MACC: An OpenACC Transpiler for Automatic Multi-GPU Use

Kazuaki Matsumura[1,2(✉)], Mitsuhisa Sato[3,4], Taisuke Boku[4], Artur Podobas[1], and Satoshi Matsuoka[1,2]

[1] Tokyo Institute of Technology, Tokyo, Japan
{matsumura.k.ak,podobas.a.aa,matsu}@m.titech.ac.jp
[2] AIST-Tokyo Tech Real World Big-Data Computation Open Innovation Laboratory, Tokyo, Japan
[3] RIKEN Advanced Institute for Computational Science, Kobe, Japan
msato@riken.jp
[4] University of Tsukuba, Tsukuba, Japan
taisuke@cs.tsukuba.ac.jp

Abstract. Graphics Processing Units (GPUs) perform the majority of computations in state-of-the-art supercomputers. Programming these GPUs is often assisted using a programming model such as (amongst others) the directive-driven OpenACC. Unfortunately, OpenACC (and other similar models) are incapable of automatically targeting and distributing work across several GPUs, which decreases productivity and forces needless manual labor upon programmers. We propose a method that enables OpenACC applications to target multi-GPU. Workload distribution, data transfer and inter-GPU communication (including modern GPU-to-GPU links) are automatically and transparently handled by our compiler with no user intervention and no changes to the program code. Our method leverages existing OpenMP and OpenACC backends, ensuring easy integration into existing HPC infrastructure. Empirically we quantify performance gains and losses in our data coherence method compared to similar approaches, and also show that our approach can compete with the performance of hand-written MPI code.

Keywords: Programming language · Compiler · Multi-GPU

1 Introduction

Graphics Processors Units (GPUs) are the workhorse of modern, state-of-the-art, supercomputers. Each node in a supercomputer node often consists of several GPUs, each carrying its own distributed memory and each being capable of executing asynchronously to one another. Due to GPU's high performance and compute-to-power ratio (FLOPs/Watt), modern supercomputers such as TSUBAME3.0 [1], DGX SATURNV [2] and the upcoming SUMMIT [3] include multiple GPUs per supercomputing node.

Programming GPUs has historically been done through low-level programming languages (often derivatives or dialects of C) such as CUDA [4] and

© The Author(s) 2018
R. Yokota and W. Wu (Eds.): SCFA 2018, LNCS 10776, pp. 109–127, 2018.
https://doi.org/10.1007/978-3-319-69953-0_7

OpenCL [5]. Here the programmer is responsible for both creating the program code, and — in cases where multiple GPUs are involved — orchestrating the concurrent execution of multiple GPUs; an often non-trivial task.

A better (and arguably more portable) way is to use compiler directives to indicate sources of potential parallelism in the application. A compiler can then use these directives to abstract the complex architectural details away from the programmer and instead automatically generated device-specific program code. One example of such directive-driven approach is OpenACC [6] and OpenMP [7].

While models such as OpenACC increase productivity through raised programming abstraction, they are currently limited in targeting only a single GPU device. The user is still responsible for manually orchestrating the multi-GPU execution.

We propose a method to enable OpenACC-annotated applications to exploit multiple GPUs. We implemented a source-to-source compiler (*transpiler*) that analyzes and optimizes OpenACC applications. Our transpiler is *transparent* to the user— kernel scheduling, data-movement and inter-GPU communication (including the recent GPU-to-GPU links) are automatically done.

Our contributions in short:

(1) A transpiler that extends the OpenACC programming model to allow applications to seamlessly use multiple GPUs.
(2) A novel communication algorithm that preserves data coherency across GPUs by extracting source-code information.
(3) An empirical evaluation of above contributions using well-known HPC benchmark, positioning the performance against hand-written MPI code and the recent Unified Memory abstraction layer.

The remaining of this paper is organized as follows. Section 2 discusses related work. Section 3 provides an overview of OpenACC. Section 4 describes our proposed method. Section 5 describes our experimental methodology. In Sect. 6, we evaluate our transpiler. Finally, Sect. 7 concludes this paper.

2 Related Work

NVIDIA provides Unified Memory [8], which allows multiple NVIDIA GPUs to share the global address space between each other. Unified Memory recently supports coherence through the NVLink interconnect [9], and allows GPUs to effortlessly communicate between each other. Unlike Unified Memory, which is very architecture dependent, our approach is more general and oblivious of which accelerator is being targeted as long as compilers' OpenACC backend supports it. Moreover, our method is able to accelerate GPU-to-GPU communications using GPU interconnects. We also see performance benefits using our method as compared to Unified Memory in Sect. 6.

Komada et al. [10] used a compiler to distribute OpenACC fairly across GPUs and execute them in parallel. Their approach is to divide loop iterations into chunks of equal size and also keep these chunks coherent across different GPUs.

Their coherence mechanism is similar to that of Unified Memory, except that the chunk size can be changed manually by the user and the chunks are prepared for each array. Unlike Komada et al., we focus on identifying where communication needs to happen between GPUs through data-flow analysis inside the transpiler.

Rameshekar et al. [11] propose to execute parts of application (written in C) on multiple GPUs by analyzing loops using the polyhedral model. The polyhedral compilation precisely detects necessary communication between GPUs using superpositions of fine regions and a buffer management. However, their approach is only applicable to loops with affine iteration and array accesses. We (unlike Rameshekar et al.) build and extend upon OpenACC, which allows us to have more information regarding the sources of parallelism, increasing generality as long as the application uses OpenACC. However, we complement their study and show how the polyhedral compilation can enhance our method in certain cases.

HYDRA [12] is a compiler system for distributed environments that use a single GPU per node. We both share a similar system of determining communication patterns between GPUs. Unlike HYDRA, which takes as input simple directives and generates a distributed application, our method leverages OpenACC and OpenMP and hence focuses parallelism within a single "shared" node. The output of our transpiler uses both OpenACC and OpenMP, and thus can further use existing OpenACC and OpenMP profiling tools to further improve performance. In evaluation, we compare hand-written MPI code with our transpiled code.

Scogland et al. [13] combine a well-designed task-based runtime with directive-driven model to facilitate efficient work-sharing in heterogeneous systems. They provide new directives that help to identify data dependency. We consider extending our work to leverage existing task-based runtime systems to perform the load-balancing.

Xu et al. [14] present new directives to extend OpenACC to support multiple accelerators. The proposal is based on an evaluation using the hybrid model of OpenACC and OpenMP.

Accelerate [15] is a purely-functional domain-specific language for array processing. Accelerate has a potential to utilize multi-GPU [16].

Also, programming models targeting accelerator clusters are proposed [17,18]. These models provide explicit functions to distribute computations over multiple accelerators.

3 Overview of OpenACC

Introduced in 2011, OpenACC aims to bridge accelerator programmability gaps by leveraging compiler directives. Rather than programming with vendor-specific languages, the programmers instead focus on exposing available parallelism in his/her source-code. A compiler uses these directives to automatically generate device-specific application code.

3.1 Execution Model

Programmers are responsible for specifying which regions of the OpenACC application are offloaded onto accelerators. A programmer, when using OpenACC to parallelize the application, must enclose regions to-be accelerated on the device using the `parallel` and `kernels` construct. Each construct is specified by its directive.

A region enclosed by the `parallel` construct is called a *parallel region* and will be executed on an accelerator. Controlling the granularity of loops found within the parallel region is done using the `loop` construct. The `loop` construct allows various controls of computation, including collective operations (by `reduction` clause), loop-carried dependency (by `independent` or `seq` clause), coarse-grained parallelism (called `gang`), fine-grained parallelism (called `worker`), and SIMD-level parallelism (called `vector`). These granularities represent the naturally forms of parallelism found in modern accelerators.

A nested loop executed parallely on an accelerator is called *kernel*. A region enclosed by the `kernels` construct will treat each of the subsequent loops in the region as accelerator kernels.

3.2 Memory Model

Prior to execution, accelerators with local memory (often called *device memory*) must receive the data they operate on from the host. Similarly, when accelerators finish computing using some data as the result, that data must be transferred back to the host.

OpenACC allows data on the device to be explicitly controlled using the `data` construct. Any region enclosed by the `data` construct will place specified data on-to the accelerator and insert proper transfers between host and accelerator according to specified clauses. Although OpenACC can automatically deduce where to place data, using the `data` clause is often encouraged for the prevention of incomplete or inefficient data transfers. A `present` clause of the `data` construct indicates that the data already is defined in the region, to prevent duplicated data transfers.

OpenACC defines the `update` directive that allows programmers to change data shared between host and accelerator. In the case of `update`, the OpenACC runtime system can reflect the recent changes to the variable on both the accelerator and the host.

3.3 A Motivational Example

We show an example which utilizes multi-GPU. In OpenACC, several efforts are required even to distribute a simple kernel over multiple GPUs.

OpenACC Single-GPU Example. We illustrate the needless and error-prone (manual) effort of re-purposing an OpenACC application into targeting multi-GPUs using a simple vector-multiplication example, seen in Fig. 1(a). For clarity

(a) Single-GPU Use

```
#pragma acc data\
  copyout(x[0:N]) present(y)
#pragma acc kernels
for (int i = 0; i < N; i++)
  x[i] = y[i] * y[i];
```

(b) Multi-GPU Use

```
numgpus = acc_get_num_devices(DEVICE_TYPE);
#pragma omp parallel num_threads(numgpus)
{
  int tnum = omp_get_thread_num();
  int sz = N / numgpus;
  int lb = sz * tnum; int ub = lb + sz;
  acc_set_device_num(tnum, DEVICE_TYPE);
#pragma acc data copyout(x[lb:sz]) present(y)
#pragma acc kernels
  for (int i = lb; i < ub; i++)
  x[i] = y[i] * y[i];
}
```

Fig. 1. Two examples illustrating the difference in OpenACC code that targets (a) single-GPU use, and (b) multi-GPU use through mixed OpenMP/OpenACC

purposes, we leave out non-trivial optimizations such as deducing (and minimizing) inter-GPU communication and coherence; such optimizations further complicate the manual re-targeting process (which is automatically handled by our proposed transpiler).

Figure 1(a) shows OpenACC directives in for a simple vector-multiplication. The programmer — knowing that the loop is inherently parallel — annotates the region with a `kernels` construct and also informs the compiler that he expects the host's memory to be updated after the loop has finished (`copyout`). The compiler will use these directives to offload the parallelized loop onto a single device.

OpenACC Multi-GPU Example. OpenACC only provides functionality for exposing parallelism onto a single accelerator. To use multiple GPUs, the programmer must manually orchestrate the execution, distribution and data transfers between the GPUs. This includes ensuring that data are coherent between the GPUs that work on similar sets of data.

Figure 1(b) shows the earlier vector multiplication example but with manual augmentation for multi-GPU execution. Here we use OpenMP (a related directive-driven model) to launch a team of threads and have each thread computation which subset of the loop's iteration-space it should execute. Finally, each thread encounters the OpenACC directives, which each launch the kernel onto the earlier (`acc_set_device_num()`) specified accelerator.

We can see that there is significant effort of code rewriting between Fig. 1(a) and (b), which further motivates the need for our work. Furthermore, in case of real applications, data dependencies between multiple OpenACC kernels must be considered.

4 MACC: A OpenACC Transpiler for Multi-GPU Use

We present MACC — a transpiler that eliminates the effort of using OpenACC in multi-GPU environments. Our transpiler allows OpenACC (which traditionally targets a single GPU) to run on multiple GPUs without changing the source code. Our approach is to source-to-source transform the OpenACC application into post source-code that exploits both OpenACC and OpenMP.

The operations performed by MACC can be condensed down to three steps:

(1) The source code is parsed and OpenACC directive understood and abbreviated notation (e.g. `"parallel loop"`, `"kernels copy(..)"`) flattened. Tightly-nested (which has just one loop inside except for the innermost) or `loop`-directive-specified loops within a `kernels` construct are transformed to use an OpenACC `parallel` construct (and `loop` directives with the `reduction` clause if any collective operation is found and the `independent` clause if our basic checker statically finds no loop-carried dependency between iterations).
(2) Array reference expressions are extracted using our data-flow analysis (described in Subsect. 4.3) and code to dynamically find data dependencies at runtime are constructed.
(3) MACC outputs post source-code, which effectively is hybrid OpenMP and OpenACC version of the original code but capable of multi-GPU execution (described in Subsect. 4.4).

The final output can then be compiled with any compiler supporting OpenACC and OpenMP. Our compiler pass is generic and thus untied to any specific compiler backend. Host-to/from-GPU and GPU-to-GPU communication in the post source-code are automatically generated based on our communication algorithm to resolve data dependencies, leveraging shared host memory and GPU interconnects (described in Subsect. 4.2).

We have deliberately chosen to have the transpiled source-code use a hybrid OpenACC and OpenMP approach. There are no formal requirements behind using OpenMP and our methodology can be extended to use less abstract models such as POSIX Threads. However, by leveraging OpenMP we can more easily use existing infrastructure (such as) to debug or analyze the performance. Furthermore, future work of ours includes code multi-versioning to enable hybrid execution of single-/multi-GPU and general purpose processor accelerators; it is here we expect our design decision to bear fruit as OpenMP traditionally (prior to the 4.0 accelerator directives) target general purpose processors.

4.1 Execution on Multi-GPU Environments

Since OpenACC primarily target loop-level parallelism of the outermost loop, we need to make sure to avoid data dependencies on the memory accesses happening between processors executing the parallelized loop.

In MACC, we divide and distribute the outermost loop of OpenACC kernels equally for each GPU. We only enable multiple GPUs when all writes in the

kernel are affine with respect to the loop counters and the write-section does not intersect with the write-section of other GPUs; we fall back on single GPU execution if the state condition does not hold. Switching between single- and multi-GPU execution occurs at runtime. Also, the number of GPUs used can dynamically be decided and changed, leaving room for autotuners.

4.2 Generating Host-to/from-GPU and GPU-to-GPU Communcation Patterns

Understanding what data-regions a GPU will work on is crucial when parallelizing loop constructs to execute over multiple GPUs. A too optimistic approach can fail to fully include all data, leading to incorrect execution; a too pessimistic approach will on the other hand lead to unnecessary transfer and maintenance overheads.

Algorithm 1. Generate copyin

1: Create DIRTY
2: **for** each gpu $i \in$ GPUs **do**
3: Communicate specified array from Host to GPU i
4: DIRTY[i].valid $\leftarrow$ false
5: **end for**

Identifying the data-regions needed for the GPUs is difficult because the order of kernel executions is dynamically decided. Therefore, we replicate Host-to-GPU communications according to `data` constructs (`copyin`) for all GPUs (Algorithm 1).

When multiple GPUs are used, it is important to resolve the data dependencies between the GPUs because each GPU is (often) a discrete device with its own distributed memory. We have adopted Kwon et al. [19] 's method (from distributed-memory programming) to identify the necessary communication across GPUs. Our implementation calculates the section of the read (called `USE`) and the write (called `DEF`) for each combination of parallel regions, GPUs and data (arrays). We apply data-flow analysis (described in Subsect. 4.3) to derive necessary information.

Before each execution of parallel regions, we compute the necessary communication among GPUs based on the superpositions of the calculated sections; after that, we update the section (called `DIRTY`) for each combination of GPUs and data/arrays. Here, we call the section whose master is a GPU, as `DIRTY`. Algorithm 2 describes this process.

All sections contain an upper- and a lower-bound. Communication between GPUs is performed either through host memory (CPU-to-GPU) or — if supported — using the interconnected (GPU-to-GPU or *P2P*). MACC also removes any duplicated transfers in order to reduce the amount of communication needed.

Algorithm 2. Generate communications before an execution of a parallel region

1: **for** each gpu $i \in$ GPUs **do**
2: **if** DIRTY[i] $\cap$ DEF[i] $= \emptyset$ or $\exists$ d $\in$ (DEF $\setminus$ DEF[i]); DIRTY[i] $\cap$ d $\neq \emptyset$ **then**
3: Communiate DIRTY[i] from GPU i to Host and all other GPUs
4: DIRTY[i] $\leftarrow \emptyset$
5: **else if** P2P IS ENABLED **then**
6: **for** each gpu $j \in$ GPUs; $j \neq i$ **do**
7: COMM[j] $\leftarrow$ DIRTY[i] $\cap$ USE[j]
8: Communicate COMM[j] from GPU i to GPU j
9: **end for**
10: **else**
11: **for** each gpu $j \in$ GPUs; $j \neq i$ **do**
12: COMM[j] $\leftarrow$ DIRTY[i] $\cap$ USE[j]
13: **end for**
14: GHs $\leftarrow$ BIND(COMM) /* Optimize GPU-to-Host communication */
15: Communicate GHs from GPU i to Host
16: **if** $\exists$ gh $\in$ GHs; DIRTY[i] $\subset$ gh **then**
17: DIRTY[i].valid $\leftarrow$ false
18: **end if**
19: **for** each gpu $j \in$ GPUs; $j \neq i$ **do**
20: Communiate COMM[j] from Host to GPU j
21: **end for**
22: **end if**
23: DIRTY[i] $\leftarrow$ DIRTY[i] $\cup$ DEF[i]
24: **end for**

Algorithm 3. Generate copyout

1: **for** each gpu $i \in$ GPUs **do**
2: Communicate DIRTY[i] from GPU i to Host
3: **end for**
4: Delete DIRTY

Algorithm 4. Generate `update` of GPU-to-Host

1: US $\leftarrow$ the update section
2: **for** each gpu $i \in$ GPUs **do**
3: Communicate DIRTY[i] $\cap$ US from GPU i to Host
4: **end for**

When encountering GPU-to-Host communication of the `data` constructs (`copyout`), only the data constructs that fail meeting coherence are transferred. Hence, each GPU will execute a copyout transfer of its own `DIRTY` section (Algorithm 3).

When encountering a `update` directive for Host-to-GPU communication, the host will update all GPUs with the new data. If `update` directive for a GPU-to-Host communication is encountered, only the sections overlapping the `DIRTY`-section (Algorithm 4) will be transferred.

4.3 Data-Flow Analysis

MACC uses data-flow analysis to identify `USE` and `DEF` sections of parallel regions. Data-flow analysis is invoked on every parallel region to extract array references/indices for read and write accesses. Array references are composed of constants and loop iterator variables, as well as variables defined outside the parallel region. Note that MACC only synthesize the code for automatically analyzing the `USE` and `DEF` sections; the actual analysis is performed at runtime during execution before every parallel region.

During data-flow analysis, we collect array references/indices as well as extracting variables that are defined or overwritten in the parallel region. We iteratively analyze the parallel region to account for all paths of the control-flow graph as long as the collected array references/indices change (so-called Iterative Data-Flow Analysis [20]).

In MACC, we do these through the following two steps:

(1) We transform source-code into static single assignment form (SSA [20]).
(2) Array indices are collected and all variables (except the loop counter) are extracted.

A variable represented by an expression using the variable itself and other values, is considered to have indefinite value. A section that is calculated using a non-affine index, when regarding variables other than loop counters as constants, or indefinite value will force the section to pessimistically contain the whole target array. For indices that are affine we can compute the sections by substituting upper- and lower-limits of the affected loop counters into them owing to convexity.

In the final generated post source-code we execute the parallel region on multi-GPU when:

(1) all write accesses for each array are affine and definite,
(2) the outermost loop of the kernel in the parallel region is dividable (which statically-or-dynamically has an affine range and statically has no loop-carried dependency), and
(3) the write-sections are not duplicated among GPUs.

Single-GPU execution is invoked if above conditions do not hold.

4.4 Output Formats

In MACC, each OpenACC directive will be transformed to use a combination of OpenMP and OpenACC. To realize the communication generation (described in Subsect. 4.2), we wrap the existing OpenACC communication routines around our own. These wrapper maintains the `DIRTY` section for each the {GPU:Array} combinations and also generates the communications. We use and link-against vendor-supplied libraries (in this case NVIDIA CUDA libraries) only when P2P (GPU-to-GPU) communication is available.

The `data` construct and `update` directive are converted into corresponding concurrent versions using OpenMP's `parallel` construct, seen in Fig. 2(a) and (b) respectively. When transforming `data`-constructs, MACC will always append appropriate `present` clause to parallel sections within `data` construct in order to specify that the data are already on the GPUs.

Figure 2(c) shows how we transpile OpenACC's `parallel` constructs. We start by identifying the loop ranges by calculating `USE` and `DEF` sections. Once we know the loop ranges, we spawn one OpenMP thread for each GPU device. Each thread then continues to generate the needed communication based on the algorithm described in Subsect. 4.2; a barrier is inserted to synchronize all threads before entering the compute part. Finally, the parallel region is executed by all the threads and the GPU they orchestrate.

If the parallel region satisfies the conditions described in Subsect. 4.3, the outermost loop is divided and the execution is distributed to each GPU. An actual example of the transpilating is shown in Fig. 3. At section calculations,

(a) `data` **construct**

```
#pragma acc data copy(x[0:N])
{ /* ... */ }
```

```
#pragma omp parallel num_threads(NUMGPUS)
{
  copyin_routine(omp_get_thread_num(),x,0,N);
}
{ /* ... */ }
#pragma omp parallel num_threads(NUMGPUS)
{
  copyout_routine(omp_get_thread_num(),x);
}
```

(b) `update` **directive**

```
#pragma acc update host(x[a:b])
```

```
#pragma omp parallel num_threads(NUMGPUS)
{
  int tid = omp_get_thread_num();
  update_host_routine(tid, x, a, b);
}
```

(c) `parallel` **construct**

```
#pragma acc parallel
{ /* PARALLEL REGION */ }
```

```
if (/* sections are changed */)
{ /* recalculate sections */ }
#pragma omp parallel num_threads(NUMGPUS)
{
  int tnum = omp_get_thread_num();
  set_gpu_num(tnum);
  set_data_section(/* ... */);
#pragma omp barrier
#pragma acc parallel
  { /* PARALLEL REGION */ }
}
```

Fig. 2. This mapping illustrating how MACC transpiles each directive including construct (left) into combined OpenMP/OpenACC code (right)

(a) Original Code

```
#pragma acc parallel loop gang reduction (+ : sum) present(a)
for (i = X; i < Y; i++) {
  sum += a[i + p];
}                                              PARALLEL REGION
```

(b) Transpiled Code

```
{
  static int sections_are_changed = 1;
  sections_are_changed =
    (sections_are_changed || last_p != p || last_X != X || last_Y != Y);

  if (sections_are_changed) {
    section_are_changed = 0; last_p = p; last_X = X; last_Y = Y;

    calc_loop_sections(loop_sections, X, Y,
                       1 /* increment */,
                       0 /* whether to execute when X==Y */);

    init_uses(a_uses, 1 /* affine */); init_defs(a_defs, 0 /* none */);
    for (i = 0; i < NUMGPUS; i++) {
      update_section(a_uses[i], loop_sections[i].lb + p);
      update_section(a_uses[i], loop_sections[i].ub + p);
    }

    if (is_overlapping(a_defs)) {
      /* reconstruct for single GPU execution */
    }
  }
}                                          SECTION CALCULATION
#pragma omp parallel num_threads(NUMGPUS) reduction (+ : sum) private (i)
{
  int tnum = omp_get_thread_num();
  set_gpu_num(tnum);

  set_data_section(tnum, a, a_uses, a_defs);
#pragma omp barrier                              COMMUNICATION

#pragma acc parallel present(a)
#pragma acc loop gang reduction (+ : sum)
  for(i = loop_sections[tnum].lb; i <= loop_sections[tnum].ub; i++) {
    sum += a[i + p];
  }                                            PARALLEL REGION
}
```

Fig. 3. This actual example showing how OpenACC kernel is transpiled and where MACC inserts section calculation, communication and parallel region

the last result is used as long as all component values are not changed from the last calculation. Since variables defined outside the parallel region are shared among threads, our multi-GPU execution can overwrite them. As an exception, variables used as loop counters are duplicated on every thread by `private` clauses of OpenMP. Reductions are firstly calculated for each GPU by OpenACC, then the overall results are computed among threads by OpenMP's reduction clause.

4.5 Polyhedral Extension

It is important for transpilers (such as the one we present) to easily be integrated into existing tool-chains, frameworks and compilations techniques.

To show this, we show that MACC can be complemented with other techniques to further the performance benefits. One such extension we support is the *polyhedral compilation*, here materialized using PLUTO [21].

By using PLUTO prior to MACC invocation, we can automatically split OpenACC kernels through loop fission, and thus extract the parallelism that MACC can exploit over multiple GPUs by just appending the directive of the `kernels` construct.

5 Experimental Methodology

5.1 Implementation

MACC was implemented as a prototype coupled with the Omni [22] compiler's C frontend/backend using XcodeML [23]. Currently, MACC only support OpenACC applications written using the C language (and not, for example, FORTRAN). This is a minor limitation (and resolving it is more of an engineering effort), since the methods and techniques introduced in this paper is general enough to not be tied to any specific programming language.

MACC also requires that arrays copied to GPU devices are contiguous, as multidimensional arrays are converted into singledimensional arrays. The size of coarse-grain parallelism `gang` specified in input is divided for each GPU equally, and other parallelisms (`worker, vector`) are kept.

We evaluated the three versions of MACC: baseline which conducts GPU-to-GPU communications through shared host memory, MACC with NVIDIA Unified Memory (UM) which entrusts data coherency to UM, and MACC with P2P. We also leveraged PLUTO together with MACC where applicable (for one of the benchmarks). We compared transpiled code against the original version of the benchmark, and also against MPI + ACC versions that we prepared by appending OpenACC directives to the official MPI code.

Each benchmark was executed 10 times and we used the average to represent the performance. We report performance with respect to computational performance or execution time (OP/s, FLOP/s or seconds depending on benchmark) as well as speedup over the original version ($speedup = t_{\texttt{original}}/t_{\texttt{multi-GPU}}$).

5.2 Topology Options

MACC allows the user to specify a topology mapping, which dictates what OpenMP thread handles what GPU device. Such information can be crucial in system with non-homogeneous links between the GPU devices.

While not the primary focus of the present study, we found that by reassigning the thread-to-GPU mapping on P2P-enabled GPUs (NVIDIA's to be precise), we can get a performance increase. The topology mapping is conveyed to MACC through an environmental variable.

5.3 Environment

We evaluated the performance using a single node on the new TSUBAME3.0 supercomputer at the Global Scientific Information and Computing Center (GSIC), Tokyo Institute of Technology. A node in the TSUBAME3.0 supercomputer contains 4 NVIDIA P100 GPUs [24]. The GPUs are interconnected in an all-to-all fashion using NVLink technology; note, however, that the links are heterogeneous and different: two of the links ($\texttt{GPU}_0 \leftrightarrow \texttt{GPU}_2$, $\texttt{GPU}_1 \leftrightarrow \texttt{GPU}_3$) have 80 GB/s bidirectional bandwidth and the remaining links have 40 GB/s bidirectional bandwidth. Each TSUBAME 3 node also contain two CPUs (Intel Xeon E5-2680v4) with a total of 28 general-purpose x86-64 cores. Table 1 provides more detailed system information.

For all experiments, we used PGI Compiler version 17.10 and NVIDIA CUDA version 9.0. Inside MACC we have OpenMP threads each orchestrate individual GPU; more specifically, the mapping is as follows: $\{\texttt{thread}_0, \texttt{GPU}_0\}$, $\{\texttt{thread}_1, \texttt{GPU}_2\}$, $\{\texttt{thread}_2, \texttt{GPU}_1\}$, $\{\texttt{thread}_3, \texttt{GPU}_3\}$. Our mapping follows the heterogeneous links of the GPU interconnect.

PGI Compiler supports UM only for dynamically allocated memory. We implemented an extension in MACC to force static allocations to be dynamically allocated.

Table 1. Specifications of TSUBAME3.0

CPU	Intel Xeon E5-2680 V4 (Broadwell-EP, 14 core, 2.4GHz) × 2
CPU memory	256 GiB
GPU	NVIDIA Tesla P100 for NVLink-Optimized Servers × 4
GPU memory	16 GB HBM2@732 GB/s / GPU
OS	SUSE Linux Enterprise Server 12 SP1
Compiler	PGI Compiler 17.10
Compiler option	-O4 -fastsse -ta=tesla,cuda9.0 -acc -mp -Munsafe_par_align -Mmovnt -mcmodel=medium
CUDA	CUDA 9.0

5.4 Benchmarks

We selected four benchmarks to evaluate our transpiler.

Himeno Benchmark [25] is a benchmark which solves a 3-D Poisson's equation by Jacobi method. We created an OpenACC version of this benchmark by adding directives to the official code. In the baseline OpenACC version of code, the calculation part and the substitution part are executed on an accelerator, and they are repeated for certain time-steps. Each part consists of three loops, and they are tightly nested and have no loop-carried dependency between the iteration. The baseline code collapses three-nested loops into single loops. The loop has SIMD parallelism `vector` (the size is 256), and coarse-grain parallelism `gang` and fine-grain parallelism `worker` are not specified (their size is decided by compiler's runtime). In multi-GPU execution by MACC, halo communications between GPUs are inserted for each execution of the calculation part. As the problem size, we chose Large $(i \times j \times k) = (256 \times 256 \times 512)$.

NAS Parallel Benchmarks CG (NPB-CG) [26] is a benchmark which calculates the minimum eigenvalue of a sparse symmetric positive matrix. We used an OpenACC version of this benchmark, created by Xu et al. [27]. In the baseline program, sparse matrix multiplications (SpMV) and eigenvalue calculations are offloaded to an accelerator and they are repeated for certain times. The SpMV execution applies `gang` to the iteration over each row, and applies both `worker` and `vector` to the inner loop over each non-zero element of the row. The `gang` size is equal to the row size of the matrix, `worker` size 4, and `vector` size 32. In multi-GPU execution by MACC, a communication of the row size from each GPU to all other GPUs are generated for each execution of SpVM. We chose the problem Class C ($\texttt{rowsize} = 150,000$) for the evaluation.

For comparison, we prepared hand-coded MPI code (MPI+ACC) of Himeno Benchmark and NPB-CG by adding OpenACC directives to the official MPI code in same parallelization style as above. Regarding only NPB-CG, the official code is written by Fortran and the number of process is limited to N^2.

The Scalable Heterogeneous Computing Benchmark Suite MD (SHOC-MD) [28] is a benchmark which performs an N-body computation (the Lennard-Jones potential from molecular dynamics) using a neighbor-list algorithm. In evaluation, the N-body computation against 73,728 atoms is performed 512 times in double precision. There is no data dependency between the 512 computations. We obtained SHOC-MD's OpenACC source-code from the official repository. The source-code has one OpenACC kernel which is just enclosed by the `kernels` construct to execute one-time N-body computation.

PolyBench/ACC [29] is a polyhedral benchmark suite targeting accelerators. We chose the covariance code (COVAR) from PolyBench/ACC to test MACC's polyhedral extension. We used the OpenACC source-code of the official repository. In evaluation, a large covariance matrix $(16,384 \times 16,384)$ is calculated. The one compute-bound kernel is originally not parallelizable by MACC due to a symmetric-matrix creation while calculating covariances.

6 Results

The performance with respect to the number of GPUs is seen in Fig. 4. Overall, we see that our transpiler do provide the means to increase the performance of the application by multi-GPU. However, depending on the application characteristics, different behaviors are observed.

Using MACC, we measured the performance of two data coherence implementations: our own described in Sect. 4 (with and without P2P support), and using NVIDIA Unified Memory (UM). Despite the fact that UM internally use P2P, we find that our implementation without P2P outperforms it in all but one case; enabling P2P in our implementation always executes faster than UM.

We also find that for applications whose data patterns require plenty inter-GPU communication (e.g. NPB-CG and in-parts the Himeno benchmark), enabling the P2P acceleration inside MACC can have significant performance increases. For applications that MACC's transform is inadequate, we show that we can leverage other optimization techniques (the polyhedral compilation in case of this evaluation) to overcome bottlenecks otherwise hard to deal with. Finally, we also find that MACC can automatically generate multi-GPU code that is performance comparable to handwritten MPI+OpenACC code.

The remaining section continues to in-detail provide the analysis on a per-benchmark basis.

Himeno Benchmark. The speedup for the Himeno benchmark is shown in Fig. 4 (a). Using only the MACC compiler yields an average speedup of 2.55× with four GPUs activated without P2P; further performance can be reached by allowing MACC to exploit the P2P communication between GPUs, which can yield an up-to 32.1% performance increase (3.36× speedup) on average. Using UM yields performance similar to MACC without the P2P addition. One MPI version ($N \times 1 \times 1$) which divides the i, j and k dimension by N, 1 and 1 respectively as with baseline MACC, and another MPI version ($2 \times 2 \times 1$) which is minimizing communications between processes yield slightly lower performance (−12.6% and −1.6% respectively) compared to baseline MACC.

NPB-CG. Performance results for the NPB-CG is shown in Fig. 4(b). We see that MACC (with and without P2P) scales with the given GPUs, yielding a 2.16× and 1.54× performance speedup respectively. MACC with P2P enabled scales stably better (19.9%, 34.9% and 40.9% when using $2 \sim 4$ GPUs respectively). Direct data transfer between MPI processes incurs a large 72.9% overhead when using 4 GPUs, which limits scalability; the average increase in performance experienced by the MPI version is 1.09×. Note that our version that use UM experience a *loss* of application performance (negative scaling) when increasing the number of GPUs. We found that UM thashes the memory (by thrashes we mean that it frequent causes page faults and page migrations), which leads to large performance losses.

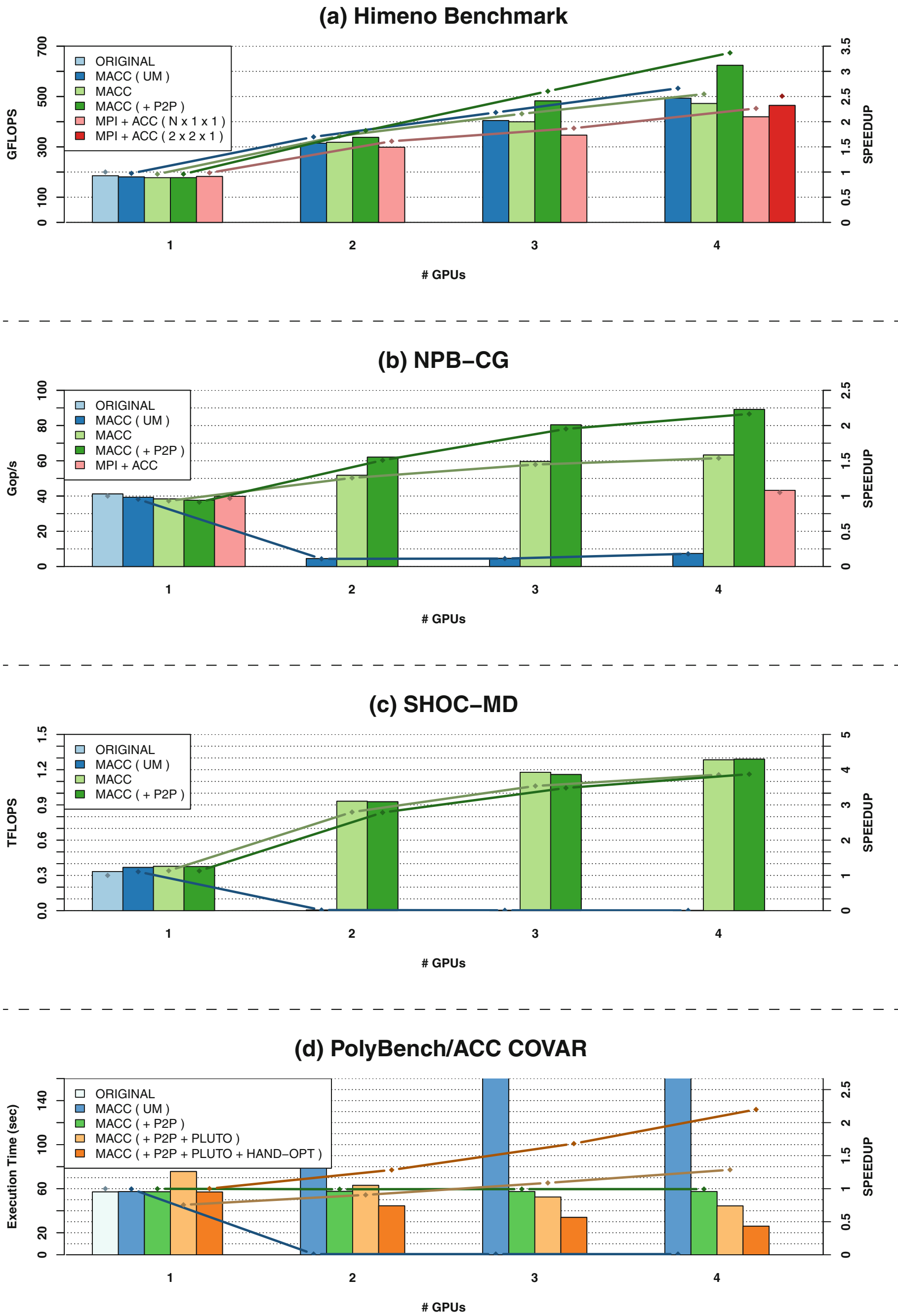

Fig. 4. This result with respect to number of GPUs displaying computational performance or execution time as well as speedup against original version

SHOC-MD. Performance results for the SHOC-MD benchmark is shown in Fig. 4(c). Unlike the other benchmarks, SHOC-MD do not benefit from P2P communication, since the application is inherently parallel. The cause for the observed superlinearity when moving from one to two GPUs (2.78× speedup on average) is not known (and it is kept even if we manually set the parallelism sizes prior to transpiling). The UM version again thrashes memory, which significantly reduced application scalability.

PolyBench/ACC COVAR. The result for the COVER benchmark is shown in Fig. 4(d). By our proposed method, the benchmark does not scale at all due to the most computation is done by a single GPU. On the other hand, by combining MACC with PLUTO, the symmetric-matrix creation (SC) and the covariance calculation (CC) are separated. This allows the application to exploit multiple GPUs on the CC. However, the SC becomes executed sequentially due to both MACC's and PGI Compiler's inability to resolve the loop-carried dependency, which drastically reduces the performance. We overcome this problem by manually adding a single directive of `loop` construct to make sure the SC loop parallelizable (HAND-OPT in Fig. 4(d)). This manual optimization can be automated by a direct operation of the polyhedral model (we consider this future work). The HAND-OPT code (SC is still performed on single-GPU though) reaches a performance of 2.20× speedup on average. Using UM drastically reduces performance— again due to memory thrashing.

7 Conclusion

In this paper, we proposed MACC — an OpenACC transpiler to automatically use multiple GPUs. We described and revealed how our transpiler performs the transformations, how data are kept coherent and how multiple GPUs are used. We showed that our proposed framework can transparently use or easily be integrated into existing infrastructure by leveraging architecture-specific P2P support (NVLink), external polyhedral compiler passes (PLUTO), and Unified Memory— an alternative to our custom data coherence protocol.

We evaluated our implementation with respect to four, well-know and important benchmarks. We quantified the performance of our transpiler. We found that our custom data coherence protocol outperforms that of Unified Memory and that using P2P communication can drastically improve scalability. We also illustrated a case where our transpiler can use external compilation strategies to overcome bottlenecks otherwise impossible to overcome. Finally, we also showed that for some applications we can compete with handwritten MPI code.

In the future, we plan to continue developing the transpiler to include more optimizations and evaluate more benchmarks. Moreover, we do plan to support a more variety of accelerators, such as FPGAs or many-core accelerators (Xeon Phis).

Acknowledgements. This work was supported by JST-CREST under Grant Number JPMJCR1303 and JPMJCR1687, and JSPS KAKENHI under Grant Number JP16F16764.

References

1. Global Scientific Information and Computing Center, Tokyo Institute of Technology. TSUBAME. http://www.gsic.titech.ac.jp/en/tsubame
2. NVIDIA: DGX SATURNV Supercomputer for AI and Deep Learning. https://www.cscs.ch/computers/piz-daint/
3. Oak Ridge Leadership Computing Facility. Summit. https://www.olcf.ornl.gov/summit/
4. NVIDIA: About CUDA. https://developer.nvidia.com/about-cuda
5. The Khronos Group Inc.: OpenCL Overview. https://jp.khronos.org/opencl/
6. OpenACC-standard.org. OpenACC. https://www.openacc.org/
7. The OpenMP ARB: The OpenMP API specification for parallel programming. http://www.openmp.org
8. Unified Memory in CUDA 6: NVIDIA. https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/
9. NVIDIA NVLink High-Speed Interconnect. NVIDIA. http://www.nvidia.com/object/nvlink.html
10. Komoda, T., Miwa, S., Nakamura, H., Maruyama, N.: Integrating multi-GPU execution in an OpenACC compiler. In: The 42nd International Conference on Parallel Processing (ICPP) (2013)
11. Ramashekar, T., Bondhugula, U.: Automatic data allocation and buffer management for multi-GPU machines. ACM Trans. Architect. Code Optim. (TACO) **10**(4) (2013)
12. Sakdhnagool, P., Sabne, A., Eigenmann, R.: HYDRA: extending shared address programming for accelerator clusters. In: Shen, X., Mueller, F., Tuck, J. (eds.) LCPC 2015. LNCS, vol. 9519, pp. 140–155. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-29778-1_9
13. Scogland, T.R.W., Feng, W.-C., Rountree, B., de Supinski, B.R.: CoreTSAR: core task-size adapting runtime. IEEE Trans. Parallel Distrib. Syst. (TPDS) **26**(11), 2970–2983 (2015)
14. Xu, R., Tian, X., Chandrasekaran, S., Chapman, B.: Multi-GPU support on single node using directive-based programming model. In: Scientific Programming (2015)
15. Chakravarty, M.M.T., Keller, G., Lee, S., McDonel, T.L., Grover, V.: Accelerating haskell array codes with multicore GPUs. In: The Sixth Workshop on Declarative Aspects of Multicore Programming (DAMP) (2011)
16. Svensson, B.J., Vollmer, M., Holk, E., McDonell, T.L., Newton, R.R.: Converting data-parallelism to task-parallelism by rewrites. In: 4th ACM SIGPLAN Workshop on Functional High-Performance Computing (FHPC) (2015)
17. Nakao, M., Murai, H., Shimosaka, T., Tabuchi, A., Hanawa, T., Kodama, Y., Boku, T., Sato, M.: XcalableACC: extension of XcalableMP PGAS language using OpenACC for accelerator clusters. In: 2014 First Workshop on Accelerator Programming using Directives (WACCPD) (2014)
18. Kim, J., Lee, S., Vetter, J.S.: An OpenACC-based unified programming model for multi-accelerator systems. In: The 20th ACM symposium on Principles and Practice of Parallel Programming (PPoPP) (2015)

19. Kwon, O., Jubair, F., Min, S.-J., Bae, H., Eigenmann, R., Midkiff, S.P.: Automatic scaling of OpenMP beyond shared memory. In: Rajopadhye, S., Mills Strout, M. (eds.) LCPC 2011. LNCS, vol. 7146, pp. 1–15. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36036-7_1
20. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools, 2nd edn. Addison-Wesley, Reading (2006)
21. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. In: ACM SIGPLAN Programming Languages Design and Implementation (PLDI) (2008). http://pluto-compiler.sourceforge.net
22. Omni Compiler Project: Omni Compiler. http://omni-compiler.org
23. Omni Compiler Project: XcodeML. http://omni-compiler.org/xcodeml.html
24. NVIDIA: Tesla P100 Most Advanced Data Center Accelerator. http://www.nvidia.com/object/tesla-p100.html
25. ACCC: RIKEN. Himeno benchmark. http://accc.riken.jp/en/supercom/himenobmt/
26. NASA Advanced Supercomputing Division. NAS Parallel Benchmarks. https://www.nas.nasa.gov/publications/npb.html
27. Xu, R., Tian, X., Chandrasekaran, S., Yan, Y., Chapman, B.: NAS parallel benchmarks for GPGPUs using a directive-based programming model. In: Brodman, J., Tu, P. (eds.) LCPC 2014. LNCS, vol. 8967, pp. 67–81. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-17473-0_5. https://github.com/uhhpctools/openacc-npb
28. Danalis, A., Marin, G., McCurdy, C., Meredith, J.S., Roth, P.C., Spafford, K., Tipparaju, V., Vetter, J.S.: The scalable heterogeneous computing (SHOC) benchmark suite. In: Third Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU-3) (2010). https://github.com/vetter/shoc/tree/openacc
29. Grauer-Gray, S., Xu, L., Searles, R., Ayalasomayajula, S., Cavazos, J.: Auto-tuning a high-level language targeted to GPU codes. In: Proceedings of Innovative Parallel Computing (InPar) (2012). https://cavazos-lab.github.io/PolyBench-ACC/

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.