because shared variables cannot be dynamically created and pointer fields are not allowed in them. Thus, we extended MESI-CUDA to remove such restrictions. Introducing dynamic management of shared variables and automatic pointer conversion on data transfer, any pointer-based dynamic data structure can be shared between the CPU and GPU with only small changes from the C code. As the results of the evaluation, pointer conversion increased the transfer time of data structures approximately 3.3 times larger in the worst case, and 1.3–2 times larger in the practical cases. Considering that non-conversion alternatives cause overhead in pointer dereferences, we regard this overhead is practical in most cases.

KEY WORDS
parallel programming language, compiler, GPU, CUDA

1 Introduction

The performance of Graphics Processing Unit (GPU) has been rapidly improved [1] surpassing CPUs. Therefore, recent GPUs are used for general high-performance computing, called General Purpose computation on Graphics Processing Unit (GPGPU) [2]. Although some standard GPGPU programming frameworks such as CUDA [3] and OpenCL [4] are provided, GPGPU programming is still very difficult and time-consuming. In such frameworks, GPU hardware architecture is shown to the user and many low-level specifications such as memory allocation and data transfer are required. Although it enables low-level hand optimization to achieve high performance, it also makes GPGPU programming difficult and reduces portability between different GPU models.

Therefore, we are developing a new framework named MESI-CUDA [5] for easier GPGPU programming. MESI-CUDA provides shared variables which can be accessed from both CPU/GPU, thus low-level specifications for data management are not needed. However, in the current version, the shared variables must be declared as static global variables and cannot contain pointer fields. Therefore, using dynamic and/or irregular data structures, such as variable-length arrays and trees, is difficult.

To support handling dynamic data structures in MESI-CUDA, we introduced several extensions such as dynamic management of shared variables and automatic pointer conversion. As a result, any pointer-based dynamic data structures can be created and shared between the CPU and GPU with minimum difference compared with conventional C programming.

This paper is organized as follows: Section 2 gives a brief introduction of the GPU, CUDA and MESI-CUDA. Section 3 shows the outline of our new extensions. The detail of language extensions and their implementation are illustrated in Section 4, 5. In Section 6, we show the evaluation results, and in Section 7, we discuss related works. In Section 8, we state the conclusion.

2 Background

2.1 GPU Architecture

GPU is a collection of streaming multiprocessors (SM), each consisting of certain number of processor cores. Although GPU cores lack many complex features such as branch prediction and execution reordering, their number is very large. Thus, the total performance of cores on a single GPU outperforms that of a single CPU.

The GPU has hierarchical memories. Each core has registers and a local memory, called General Purpose computation on Graphics Processing Unit (GPGPU) [2]. Although some standard GPGPU programming frameworks such as CUDA [3] and OpenCL [4] are provided, GPGPU programming is still very difficult and time-consuming. In such frameworks, GPU hardware architecture is shown to the user and many low-level specifications such as memory allocation and data transfer are required. Although it enables low-level hand optimization to achieve high performance, it also makes GPGPU programming difficult and reduces portability between different GPU models.

Therefore, we are developing a new framework named MESI-CUDA [5] for easier GPGPU programming. MESI-CUDA provides shared variables which can be accessed from both CPU/GPU, thus low-level specifications for data management are not needed. However, in the current version, the shared variables must be declared as static global variables and cannot contain pointer fields. Therefore, using dynamic and/or irregular data structures, such as variable-length arrays and trees, is difficult.

To support handling dynamic data structures in MESI-CUDA, we introduced several extensions such as dynamic management of shared variables and automatic pointer conversion. As a result, any pointer-based dynamic data structures can be created and shared between the CPU and GPU with minimum difference compared with conventional C programming.

This paper is organized as follows: Section 2 gives a brief introduction of the GPU, CUDA and MESI-CUDA. Section 3 shows the outline of our new extensions. The detail of language extensions and their implementation are illustrated in Section 4, 5. In Section 6, we show the evaluation results, and in Section 7, we discuss related works. In Section 8, we state the conclusion.
CUDA (Compute Unified Device Architecture) [3, 7, 8] is a GPGPU programming framework using extended C/C++. Fig. 1 is an example of CUDA program performing two parallel computations; computing each element of the 1-dimensional array b from the corresponding element of the array a, and updating every element of the 2-dimensional array x. CUDA-specific code is shown in bold font.

In CUDA, CPU and GPU are called host and device, respectively. Functions, declared with the __device__ or __global__ qualifier, are called kernel functions and executed on the device (Fig. 1 l. 6–13). The other functions (called host functions in this paper) are executed on the host. Any host function can invoke a __global__ kernel function specifying numbers of thread blocks and threads per block (l. 29, 31). Kernel threads are created and run the kernel function on the GPU in parallel. Using block/thread IDs to specify the data element to process, data parallel processing can be described easily (l. 7–8, 11–12).

The local variables in kernel functions are compiled as registers/local memory accesses and are private for each thread. On the other hand, the GPU global memory, physically on the off-chip memory (called device memory), can be accessed from all threads. Explicit data transfers between the host/device memories are required to share values between host/kernel functions (Fig. 1 l. 27–28, 30, 32).

Data transfers from host to device and device to host are called download and readback transfer, respectively. The memory areas on the device used as sources/destinations of transfers are allocated/deallocated using CUDA functions (l. 23–25, 39–41).

Kernel executions and data transfers can be overlapped using asynchronous transfers. For this purpose, streams must be created (l. 17–19), and the source/destination memory areas on the host must be allocated/deallocated using CUDA functions to page-lock them (l. 20–22, 36–38). Specifying a stream to each kernel invocation and data transfer, the invocations/transfers in the same stream are performed in the assignment order, while a kernel execution and a data transfer of different streams can be performed in parallel. In this example, the download transfer of the array x and the readback transfer of the array a (l. 28, 30) can be overlapped with the kernel executions process_int and process_float (Fig. 1 l. 29, 31), respectively. Synchronizations are also necessary to assure asynchronous readback transfer is completed. (l. 33–34).

As the sample shows, explicit data transfers and execution/transfer scheduling in CUDA programming enable strong hand-optimization but also requires large programming effort. Unified Virtual Addressing enables using single address space but explicit data transfer is still necessary. Mapped memory enables direct access of page-locked host memory from the device but may be inefficient because implicit transfer occurs for each access to the same data.

### 2.3 MESI-CUDA

#### 2.3.1 Outline

To achieve easier GPGPU programming, we are developing a new framework called MESI-CUDA [5], hiding low-level features of CUDA from the user.

The parallel execution feature is same as CUDA; kernel functions are explicitly invoked as threads. The large difference is that MESI-CUDA provides shared variables which can be accessed on both host and device. Therefore, explicit memory management and data transfer are

---

1. The line numbers in Fig. 1, 2, 3, 7 are for explanation purposes only.

2. Kernel functions can declare shared variables between kernel threads in the same block, using __shared__ qualifier. Because we do not use such variables in this paper, the word shared variables refers to MESI-CUDA shared variables.
functions can refer to a variable either directly by name or by a pointer argument to parameters. Using shared variables, explicit memory allocation/deallocation on host/device memory, data transfer, synchronization, and scheduling using streams, are eliminated. Thus, the code is largely simplified without low-level specifications. Multi-dimensional arrays can be accessed using indices, without annoying offset expressions. (Fig. 1 l. 12, Fig. 2 l. 14).

2.3.3 Implementation

Because the host and device do not share the physical memory, the shared-memory model of MESI-CUDA is a kind of virtual shared memory (VSM). Generic VSM implementation is difficult on the device because it has no hardware supportinng VSM and OS is not running. Thus, the shared-memory model is implemented in the application layer; a shared variable is implemented as two memory areas on the host/device, and static code such as memory allocation and data transfer is inserted to emulate the shared variable. The MESI-CUDA compiler mecc is a source-to-source compiler. It translates user’s MESI-CUDA program into CUDA program in the following steps, and compiles the generated program using the CUDA compiler.

2.3.4 Current Issue

Current shared variables has the following restrictions:

1. Must be statically declared as global variables; cannot be created dynamically and the array size cannot be determined at runtime.
2. Pointer fields are not allowed; cannot be any pointer type or struct type with pointer members.

Compared with the dynamic malloc/free interface in CUDA, static declaration of shared variables in MESI-CUDA enables simpler programming for the user with automatic memory management and multi-dimensional array indices. Furthermore, better compiler optimization can be expected because static analysis, such as access range analysis, is easier in array-based programs. However, practical program often needs to determine the array size at runtime.

These restrictions also makes handling dynamic and/or irregular data structures, such as lists, trees, and graphs, difficult and inefficient. Such structures require references between data elements. Although using physical pointers as the references is the most easy and efficient way in C programming, such pointer-based structure cannot be stored in MESI-CUDA shared variables because the pointer fields are prohibited. The lack of dynamic shared variable creation also makes it difficult sharing dynamic structures in which memories are often allocated on demands.

Fig. 3 is an example C program handling typical dynamic data structure; a variable-length text buffer which can store a large text file with unknown number of lines on memory.

The program uses a variable-length array supported in C99, GCC, etc.
the overhead of fine-grained memory allocation, a certain amount of memory buffer, typed buffer_t, is dynamically allocated on demand. Assuming the text is read-only in the program, each line with different length is packed in an array of pointers like lines[n] in the program, each line with different length is packed in an array of pointers like lines[n].

Assuming processing each line has no dependency, it can be easily parallelized on GPU by transferring the data structure and each kernel thread processing lines[i] can be easily parallelized on GPU by transferring the data structure and each kernel thread processing lines[i] with the facility to point arbitrary data is practically sufficient. Using the value as relative address from the buffer member p, assigned to point a certain field in another shared variable t on the host. The logical/physical structures are shown in Fig. 5 center and left, respectively. Dereference using p value on the device is incorrect because it is the absolute address on the host (Fig. 5 right (a)). Using the value as relative address from p is also incorrect because the distance of s and t may be different on the host and device (b). The address of t on the device and the offset must be used for the correct dereference (c). Thus, a host memory pointer ph, pointing any location within a shared variable s, can be converted to the corresponding device memory pointer pd by the following equation:

\[ pd = _{dev}s + ph - _{host}s \]
4 Language Extensions

4.1 Shared Variables with Static Scope

In addition to the global variables, __global__ qualifier is also permitted for local variables in host functions. Such a variable works as a temporal shared variable and destroyed on exiting the block of its declaration. Local variables in kernel functions cannot be declared as shared variables.

Because such variables have the same scope and lifetime with C auto variables, they have some restrictions. First, kernel functions are out of their scope thus cannot access them by name. Therefore, the references must be passed as the arguments of kernel invocations or pointer fields of other accessible shared variables. Second, the kernel function accessing them must be invoked within the block of their declarations because they are destroyed after exiting the block.

For the arrays declared as shared variables in the block, the size expression may include variables like C99.

4.2 Dynamic Creation of Shared Variables

To allow the user controlling numbers and lifetime of shared variables, we also provide traditional malloc/free style interface in Table 1.

Table 1. Functions for Dynamic Shared Variables

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cudaMemcpy</td>
<td>Copy data from host to device</td>
</tr>
<tr>
<td>cudaMemcpyAsync</td>
<td>Copy data from host to device with</td>
</tr>
<tr>
<td>cudaMemcpyAsyncAsync</td>
<td>completion notification</td>
</tr>
<tr>
<td>cudaMemcpyToSymbol</td>
<td>Copy data to device with symbol</td>
</tr>
<tr>
<td>cudaFree</td>
<td>Free host memory</td>
</tr>
<tr>
<td>cudaFreeAsync</td>
<td>Free host memory asynchronously</td>
</tr>
<tr>
<td>cudaFreeAsyncAsync</td>
<td>Free host memory asynchronously</td>
</tr>
<tr>
<td>cudaFreeSymbol</td>
<td>Free host memory with symbol</td>
</tr>
<tr>
<td>cudaHostAlloc</td>
<td>Allocate host memory</td>
</tr>
<tr>
<td>cudaHostAllocAsync</td>
<td>Allocate host memory asynchronously</td>
</tr>
<tr>
<td>cudaHostAllocSymbol</td>
<td>Allocate host memory with symbol</td>
</tr>
<tr>
<td>cudaMalloc</td>
<td>Allocate device memory</td>
</tr>
<tr>
<td>cudaMallocAsync</td>
<td>Allocate device memory asynchronously</td>
</tr>
<tr>
<td>cudaMallocAsyncAsync</td>
<td>Allocate device memory asynchronously</td>
</tr>
<tr>
<td>cudaMallocSymbol</td>
<td>Allocate device memory with symbol</td>
</tr>
<tr>
<td>cudaMemAlloc</td>
<td>Allocate memory (host/device)</td>
</tr>
<tr>
<td>cudaMemAllocAsync</td>
<td>Allocate memory (host/device)</td>
</tr>
<tr>
<td>cudaMemAllocAsyncAsync</td>
<td>Allocate memory (host/device)</td>
</tr>
<tr>
<td>cudaMemAllocSymbol</td>
<td>Allocate memory (host/device)</td>
</tr>
</tbody>
</table>

4.2.1 Pointer Fields in Shared Variables

Pointer fields of arbitrary types are permitted in shared variables if the following restrictions are satisfied:

- All types and relative addresses must be static and known at compile time. Thus, union cannot be used.
- Unless the pointer field type is void *, the field must be NULL value or point within any shared variable.

If a pointer assignment occurs in a host function, the value of the pointer field is an address in the host memory. On the transfer, the value is converted to the corresponding address in the device memory. Similar conversion also occurs on the readback transfer. Thus, pointer operators, such as * and & can be used in host/kernel functions and links in data structures implemented using pointers are maintained.

Using this extension, C code handling pointer-based data structures can be easily ported to MESSI-CUDA. For example, Fig. 3 program can be parallelized as Fig. 7. Modifications are shown in bold font and the omitted lines are not changed. Practically, process_each_line() will need to be tuned for GPU, but sharing the dynamic data structure requires only small changes on memory allocation.
5 Implementation

5.1 Implementation Issues

To implement the language extensions described in Section 4, several issues must be considered.

First, mecc cannot statically embed all code for data transfer like the current implementation. The number of shared variables and their sizes may be unknown at compile time. Furthermore, shared variables needed to transfer may be dynamically changed because the pointer fields enable indirect accesses to other shared variables.

Second, pointer conversion requires the host/device addresses of the pointed shared variable. Such addresses also need to be determined at runtime.

In the following sections, the case of download transfers is illustrated. Readback transfers can be implemented similarly, thus we discuss the difference in Section 5.4.

5.2 Runtime Library

To handle dynamic factors of our extensions, we implemented a small runtime library to provide required functions. The runtime library implements a shared variable table (SVT) which stores attributes of each existing shared variable (Fig. 8). Table 2 is the major runtime functions.

5.2.1 Generating Typed Conversion Functions

Although the creation of shared variables, data transfer, and pointer conversion may occur dynamically, all types of shared variables statically appear in the declaration of either shared variables or pointers given to cudaMallocGlobal(). For each such type $t$, typed conversion functions $\text{conv}_t\_h2d()$ and $\text{conv}_t\_d2h()$ are generated if $t$ has any pointer fields. The pointer fields in each type can be easily specified as struct members or array elements. Thus, the functions convert every pointer fields calling $\text{conv}_t\_h2d()$ and $\text{conv}_t\_d2h()$. For example, typed conversion functions in Fig. 9 are generated for Fig. 3 program.

5.2.2 Creation/Destruction of Shared Variables

The runtime library function $\text{malloc}_\_g()$ is a low-level function for shared variable creation. It allocates areas of $\text{type}\_\text{size} \times \text{count}$ on both host/device memories, makes $hp$ and $dp$ to point them, and creates a SVT entry.

The declaration of a static shared variable $s_i$ of type $t_s$ is replaced with the declaration of two $t_s$ pointers $\text{host}_s$ and $\text{dev}_s$. The code for creating $s_i$ is generated as follows:

$\text{malloc}_\_g(\&\text{host}_s, \&\text{dev}_s, \text{sizeof}(t_s), n, h2d, d2h)$

If $s_i$ is a scalar variable, $n = 1$. Otherwise, $n$ is the size of the array $s_i$. If $t_s$ has any pointer fields, $h2d$ and $d2h$ are the function pointers $\text{conv}_t\_h2d$ and $\text{conv}_t\_d2h$, respectively. Otherwise, NULL is given. The destruction code of $s_i$ is also generated which deallocates memory areas and deletes the corresponding SVT entry.

For a local shared variable, creation/destruction code is inserted at the beginning and the end of the block of its declaration. For statements such as break and return, the code destroying all local shared variables of the nested inner blocks is inserted.

$\text{cudaMallocGlobal}(\&p, \text{count})$ is also replaced with a $\text{malloc}_\_g()$ call. If $p$ is declared as a $t_d$ pointer, $\text{sizeof}(t_d)$ is given as the $\text{type}\_\text{size}$ argument and function pointers to $\text{conv}_t\_h2d$ and $\text{conv}_t\_d2h$ are given as $h2d$ and $d2h$ arguments, respectively. $\text{cudaFreeGlobal}()$ is simply replaced with the destruction code described above.

5.2.3 Data Transfer

Even in our new implementation, mecc statically extracts the target of data transfer and inserts transfer code. However, a call of generic transfer function $\text{trans}_t\_h2d()$ is inserted instead of directly calling $\text{cudaMemcpy}()$.

The pseudo code of $\text{trans}_t\_h2d()$ is shown in Fig. 10. Given a pointer $hp$ to a shared variable $s_i$ to transfer, SVT is looked up to find the entry $e$ of $s_i$. If the typed conversion functions are registered, all fields of $s_i$ is duplicated and the pointer conversion is applied to the copy. Finally, transfer is performed using $\text{cudaMemcpy}()$.

$\text{Asyncronous transfer is not implemented yet for the new extensions.}$
trans

h2d()

h2d()

host
cudaMemcpy(e->ds, src, e->type_size * e->count, cudaMemcpyHostToDevice);
set status of e as queued;
*p = e->ds + *p - e->hs;
}

Figure 10. Pseudo Code of _trans_h2d()

void _trans_h2d(char *hp){
    _shared_entry_t * e = find SVT entry of hp;
    char *src = e->hs;
    if (status of e == done) return;
    if (e->h2d != NULL){
        memcpy(buf, e->hs);
        for (int i = 0 ; i < e->count ; i++){
            e->h2d(cp);
            cp += e->type_size;
        }
        src = buf;
    }
    cudaMemcpy(e->ds, src, e->type_size*e->count,
                cudaMemcpyHostToDevice);
    set status of e as done;
    while ((e = dequeue from transfer queue) != NULL)
        _trans_h2d(e->hs);
}

Figure 11. Pseudo Code of _conv_h2d()

As shown in Fig 9 example, the typed conversion functions called from _trans_h2d() call _conv_h2d() to convert each pointer field. The pseudo code of _conv_h2d() is shown in Fig. 11. Given a pointer to a pointer field p in a shared variable, SVT is looked up to find the entry of the pointed shared variable. Then, the pointer value of the device memory is computed and stored in p.

As discussed in Section 5.1, shared variables indirectly accessible also must be transferred. To handle cyclic links and avoid redundant transfers of the same variable, we enqueue every pointed shared variable newly found during the pointer conversion (Fig. 11) and transfer them after the current target is transferred (Fig. 10).

5.3 Optimization

While the execution cost of most code in _trans_h2d() is ignorable compared with the data transfer overhead, the cost of pointer conversion has a large impact because it is applied to every pointer field.

Optimization of SVT lookup Because each pointer conversion requires a SVT lookup, speeding up the lookup is important. We implemented SVT as a sorted array and use binary search for the lookup. Although SVT entry is dynamically added/deleted, the cost of maintaining the sorted array is acceptable considering SVT size is not so large.

We also implemented lookup using a hash. However, SVT lookup is not an injective function; for any host pointer p satisfying e->hs ≤ p ≤ e->he, the entry e must be returned. Using address/variable size as the hash key, any pointer p pointing within a variable can find the entry checking two keys at maximum.

Another technique to reduce lookup time is introducing an entry cache for the most recently used entry; after a lookup, the obtained entry is stored as the previous entry. When the next lookup occurs, the previous entry is tested first and the lookup is skipped if it is the desired entry. Because pointer fields often points within the same variable, this simple cache is expected to reduce the lookup time.

Optimization of Pointer Conversion Destructive conversion also reduces the overhead. If the host function does not read the shared variable after the transfer or readback transfer occurs before the read, the duplication in _conv_h2d() can be omitted overwriting the original.

Conversion on the device may also reduce the overhead, because the conversion can be always destructive and can be parallelized. While SVT is expected to be small to be stored in the shared memory, reading/updating the pointer field in the device memory will be the bottleneck.

Our current scheme transfers all indirectly-accessible shared variables, thus unnecessary transfer may occur. Using static analysis, the conversion of a pointer field p and the recursive transfer of the pointed variable can be statically eliminated if any code accessing p does not occur in kernel functions. However, it will be ineffective for recursive structures such as trees. More dynamic approach is introducing dirty bit to SVT entry and mecc inserting code to set the corresponding bit on each write to shared variables. Because the bit is per shared variable, only single set operation is needed for a series of deterministic write on the same shared variable. Thus, we expect the overhead can be reduced enough by the optimization.

Low-level Optimizations Because the pointer conversion enormously repeats small procedure, low-level optimizations are effective. Forcing inlining all functions called below typed conversion functions halved the overhead. Replacing slow operations was also effective; changing hash size from a prime number to $2^k$ and modulo hash function to $k$-bit mask largely reduced SVT lookup time.

5.4 Readback Transfer

Basically, readback transfers can be implemented similar to download transfers; for the transfer target s, values are transferred from _dev_s to _host_s, then reverse pointer conversion is applied. One small difference is that the pointer conversion on the host can always be destructive because the device has its copy on the device memory.

Another difference is that the receiver, not the sender, must determine the transfer target. Therefore, recursive transfers caused by pointers will disturb overlapping of execution/transfer. Recursive readback transfers cannot be queued using the stream, because completing the preceding transfer and its pointer conversion determines the tar-
Figure 12. Conversion Overhead in Quadtree Transfer

Table 3. Evaluation Environment

<table>
<thead>
<tr>
<th>PC</th>
<th>Intel Core i7 930 (2.8GHz), 6GB memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>Tesla C1060</td>
</tr>
<tr>
<td>Software</td>
<td>CentOS 5.8 x86_64, CUDA 4.0 V0.2.1221, GCC 4.1.2</td>
</tr>
</tbody>
</table>
Using cache largely reduces the relative time to 1.8–3.4 in both SVT lookup method. However, cache hit ratio depends on the data structure. In this benchmark, building the tree in preorder causes certain locality on pointer links. The cache will be more ineffective on irregular trees.

Destructive conversion always reduces the overhead skipping the duplication. Although it cannot be always applied, the relative transfer time was suppressed below 2.0.

Nevertheless the overhead is quite large, it can be regarded as nearly the worst case. Pointer-based data structures always have non-pointer fields, because pointers only represent links. Pointer conversion is not applied to such fields, thus the overhead is smaller for the structures of larger non-pointer fields. The relative transfer time using hash and cache is decreased from 3.3 to 1.9 when the value size $S_v$ is increased from 4 to 256 bytes with $S_v=256K$ (d).

6.3 Variable-Length Text Buffer

As an example of practical data structures, we also evaluated the transfer time of variable-length text buffer in Fig 3. As a huge but realistic input text file, we concatenated the contents of “The King James Bible” [9] 100 times (9.5M lines, 414MBbytes). The results are shown in Fig. 13. The $x$ axis is the text bytes stored in a text buffer (BUF_SIZE), and the $y$ axis is the relative transfer time compared with the transfer without pointer conversion.

Pointer conversion is not applied to the text itself and cache hit ratio is high because successive elements of lines point the same buffer. Therefore, the relative transfer time was 1.3–1.6 using hash and cache, and suppressed to 1.2–1.4 also using destructive conversion.

Nevertheless the evaluation results may seem showing large overhead even optimization techniques are used, we regard it is practical enough in most cases because of two reasons. First, avoiding pointer conversion requires alternative techniques such as implementing logical pointers, causing overhead during the execution. Second, GPGPU programs need to be computation-intensive to achieve high performance and the transfer time is not a dominant factor in many programs, thus the overhead of our scheme will not have large impact. However, further evaluation is needed measuring total execution time of practical applications.

7 Related Work

Many schemes have been proposed as the better GPGPU programming environments. Major approaches can be categorized into (a) the auto-parallelizing compilers [10], (b) translators from other parallel languages [11, 12], (c) automatic generation of low-level code for conventional framework [13, 14], and (d) template libraries [15, 16, 17].

MESI-CUDA is classified in the category (c). Our purpose is to obtain a practical GPGPU programming framework; easier than CUDA, but still can achieve high performance. Therefore, we introduced minimum language extension into CUDA instead of mapping other language into GPU architecture like category (a), (b).

The library approach (d) is also practical. The implementation cost is low without modifying the compiler. Template-base libraries have high reusability and the overhead is small due to the static polymorphism. Thus, many major works, such as Thrust [16] and Glift [17], adopt this approach to handle dynamic/irregular data structures. However, we aim to achieve high performance using static analysis to generate optimized code, whereas the library approach cannot introduce new optimization to the compiler. While template libraries provide abstract data structures hiding the implementation, our approach only hides physical pointer values and the implementation of data structures is left for the user. Thus, any C data structure can be shared with GPU porting C code easily.

Rthreads [18] and Compile-Time Virtualisation (CTV) [19] also adopt application-level virtual shared memory models. Compared with these generic approaches, our shared variable model is specific to GPGPU but the description of low-level synchronization is eliminated.

Converting pointer-based structures to the equivalent structures in different space is known as pointer swizzling [20] in the field of object-oriented database. Typical pointer swizzling converts between the structures of persistent objects in the database and the equivalent structures of objects on memory. Thus, pointer conversion is between some logical references, such as object ID, and physical pointers. Sun RPC [21] also enables passing pointer-based dynamic data structures on remote procedure calls. To support hosts of different architectures, it uses a machine-independent data description and encoding protocol called XDR. Data types can be defined using a C-like language which is compiled into C data types and stub code.

Compared with these conversion schemes, our pointer conversion is much simpler using addresses and offsets, because it is memory-to-memory on the same host. However, reducing the conversion overhead is very important, since data transfer over PCIe is much faster than disc access and data transfer over network. Another advantage of our approach is that we directly handle C pointers, thus introducing other descriptions, such as XDR, is not needed.
8 Conclusion

For the practical use of GPGPU for high performance computing, the programming environment needs to achieve both execution performance and low programming cost. Thus, we are developing a programming framework MESI-CUDA. Providing shared variables, MESI-CUDA hides CUDA’s low-level specifications such as memory allocation and data transfer from the user. However, handling dynamic data structures is difficult because dynamic creation and pointer fields are not allowed for shared variables.

In this paper, we presented new extensions of MESI-CUDA to support dynamic data structures. Our scheme introduces dynamic management of shared variables and automatic pointer conversion on data transfer, thus pointer-based dynamic data structures can be easily shared. The advantage of our approach is that any pointer-based dynamic data structure can be shared between the CPU and GPU, only by modifying variable declarations and dynamic memory allocation functions.

As the results of the evaluation, the overhead of pointer conversion increased the transfer time of the whole data structure to approximately 3.3 times larger in the worst case. In practical case, it was suppressed to 1.3–2 times larger. We regard it is not ignorable but practical in most cases. However, we expect applying more optimizations including many techniques mentioned in Section 5.3 will further reduce the overhead. The evaluation using practical programs is also a future work.

Acknowledgement

This work was supported by JSPS KAKENHI Grant Number 24500060.

References