JACC: An OpenACC Runtime Framework with Kernel-Level and Multi-GPU Parallelization

Kazuaki Matsumura, Simon Garcia De Gonzalo, Antonio J. Peña
Barcelona Supercomputing Center (BSC)
{kazuaki.matsumura, simon.garcia, antonio.pena}@bsc.es

***Abstract*—The rapid development in computing technology has paved the way for directive-based programming models towards a principal role in maintaining software portability of performance-critical applications. Efforts on such models involve a least engineering cost for enabling computational acceleration on multiple architectures while programmers are only required to add meta information upon sequential code. Optimizations for obtaining the best possible efficiency, however, are often challenging. The insertions of directives by the programmer can lead to side-effects that limit the available compiler optimization possible, which could result in performance degradation. This is exacerbated when targeting multi-GPU systems, as pragmas do not automatically adapt to such systems, and require expensive and time consuming code adjustment by programmers.**

This paper introduces JACC, an OpenACC runtime framework which enables the dynamic extension of OpenACC programs by serving as a transparent layer between the program and the compiler. We add a versatile code-translation method for multi-device utilization by which manually-optimized applications can be distributed automatically while keeping original code structure and parallelism. We show in some cases nearly linear scaling on the part of kernel execution with the NVIDIA V100 GPUs. While adaptively using multi-GPUs, the resulting performance improvements amortize the latency of GPU-to-GPU communications.

***Index Terms*—Multi-GPUs, Runtime System, Code Generation, Directive-Based Programming**

I. Introduction

Designing and building supercomputers is a complex task in the field of high-performance computing (HPC). The hardware, middleware and algorithms need to effectively collaborate to achieve ideal results for massive and practical problems. To facilitate the easy usage of supercomputers, compiler technologies have been developed with highly automated program optimizations that use domain-specific knowledge and understandings of target architectures [1].

Directive-based programming has been employed for enabling accelerator use, while replacing vendor-specific coding with directive insertion. Keeping software portability with minimum engineering efforts upon sequential code, OpenACC and OpenMP are now widely used for accelerator programming [2], [3]. However, pursuing ideal performance is often challenging. The insertion of directives by the programmers results in compilation side-effects that lead to less program-characteristics exposure for compilation [4]; thus, programmers aiming at better efficiency are forced to reshape their code merely for adjusting to the environment such as compilers, software stacks and heterogeneous architecture. Moreover, to follow the program modification, additional runtime parameters are often introduced for each program segment. Therefore, managing rewritten code is far from clear regardless of the complexity of transformation. Specifically, the parallelism among kernels is rarely addressed and the multi-device utilization is basically dismissed due to the little usability of data dependency information.

To explore new optimization opportunities, this paper extends OpenACC to hide code redundancy of optimization behind the runtime system in order to facilitate compiler development. While requiring no modification on original programs, our framework JACC* provides an environment for dynamic analysis, rescheduling and distribution of execution along with on-the-fly kernel specialization by wrapping up existing OpenACC compilers. Although other directive-based programming work develops dedicated runtime systems for specific optimization [5], [6], our framework integrates the compilation and runtime phases so as to utilize both aspects for additional efforts especially aiming at exploiting parallelism.

Additionally, we address multi-GPU work distribution, while considering the high memory latency of GPUs. To accomplish this, we add a novel code-translation technique named *predicated-based filtering* to automate multi-device use. We never split loop ranges nor introduce fine dependency analysis, but divide data ranges to be updated on each device. This idea allows to distribute highly-tuned code without changing code structure nor parallelism. Our contributions are as follows:

- We create JACC, an OpenACC framework which facilitates various dynamic features including runtime data analysis and compilation. JACC enables kernel-level parallelization through an asynchronous mechanism.
- We propose and describe a new multi-GPU kernel distribution method by leveraging JACC for complex applications. Our proposed technique is successful at multi-GPU execution on application that previous work fails to offload to multiple devices.
- We propose and describe an adaptive algorithm that automatically determines the adequate kernels for

*Available at http://github.com/epeec/JACC.

DOI 10.1109/HiPC53243.2021.00032

multi-GPU execution. Our adaptive algorithm considers the bandwidth of the peer-to-peer communication between GPUs when selecting kernels for multi-GPU execution.

- We evaluate all our contributions by using two different OpenACC compilers on a NVIDIA V100 multi-GPU system. We show that using our proposed methods and techniques available through JACC, we are able to achieve nearly linear scaling when excluding the latency of communication. Additionally, we are able to successfully improve the performance of a variety of manually-tuned NAS Parallel Benchmarks.

The rest of the paper is structured as follows. Section II discusses the recent trend of GPUs alongside a brief overview of OpenACC. Section III introduces our runtime system with basic extensions. In Section IV, predicate-based filtering is described. Section V shows our experimental methodology. Section VI evaluates our proposed technique on the state-of-the-art hardware. Section VII discusses related work. Finally, Section VIII concludes this paper.

II. Background

A. GPUs

Graphics Processing Units (GPUs) originated as graphic processors supporting massive parallelism. In contrast to CPUs, which are used for general computations and system management, GPUs are usually dedicated to those parts of applications featuring a high degree of parallelism. Since GPUs can hide the latency of memory requests by overlapping it with execution, the peak bandwidth of GPUs is significantly larger than that of CPUs. Furthermore, the hierarchical memory system including multi-level caches, shared memory and registers, allows exploiting spatial and temporal locality. We show the recent performance changes of NVIDIA GPUs in Fig. 1. Until P100, which began employing the second generation of High Bandwidth Memory (HBM2), the performance gap between memory bandwidth and computational throughput had been growing larger generation after generation. Therefore, memory accesses tend to be the performance bottleneck of applications. Currently, the NVLink interconnect [7] is enhancing peer-to-peer communication among multiple GPUs. The NVLink bandwidth allows not only compute-intensive but memory- intensive applications to utilize several devices. Notably, Unified Memory (UM) realizes data accesses that automatically solve data coherence

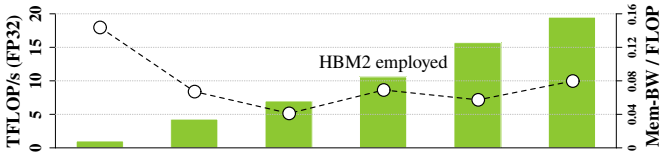

Fig. 1: Performance changes of recent GPUs. The bars show peak performance in TFLOP/s. The ratio to the memory bandwidth is shown by the line plot.

in the entire system while performing peer-to-peer communications. However, frequent page faults cause heavy performance degradation [8]; hence, users are basically required to avoid data sharing or perform explicit memory copies among GPUs.

B. OpenACC

Directive-based programming models have become common in HPC. Application developers can take them for granted in most supercomputing environments equipped with modern accelerators such as GPUs or FPGAs. Programmers benefit from their ability to target the same code easily to different possible accelerator architectures. OpenACC [2] offers compiler directives to program accelerators in existing languages. Without introducing vendor-specific languages such as CUDA, users are allowed to parallelize their code and rely on the compiler for generating device-specific application code. Fig. 2 shows an OpenACC code written in C that updates array x with the multiplication of array y.

```
1  #pragma acc data copyout(x[0:N]) present(y)
2  #pragma acc parallel loop
3  for(int i=0; i<N; i++) x[i] = y[i] * y[i];
```

Fig. 2: Accelerator programming in OpenACC

The components of OpenACC are made of kernels and routines. An OpenACC kernel is the unit of program execution on accelerators to be launched with specified parallelism (consisting of `gang`/`worker`/`vector`). Since the host code is executed on CPUs, kernel execution can be asynchronous to CPU execution and multiple kernels can be simultaneously run on the same device. The environment for kernel execution, such as device setting, data copies to/from devices and synchronous behavior, can be controlled by OpenACC routines.

OpenACC directives are provided for specifying code segments as kernels or defining data on devices along with several options (Lines 1-2 of Fig. 2). Although data-related directives can be replaced by routine calls, OpenACC kernels have to be embedded on original source files with directives to be converted to device-specific code at compile time. Therefore, additional code segments have to be put in place along with additional variables in order to be calculated from

```
1  for (int d = 0; d < NUM_DEVICES; d++) {
2    acc_set_device_num(d, 0);
3    int length = N/NUM_DEVICES;
4    int init   = length * d;
5    int until  = length * (d + 1);
6    #pragma acc data copyout(x[init:length])\
7                    present(y) async(d)
8    #pragma acc parallel loop async(d)
9    for(int i=init; i<until; i++)
10     x[i] = y[i] * y[i];
11 }
```

Fig. 3: Multi-device use in OpenACC

various dynamic parameters at runtime. Fig. 3 shows the example of multi-device utilization in OpenACC with asynchronous execution to each other devices. Additional code segments surround the kernel, calling an OpenACC routine to switch devices (Line 2 of Fig. 3) and setting variables to divide the loop execution (Lines 3-5 of the same figure). In real applications, many other factors such as runtime information and device-to-device communication are concerned; hence, in-situ kernel declarations bring additional complexities to directive-based software development. Also, OpenACC kernels are statically declared; thus, for a complex dynamic application the programmer is required to prepare adjusted kernels before compilation, regardless of whether runtime information is available.

III. JACC: Runtime-Extended OpenACC

Our work introduces dynamic analysis and compilation to OpenACC directive-based programming, allowing further efforts on optimization at runtime. All the components of OpenACC, here, are provided as runtime routines leveraging existing compilers. By transforming directives into a sequence of routine calls, OpenACC compilers can enable on-the-fly features such as kernel specialization and load-balancing.

A. JACC

We build JACC, a just-in-time compilation system for OpenACC, in which input directives are replaced with runtime routines. JACC hides every OpenACC feature behind a runtime library to cushion dependency to specific compilers. Once a kernel is compiled for the first time, its device code is cached to be reused for subsequent launches. Even though JACC is developed upon existing compilers, it allows calling of CUDA routines and kernels through its library. Fig. 4 shows the converted code of Fig. 2 to call runtime routines. First, combined directives (e.g. `parallel loop` of Line 2 of Fig. 2) are decomposed into three basic directives of `parallel`, `loop` and `data`. Then, for each directive, JACC inserts corresponding routines that are implemented in its library, shown in Fig. 4 (Lines 2, 5 and 12).

During program execution, JACC data-related routines that wrap OpenACC routines (Lines 2 and 12 of Fig. 4) assume the roles of the original directives. The routine

```
1  /* Entry of #pragma acc data */
2  jacc_create(x, N * sizeof(float));
3
4  /* #pragma acc parallel loop */
5  jacc_kernel_push(
6    "#pragma acc parallel present(x, y)\n"
7    "#pragma acc loop\n"
8    "for(int i=0; i<N ; i ++) /* ... */",
9     /* args */, /* flags */);
10
11 /* Exit of #pragma acc data */
12 jacc_copyout(x, N * sizeof(float));
```

Fig. 4: Converted code by JACC (arguments omitted)

`jacc_kernel_push` launches kernel execution while accepting source code in a string with arguments that hold runtime information (Lines 5-9 of the same figure). It should be noted that the `loop` directive is used for marking parallelism; therefore, the directive is kept in kernel strings. When the routine finds no compiled kernel for given source code or needs to update existing kernels, function code is generated to emit device code by a specified compiler and to have additional arguments for code extension. After linked dynamically, this function is called through a foreign function interface (FFI) for passing arbitrary arguments. JACC's library for each routine is extended to collect runtime information and support dynamic features.

B. Basic Extension

OpenACC is a high-level programming model designed for accelerator abstraction, which accepts program optimizations through both direct APIs and program modification in the base language (e.g. C, C++, Fortran). Whereas the OpenACC APIs are added explicitly to a program as directive clauses with a specific intent, the intent and effect of base-language modifications to a program are implicit. There exist a large body of work that studies the implications of base-language modification on OpenACC compilation. JACC works as a runtime solution for dynamic optimization for both the OpenACC APIs and base-language program modification. Because JACC's ability to handle both types of optimizations, it can automatically overlap kernel execution, and thus achieve inter-kernel parallelism. Also, additional on-the-fly kernel specialization using runtime information extracted from profiling results for better resource use and intra-kernel parallelism are possible.

1) Automated Asynchronous Execution: Since JACC automatically provides a function interface for each OpenACC kernel, additional runtime information needed for extended execution shown in lines 2-5 of Fig. 3, can be passed through as arguments to the JACC function interface without the need to generate redundant code snippets. Instead of compiler-generated code, which is hard to manage as global program information, JACC's runtime calculates the required arguments on its own runtime environment and then proceeds to call the kernel functions using them. The benefit of this approach is that information across multiple execution instances can be easily shared for further optimizations. Moreover, JACC provides the ability to update kernels dynamically with additional runtime information after program compilation is invoked, thus, providing a straightforward mechanism for runtime extensions of original kernel declaration.

For asynchronous execution, we automatically overlap kernel launches and data operations with each other as well as host execution. Each JACC routine has the ability to track array references, and if data dependencies are encountered across two or more routines, JACC will schedule them in the same asynchronous queue. When multiple queues are concerned, synchronous operations are performed only

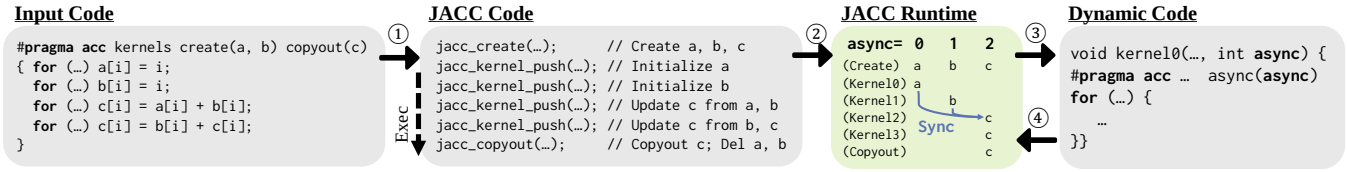

Fig. 5: Execution flow of automated asynchronous execution. First, the JACC code is generated from the input (①). Following the execution, the runtime manages data declarations and checks data dependency among asynchronous queues based on data ranges of actual kernel arguments (②). Dynamic code is created from kernel code to be executed on a destined queue based on the kernel argument **async** (③) and synchronization is performed with the host if necessary (④).

among those queues that require them, while skipping redundant synchronization on already solved dependencies. JACC achieves this by maintaining timestamps of data accesses and the most recent synchronization among queues. If there is no data dependency to prior execution, the least recently used queue is selected. We guarantee original code semantics by obligatory synchronization that is performed immediately after kernel execution that deals with array writes to undefined data regions and explicit variable updates such as reduction. Data ranges linked to given pointers are tracked through JACC's runtime routines, and managed in a red-black tree as OpenACC compilers do to accept any address of declared data [9].

Fig. 5 shows JACC's automatic asynchronous code optimization and execution flow. During execution by the JACC's runtime (Step 2 of Fig. 5), the synchronization between queues is performed. For example, before `Kernel2` execution is allowed, a synchronization call is performed to wait for the updated arrays `a` and `b`. However, for `Kernel3`, the dependency on `b` is already solved by the previous synchronization, thus the execution does not wait for other queues. For the overlapping execution, during the dynamic code generation (Step 3 of Fig. 5), the OpenACC clause `async` is set for each kernel declaration, and asynchronous execution queues are selected.

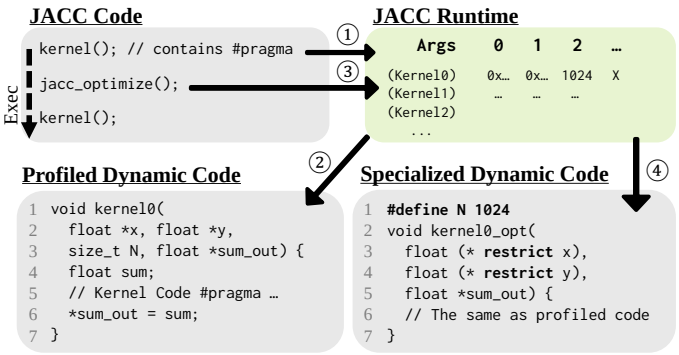

Fig. 6: Compilation flow with on-the-fly kernel specialization. To collect runtime information about kernel arguments, mainly addresses of pointers and values of variables, profiling execution is conducted (①) with original kernel declarations (②). With a user-invoked compilation event (③), specialized code is generated based on the profile results and thereafter used for succeeding kernel execution (④). The `sum` variable in the profiled code is exported through the `sum_out` pointer, thus it is kept for specialized code as a dynamic variable (notated as "X" in the argument log of the JACC runtime).

2) Kernel Specialization: We attempt to refine resource use by attaching runtime information to function code so as to enable aggressive optimization in kernel compilation. The compilation flow is shown in Fig. 6. For specialization, profile execution is conducted before the optimization; then, at an additional compilation event (`jacc_optimize` of JACC code in Fig. 6), parameters being invariable during the execution are substituted with constants (Line 1 of specialized code in Fig. 6) to lower on-chip resource use. Besides that, pointer references that never conflict with others are declared with the `restrict` keyword to ensure intra-kernel parallelism (Lines 3-4 of specialized code in Fig. 6). Even though user-invoked events can be substituted automatically by existing just-in-time compilation techniques [10], we explicitly invoke the desired compilation optimizations to measure the potential performance benefits.

IV. Multi-GPU Utilization with Predicates

We further improve utilization of intra-kernel parallelism by enabling multi-GPU execution with JACC. Whereas previous studies have persistently focused on loop splitting over plural GPUs [5], [11], this work divides data regions that each GPU updates to support real applications that usually entangle memory accesses among loop iterations.

A. Predicate-Based Filtering

Our technique, named *predicate-based filtering*, limits memory accesses depending on data regions that the GPU writes to, assuming that redundant computational code and parameters do not degrade performance due to low computational latency and high memory latency on GPUs. First, we introduce data ranges for each updated array so that array writes can be filtered based on the assigned range. For instance, in C code, array write `a[i]*=2` is rewritten to `(a_lb <= i && a_ub >= i) ? a[i]*=2 : a[i]`, where `a_ub` and `a_lb` indicate the upper and lower bound of array `a`, that are specified depending on the GPU. In Fortran, since there is no nested assignment, we use IF statement for filtering, with subsequent ELSE statement which contains an assignment of the same expression (`a(i)=a(i)`) that is later optimized away but facilitates compiler analysis. Additionally, we develop data-flow analysis for the innermost parallel region in each kernel to detect data dependencies between arrays. Then, we filter them to restrict accesses while solving dependencies as shown in Fig. 7. This analysis converts both array and variable references into

```
1  a[i]=x; b[i]=a[i]; x=c[j]; a[k]=x; b[k]=a[k];
```

```
1  /* a[i]=x */
2  ((a_lb<=i && a_ub>=i)||
3   (b_lb<=i && b_ub>=i)) ? a[i]=x:a[i];
4  /* b[i]=a[i] */
5  ((b_lb<=i && b_ub>=i)) ? b[i]=a[i]:b[i];
6  /* x=c[j] */
7  x=((a_lb<=k && a_ub>=k)||
8     (b_lb<=k && b_ub>=k)) ? c[j]:0;
9  /* a[k]=x */
10 ((a_lb<=k && a_ub>=k)||
11  (b_lb<=k && b_ub>=k)) ? a[k]=x:a[k];
12 /* b[k]=a[k] */
13 ((b_lb<=k && b_ub>=k)) ? b[k]=a[k]:b[k];
```

Fig. 7: Example of predicate-based filtering in C code. Original (top) and filtered code (bottom). References to array `a` have predicates for updating array `b` and itself (Lines 2-3 and 10-11), the references to array `b` have for itself (Lines 5 and 13), and the reference to array `c` has for array `a` and `b` (Lines 7-8).

the static single assignment (SSA) form, and iteratively finds dependencies among array accesses.

Updated data are sent to all other GPUs after each kernel execution to establish data coherency. Device-memory allocations and host-to-GPU communications are replicated on all the GPUs and the primary GPU is used for GPU-to-host transfers. To guarantee the result of our analysis, we check kernel arguments so as to duplicate computation and disable communications on data that are referred through more than two pointers which at least one of them is read and one is written (i.e. aliased pointers, which are usually avoided in OpenACC programs for loop independence). When several pointers share the same array to update, we merge their access ranges to follow the widest. The necessary computation for array-write indexing is always duplicated. Regarding reduction or variable writes that are explicitly exported to host, we filter the computation based on the range of the outermost parallel iterator.

B. Division of Multidimensional Arrays

While being applicable to all OpenACC kernels as far as array writes are concerned, our filtering technique needs to duplicate execution on each GPU when references between split ranges (such as all-to-all dependencies in Fig. 8) are found inside the kernel. We alleviate this restriction by leveraging dimensional information.

If multidimensional arrays are linearly split regardless of the dimensional characteristic, the data dependency could be dispersed to the entire sections of array accesses. For example, the write to `lhsY` in Fig. 8 (Lines 7-14) would be filtered for succeeding reads; thus, all the computations would be duplicated on each GPU. Here, we utilize parallel iterators (such as `i` and `k` in Fig. 8) to locate *parallel dimensions*, where arrays can be split without duplicated computation. Based on the number of iterators each dimension contains, we select the parallel dimension for each updated array to have the most parallel iterators while containing the least sequential iterators (such as `m` and `n`). When there are several candidates, we choose the leftmost dimension in the C language and the rightmost dimension in Fortran to gain better performance with suitable accesses to the memory layout (row-major and column-major order, respectively).

Each kernel execution is performed while equally dividing parallel dimensions among GPUs and accompanied by the GPU-to-GPU communication through CUDA routine `cudaMemcpy2DAsync`. Each array is concurrently transferred regarding other data and other GPUs. We synchronize GPUs at the beginning and ending of the communication.

```
1  #pragma acc parallel loop gang
2  for (i = 1; i <= gp02; i++) {
3   #pragma acc loop worker vector
4   for (k = 1; k <= gp22; k++) {
5    for(m = 0; m < 5; m++)
6     for(n = 0; n < 5; n++) {
7      lhsY[n][m][BB][jsize][i][k] =
8        lhsY[n][m][BB][jsize][i][k]
9        - lhsY[n][0][AA][jsize][i][k]
10       * lhsY[0][m][CC][jsize-1][i][k]
11    /* - lhsY[n][1..4][AA][jsize][i][k]
12       * lhsY[1..4][m][CC][jsize-1][i][k] */;
13     }}}
```

Fig. 8: Kernel code from NPB-BT. Two inner loops are unrolled in actual code. Linear splits cause all-to-all dependencies among statements.

C. Adaptive Utilization

In order to avoid lower performance due to data distribution overheads, we enable multi-GPU execution for each kernel in an adaptive way, while otherwise duplicating computation on all GPUs and performing no GPU-to-GPU communication.

First, we start the execution on the mode of duplication. After an initial warm-up run, we profile the average ratio of array-write size (WriteSize) to execution time ($time_{\text{Kernel}}$) as eff_{dup}, until we observe five executions that satisfy the requirement to have the peak performance be better than duplication:

$$time_{\text{Kernel}} > time_{\text{Kernel}}/n_{\text{GPUs}} + \text{WriteSize}/peak_{\text{P2P}} \quad (1)$$

Here, $peak_{\text{P2P}}$ is the unidirectional bandwidth of one GPU-to-GPU connection (e.g. 25GB/s in NVIDIA DGX-1) and n_{GPUs} is the number of GPUs used.

After switching to multi-GPU execution, we disable it when either one of the two following conditions is satisfied at least five times and the average difference of the left value and the smaller right value goes above zero in equations (2-3).

1) The total execution time including the communication time ($time_{\text{Comm}}$) becomes longer than the kernel execution time multiplied by n_{GPUs}:

$$time_{\text{Kernel}} + time_{\text{Comm}} > time_{\text{Kernel}} \times n_{\text{GPUs}} \quad (2)$$

2) The total execution time surpasses the profiled execution time of duplication:

$$time_{\text{Kernel}} + time_{\text{Comm}} \quad > \quad eff_{\text{dup}} \times \text{WriteSize} \qquad (3)$$

The first condition excludes the case that the GPU-to-GPU communication has larger latencies than expected. The second prevents performance degradation caused by kernels that are unsuccessfully parallelized.

D. Implementation

We integrate predicated-based filtering into JACC, which translator is implemented as a XcodeML [12] converter. The execution flow of predicate-based filtering is shown in Fig. 9. From OpenACC code in C or Fortran, our implementation generates JACC code, in which kernel code is embedded as strings. Although the kernel code can be translated at runtime, we apply predicated-based filtering beforehand for our experiments; thus, the embedded kernel code already has the predicates. JACC's runtime code generator is utilized for setting array ranges and constructing multi-GPU reduction code based on the arguments of runtime routines. Runtime overheads of JIT dynamic compilation are well known, and itself a target of research [10], however it is outside the scope of this work and left as a possible future area for optimization. Thus, we evaluate the performance without dynamic compilation overheads (Section III-B2), which is on average about 2 seconds per kernel for the initial compilation.

For simultaneous execution of multiple GPUs, we use OpenMP rather than OpenACC's asynchronous mechanism that holds some non-negligible latencies [13], [14]. OpenMP's pragmas are put only inside JACC's library. Whereas GCC does not allow the mix of OpenACC and OpenMP, our separated-compilation strategy realizes a combinatory use for both PGI and GCC. The OpenMP use is not inherent here and our techniques introduced in this paper are general enough to support other compilers and other programming models.

V. Methodology

A. Hardware and Software

We measure the performance changes of our proposed techniques using the NVIDIA Tesla V100 SXM2 GPUs (16GB Memory) on NVIDIA DGX-1. DGX-1 contains eight GPUs, where each four GPUs are interconnected with NVLink in an all-to-all fashion. Additionally, each GPU has one NVLink connection to one of the other four GPUs, while the remaining GPUs and CPUs are connected with PCI Express Gen3 x16. We use only the tightly coupled four GPUs to perform our experiments where each link offers a unidirectional bandwidth of 25GB/s while $GPU_0 \leftrightarrow GPU_3$ and $GPU_1 \leftrightarrow GPU_2$ are dually linked. Tesla V100 has a peak single-precision performance of 15.7 TFLOP/s and a peak memory throughput of 900GB/s. DGX-1 uses dual 20-core Intel Xeon processors.

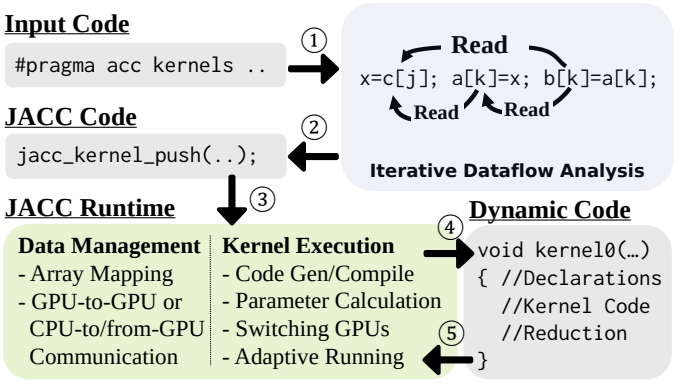

Fig. 9: Execution flow of predicate-based filtering. Having the results of iterative dataflow analysis (①), JACC code is generated from the input (②). On the execution (③), dynamic code is created from kernel code, extended declarations and reduction code in accordance with runtime information (④). Once the dynamic code is compiled with a specified compiler, the kernel execution is conducted with dynamic parameters while switching GPUs and reflecting results to the host and each other devices (⑤).

For the compilation, we use the PGI compiler 20.9 and GCC 10.2.0 with CUDA 11.0. Currently, the Fortran translation is tested only for predicate-based filtering with PGI. The experiments with GCC are conducted while omitting the `worker` parallelism (described in Section II) so as not to exceed the maximum number of threads allowed in GCC per thread-block.

B. Benchmarks

For the evaluation, we use a manually-tuned OpenACC version of NAS Parallel Benchmarks (NPB) written in C [15] and three Fortran mini-apps: CloverLeaf [16], CCS-QCD [17] and the Himeno benchmark [18]. Each benchmark of NPB is executed with the largest problem size for the target GPUs (Class C) except EP, where we choose Class D for longer execution. Moreover, to prolong the execution time of CG and MG, we multiply the number of iterations by 50. With regard to CloverLeaf, we select the same input as the SPEC ACCEL benchmark suite [19], while maximizing the GPU memory utilization of the other two mini-apps.

EP is the only application to be compute-bound for arithmetic operations of random-number generation involving fewer array accesses, while other benchmarks become memory-bound on the state-of-the-art accelerators. There is no data dependency among parallel threads in EP.

BT/LU/SP deal with three-dimensional computational fluid dynamics (CFD) with different solvers. BT updates multidimensional arrays from 3D to 6D, especially 4D to 6D have the leftmost 1D to 3D dimensions for loop-independent indices as shown in Fig. 8 (Lines 7-14), respectively. LU/SP use 3D to 4D arrays in a similar fashion to BT. LU has relatively quicker execution for each kernel compared to BT/SP and the latency of BT mainly consists of the execution of several time-consuming kernels.

FT conducts 3D fast Fourier transform (FFT) incurring all-to-all accesses, and MG is the benchmark of multigrid computation which require long and short communications. CG calculates a minimum eigenvalue of a sparse symmetric positive matrix with the conjugate gradient method, causing irregular accesses to an updated 1D array.

CloverLeaf is a hydrodynamics mini-application consisting of 100+ kernels which sufficiently demonstrates the workflow of real-world applications. Based on Euler's method, 2D stencil grids are updated by each kernel having minimum control logics and halo accesses through double buffers for avoiding dependencies among loop iterations.

CCS-QCD simulates lattice quantum chromodynamics (QCD) with a linear-equation solver for a large sparse matrix in 3D. Himeno iteratively updates a 19-point stencil grid according to Jacobi's method, which code structure is far simpler than the other two mini-apps.

We report performance after an initial warm-up run that causes runtime compilation and profiling for adaptive utilization. The benchmark-reported data is quoted for the result of NPB and the total execution times for the Fortran mini-applications.

VI. Results

A. Basic Extension

Fig. 10 shows the performance changes with JACC's basic extension for both the PGI compiler and GCC using NPB. The `JIT w/ (PGI/GCC)` bars indicate the performance of converted code without any optimization. Here, only one asynchronous queue is used with `+Async`, whereas 16 queues are used with `+Overlap`. Along with that, the `+Var Opt` execution adds kernel optimization with constant parameters transformation discussed in Section III-B2. Furthermore, `+Restrict` adds `restrict` to pointers. Since GCC produces incorrect results with the original code of CG/LU/MG, they are omitted from results.

First, performance degradation is observed for converted code compared to original code in the case of BT/LU with PGI, where generated code fails to leverage static array sizes for some optimization at compile time because static arrays are separately declared. However, improvements are observed in the case of MG with PGI and EP with GCC. Otherwise, original performances are mostly kept. On

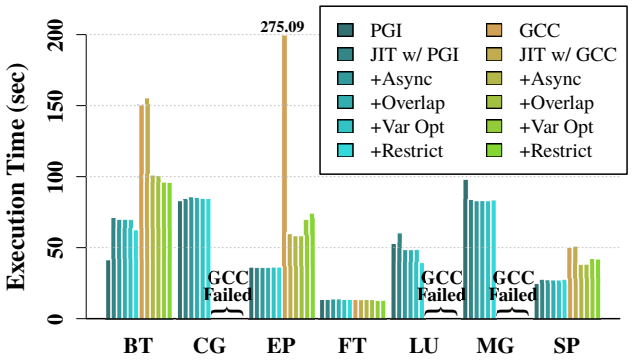

Fig. 10: Async & kernel opt on NVIDIA Tesla V100 SXM2

average, asynchronous execution with single queue achieves better performance by 3.43% with PGI and 22.08% with GCC, respectively. However, the time-consuming kernels in each benchmark prevent overlap execution; thus, `+Overlap` does not improve the performance from `+Async`. With `+Restrict`, we achieve better performance up to 23.39% in the case of BT/LU with PGI and 5.59% less performance in EP with GCC. The performance difference between PGI and GCC is primarily caused by the latency of memory allocation; PGI owns memory pools for device memory, while GCC does not.

The `+Var Opt` version has no performance change in most cases and rather worsen efficiencies in the case of EP/SP with GCC. Further exploration showed that some cases of `+Var Opt` suffer from limited arithmetic unit utilization caused by ineffective threads which are created due to reduced register use. On the other hand, performance improvements of `+Restrict` are achieved by parallelized memory accesses which require additional registers.

B. GCC Custom Allocation

Since the original version of GCC suffers the performance degradation by GPU-memory allocation, we integrate memory pools into GCC's runtime library libgomp for our multi-GPU experiments to show explicitly the performance improvements by kernel parallelization. We prepare two pools: one is for user-invoked memory allocation such as through pragmas and runtime routines. Another is for runtime-managed allocation of variables and stacks which tends to be much smaller than the former. We manage those pools to keep unused memory segments and reuse them for new allocation by selecting the smallest but capable one on the device.

With the memory-pool integration, GCC's efficiency becomes competitive to PGI while having -7.83% ~ 5.05% better throughputs for the plain JACC code except the case of EP, where the kernel execution poses a 38.31% overhead due to GCC's device-code efficiency, as shown in Fig. 11.

C. Multi-GPU Utilization

1) Total Improvements & Kernel Speedups: We show the overall performance and kernel speedups with predicate-based filtering in Fig. 11. When compared to single-GPU execution, the total execution time with our proposed technique is better in five among 10 evaluated benchmarks, from 4.05% up to 43.43% when enabling four GPUs. Especially, when only the kernel execution and GPU-to-GPU transfers are concerned, six benchmarks (BT/CG/LU/SP/CCS-QCD/Himeno) improve the execution time by 23.9% on average as shown in Table I. Other benchmarks still remain unchanged with some slight degradation up to 3.36% while having several kernels enabled for multi-GPU execution. The noticeable slowdowns we observe in the total execution time of LU/MG are caused due to other factors necessary for multi-GPUs such as memory allocation and synchronization. As an opposite fashion to `+Var Opt`, the predicate-appended code mostly

TABLE I: Performance details with PGI in use of four GPUs (The result of duplicated execution on all the GPUs is used for the Kernel Dup column only; Other columns use the results of adaptive execution. The Kernel Adapted column shows the kernel execution time for multi-GPU adapted kernels with the average. The **bold** values indicate performance improvements from the duplicated execution)

Name	Num Kernels	Kernel Dup [ms]	Num Adapted	Comm + Kernel [ms]	Kernel Total (ave) [ms]	Kernel Adapted (ave) [ms]	Comm (ave) [ms]	Average WriteSize	GPU-to-GPU Bandwidth
BT	46	88,715	3	**63,856**	46,639 (0.44)	14,940 (24.75)	17,217 (28.52)	684.10 MB	23.99 GB/s
CG	16	75,178	7	**44,137**	38,720 (0.08)	34,365 (0.12)	5,417 (0.02)	0.10 MB	5.40 GB/s
EP	4	37,485	3	37,787	37,710 (24.55)	37,710 (24.55)	77 (0.05)	0.03 MB	0.54 GB/s
FT	12	8,472	4	8,757	6,096 (47.29)	4,983 (62.37)	2,661 (33.31)	806.92 MB	24.23 GB/s
LU	59	76,325	7	**71,614**	64,342 (0.09)	3,886 (0.03)	7,272 (0.06)	0.39 MB	6.28 GB/s
MG	16	83,586	3	83,654	83,654 (0.47)	5,604 (0.35)	0 (0.00)	0.00 MB	0.00 GB/s
SP	65	27,809	3	**19,609**	16,648 (0.64)	1,969 (1.64)	2,961 (2.47)	58.24 MB	23.60 GB/s
CloverLeaf	114	55,017	3	55,272	54,079 (0.22)	10,980 (4.27)	1,193 (0.46)	5.46 MB	11.76 GB/s
CCS-QCD	27	26,517	11	**24,641**	13,031 (0.97)	5,811 (1.91)	11,610 (3.82)	91.67 MB	24.02 GB/s
Himeno	2	33,726	1	**23,149**	11,894 (5.93)	8,318 (8.30)	11,255 (11.23)	271.47 MB	24.18 GB/s

holds the performance of the plain JACC code with single-GPU use.

Profiling the kernel speedups with no adaptive execution showed that predicate-based filtering parallelizes many kernels. Using the memory-intensive benchmarks BT/SP, PGI achieves 2.83x and 3.59x improvements on four GPUs and GCC does 4.13x and 3.85x, respectively. For LU, however, the shorter than 1ms running time of each kernel execution limits acceleration to 1.40x, involving overheads for duplicating program structures on all the GPUs. EP does not have any improvement due to its compute-bound nature. Comparing adaptive and non-adaptive execution, CG has almost the same improvement despite other benchmarks are prevented from full parallelization due to the high communication-kernel ratios. For example, in Table I, around the 20% execution of CloverLeaf is distributed over multi-GPUs but those kernels are not well enhanced, while the remaining execution is duplicated because of the excessive communication latencies, hence, resulting in no speedup.

2) Data-Size Scaling: Fig. 12 shows the performance scaling with Himeno using different program sizes. Since we equally split array ranges for each GPU, the transfer size per GPU-to-GPU connection becomes smaller and the proportion of communication decreases when the number of GPUs is increased. From two to four-GPU use, we see different scaling of total GPU-to-GPU transfers: 1.70x speedup with size M, 1.95x with L and 1.99x with XL. In regard to kernel performance scaling from single to four GPUs, we achieve 1.53x, 2.19x and 3.64x improvements for size M, L and XL, respectively. For size M, multi-GPU execution suffers the overheads of both kernel and communication. Better scaling can be obtained with longer kernel execution and larger transfers as in the case of BT, which is successfully parallelized with communications of a six-dimensional array decomposed per GPU in 75 segments of 8MB size, having original kernel execution longer than 10ms.

Our technique further reduces the GPU-to-GPU communication latency as more GPUs are used. As future architectures move to having many accelerators with all-to-all interconnects, applications could benefit further from predicate-based filtering.

3) Comparison: Related work MACC [5] successfully parallelizes only two of those applications over multi-GPUs

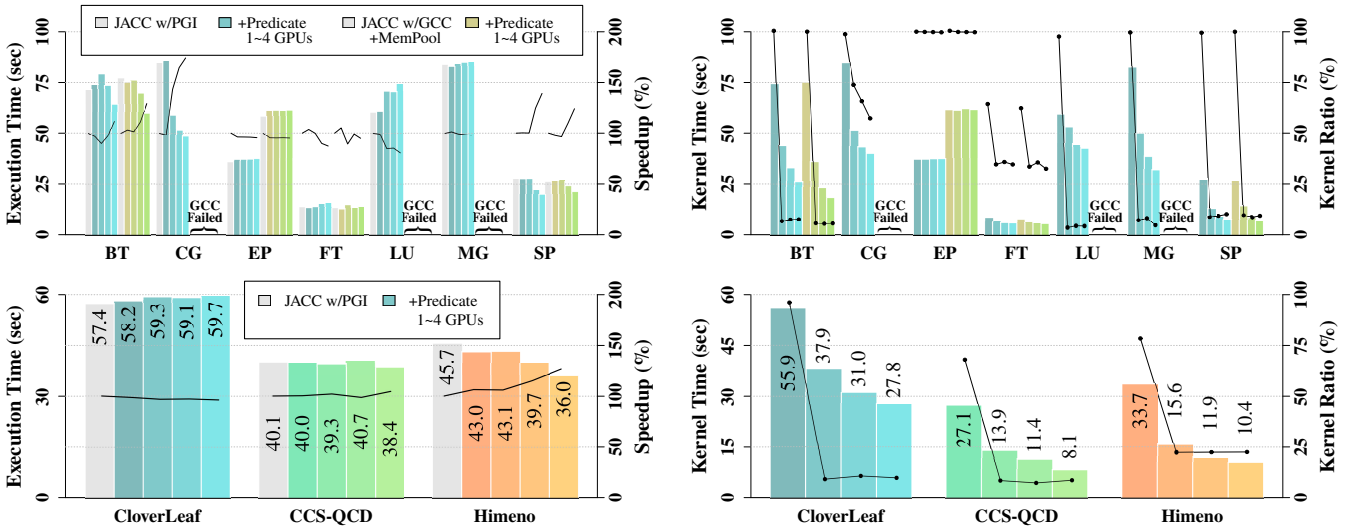

Fig. 11: Performance scaling of predicate-based filtering using NPB with PGI/GCC (top) and using the Fortran mini-apps with PGI (bottom). Left figures provide the execution times with bars and the speedups with lines compared to the plain JACC code. Right figures show the kernel time when we enable multi-GPU execution with no adaptive algorithm; the kernel ratio to the total time is given by the line.

based on code-level access-range analysis: CG and Himeno, which performance bottlenecks have non-overlapping linear writes for each loop iteration. Other multi-GPU work based on memory coherence mechanisms [11], [20] is also unable to support the remaining benchmarks without user effort.

We compare our technique to loop splitting. Fig. 12 includes the scaling of the manual code which uses the same algorithm as MACC. We notice that the loop-splitting code has better scaling for kernel execution: from single to four GPUs, it achieves 2.08x, 3.57x and 5.26x improvements for size M, L and XL, respectively. Moreover, the optimized communication for the stencil application significantly reduces the latencies. Those domain-specific approaches can be automated as long as compiler analysis allows; thus, we consider integrating them into our work for future refinement.

Fig. 13 shows the comparison between our technique and MACC using CG. MACC automates loop splitting of all the kernels in CG but employs no adaptive algorithm. From two to four-GPU utilization, our technique achieves better efficiency up to 3.50% by disabling multi-GPU execution for lower-latency kernels. Besides that, the adaptive execution has smaller kernel latencies than non-adaptive execution due to the same reason, with 44.09% better total efficiency.

VII. Related Work

Several studies conduct optimization upon the source code of directive-based programming models. In [21], Tian et al. perform scalar replacement on OpenACC code, that substitutes redundant array accesses with scalar references until the compiler reports that all available registers are utilized or all reused references are replaced. Barua et al. [22] optimally unroll OpenACC loops while estimating memory throughputs based on ILP. OptACC [6] finds a better OpenACC parallelism with either grid or direct search. JACC eases the implementation of those extensional work in a portable fashion to the user's environment while utilizing dynamic information.

There is some work addressing automated multi-GPU utilization with OpenACC. MACC [5] provides dynamic access-range analysis to distribute execution with GPU-to-GPU communication. Although MACC achieves better performance than a UM system, its analysis is only applicable to affine loops. Komada et al.'s compiler [11] keeps data coherence by tracking array writes in a similar way to UM but incurring additional array writes for it and performing data transfers after each kernel execution. Both previous work divides loop execution equally for each GPU and principally does not allow any intersection of array updates among devices, thus, cannot support many applications that our work parallelize.

Distributed-memory systems including multi-GPUs are also discussed regardless of programming models. Loop models such as polyhedral model have been widely employed to detect data dependency among compute nodes [23]. However, the input is typically restricted to affine or almost-affine loops and those work fixes workloads on each node before computing dependency, which involves intricate communication patterns or imposes loop transformations beforehand. Some libraries and frameworks are dedicated to multi-GPU execution through an abstraction which entails GPU-to-GPU communication [24]. Software-level memory managements that maintain data coherency are also capable for accommodating program distribution with little user intervention but manual efforts are required to adequately partition updated arrays while not overlapping them or otherwise introducing overheads [25]. Our predicate-based filtering provides a new way to

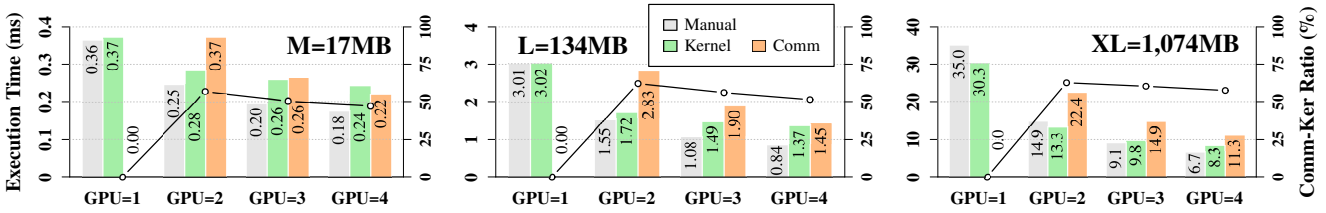

Fig. 12: Scaling with the number of GPUs for the stencil kernel in the Himeno benchmark. The grey bar shows the average execution time of a manually-tuned loop-splitting version which includes the communication time and the kernel time. The kernel and communication latencies by our technique are shown by the green and orange bars. The line plots the communication-to-kernel latency ratio. Since the kernel requires only 3D halo accesses, the optimized transfer finishes within 30us, 50us and 110us for the size M, L and XL, respectively, regardless of the number of GPUs used. The displayed data size equals the total grid proportions that all the GPUs update.

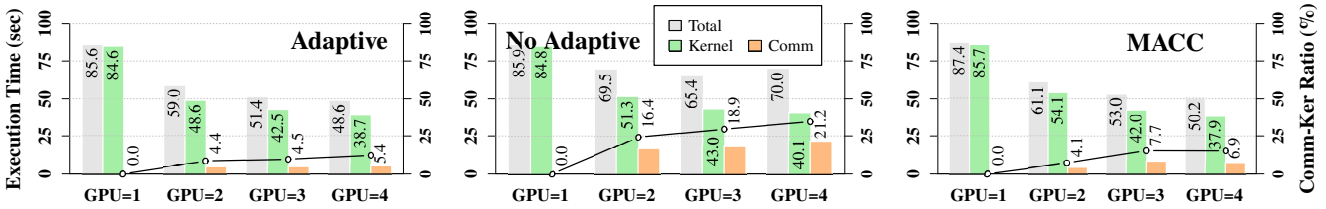

Fig. 13: Comparison between predicate-based filtering with/without the adaptive algorithm and MACC using NPB-CG. The total execution, kernel execution and GPU-to-GPU transfer times are shown by the grey, green and orange bars, respectively. The latency ratio of communication to kernel is depicted by the line.

parallelize many kernels based on source-code level transformation and dynamic information.

Dynamic compilation brings additional opportunities for performance improvement to the runtime system. NVIDIA's jitify [26] is a library that simplifies the use of CUDA Runtime Compilation. KernelGen [27] is a Fortran/C compiler that automates GPU code generation with polyhedral loop analysis of LLVM IR. Those works present dynamic features such as runtime alias analysis and parameter tuning alongside kernel specialization. On the other hand, JACC wraps OpenACC compilers and holds C/Fortran code for optimization.

VIII. Conclusion

Over the last decade, substantial work has been proposed for the optimization of directive-based programming models while facing difficulties with its implementation to arrange compiler groundwork. In this paper, we presented JACC, an OpenACC framework which facilitates runtime extension. Organizing data mappings and kernel arguments as runtime information, JACC creates dynamic code from original kernels and compiles it with a specified compiler in order to support on-the-fly code extension automatically. Due to the memory-bound nature of GPUs, we proposed predicate-based filtering, a novel code-translation technique of multi-GPU utilization, for distributing highly-tuned applications without additional user effort. JACC employs an adaptive algorithm for switching distribution based on the overhead of GPU-to-GPU communication. Having many kernels parallelized on a multi-GPU environment, we showed the performance improvements of several tested benchmarks where precise data-dependency analysis is always restrained.

Acknowledgement

The EPEEC project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 801051. We gratefully acknowledge the support of NVIDIA Solutions Lab who provided us the remote access to NVIDIA DGX-1. We would like to acknowledge the NVIDIA AI Technology Center (NVAITC) Europe for their valuable help.

References

[1] M. J. Wolfe, C. Shanklin, and L. Ortega, *High Performance Compilers for Parallel Computing*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[2] T. O. Organization, "OpenACC," 2011. [Online]. Available: https://www.openacc.org/

[3] T. O. ARB, "OpenMP," 1997. [Online]. Available: https://www.openmp.org/

[4] M. Khalilov and A. Timoveev, "Performance analysis of CUDA, OpenACC and OpenMP programming models on TESLA V100 GPU," in *Journal of Physics: Conference Series*, vol. 1740, no. 1. IOP Publishing, 2021, p. 012056.

[5] K. Matsumura, M. Sato, T. Boku, A. Podobas, and S. Matsuoka, "MACC: An OpenACC transpiler for automatic multi-GPU use," in *Supercomputing Frontiers*, R. Yokota and W. Wu, Eds. Cham: Springer International Publishing, 2018, pp. 109–127.

[6] C. Montgomery, J. L. Overbey, and X. Li, "Autotuning OpenACC work distribution via direct search." New York, NY, USA: Association for Computing Machinery, 2015.

[7] NVIDIA Corporation, "NVLink & NVSwitch: Advanced multi-GPU systems — NVIDIA," 2020. [Online]. Available: https://www.nvidia.com/en-us/data-center/nvlink/

[8] L. Li and B. Chapman, "Compiler assisted hybrid implicit and explicit GPU memory management under unified address space." New York, NY, USA: Association for Computing Machinery, 2019.

[9] M. Wolfe, S. Lee, J. Kim, X. Tian, R. Xu, B. Chapman, and S. Chandrasekaran, "The OpenACC data model: Preliminary study on its major challenges and implementations," *Parallel Computing*, vol. 78, pp. 15–27, 2018.

[10] P. A. Kulkarni, "JIT compilation policy for modern machines," vol. 46, no. 10, 2011.

[11] T. Komoda, S. Miwa, H. Nakamura, and N. Maruyama, "Integrating multi-GPU execution in an OpenACC compiler," in *2013 42nd International Conference on Parallel Processing (ICPP)*. IEEE, 2013, pp. 260–269.

[12] O. C. P. R. CCS), "XcodeML," 2009. [Online]. Available: https://omni-compiler.org/xcodeml.html

[13] S. Ghosh, T. Liao, H. Calandra, and B. M. Chapman, "Experiences with OpenMP, PGI, HMPP and OpenACC directives on ISO/TTI kernels," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, 2012, pp. 691–700.

[14] J. Hahnfeld, C. Terboven, J. Price, H. J. Pflug, and M. S. Müller, "Evaluation of asynchronous offloading capabilities of accelerator programming models for multiple devices," in *Fourth Workshop on Accelerator Programming Using Directives (WACCP)*, S. Chandrasekaran and G. Juckeland, Eds. Cham: Springer International Publishing, 2018, pp. 160–182.

[15] R. Xu, X. Tian, S. Chandrasekaran, Y. Yan, and B. Chapman, "Nas parallel benchmarks for GPGPUs using a directive-based programming model," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2014, pp. 67–81.

[16] J. A. Herdman, W. P. Gaudin, S. McIntosh-Smith, M. Boulton, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis, "Accelerating hydrocodes with OpenACC, OpenCL and CUDA," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, 2012, pp. 465–471.

[17] R. CCS, "Fiber," 2014. [Online]. Available: http://fiber-miniapp.github.io/

[18] ——, "Himeno benchmark," 2001. [Online]. Available: https://i.riken.jp/en/supercom/documents/himenobmt/

[19] S. P. E. Corporation, "SPEC ACCEL®," 2014. [Online]. Available: https://www.spec.org/accel/

[20] T. Diop, S. Gurfinkel, J. Anderson, and N. E. Jerger, "DistCL: A framework for the distributed execution of OpenCL kernels," in *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, 2013, pp. 556–566.

[21] X. Tian, D. Khaldi, D. Eachempati, R. Xu, and B. Chapman, "Optimizing GPU register usage: Extensions to OpenACC and compiler optimizations," in *2016 45th International Conference on Parallel Processing (ICPP)*, 2016, pp. 572–581.

[22] P. Barua, J. Shirako, and V. Sarkar, "Cost-driven thread coarsening for GPU kernels." New York, NY, USA: Association for Computing Machinery, 2018.

[23] M. Classen and M. Griebl, "Automatic code generation for distributed memory architectures in the polytope model," in *Proceedings 20th IEEE International Parallel Distributed Processing Symposium (IPDPS)*, 2006, pp. 7 pp.–.

[24] T. Ben-Nun, M. Sutton, S. Pai, and K. Pingali, "Groute: An asynchronous multi-GPU programming model for irregular computations." New York, NY, USA: Association for Computing Machinery, 2017.

[25] P. Pandit and R. Govindarajan, "Fluidic kernels: Cooperative execution of OpenCL programs on multiple heterogeneous devices." New York, NY, USA: Association for Computing Machinery, 2014.

[26] NVIDIA Corporation, "GitHub - NVIDIA/jitify." 2017. [Online]. Available: https://github.com/NVIDIA/jitify

[27] D. Mikushin, N. Likhogrud, E. Z. Zhang, and C. Bergström, "KernelGen – the design and implementation of a next generation compiler platform for accelerating numerical models on GPUs," in *2014 IEEE International Parallel Distributed Processing Symposium Workshops*, 2014, pp. 1011–1020.