HYDRA : Extending Shared Address Programming For Accelerator Clusters

Putt Sakdhnagool, Amit Sabne, and Rudolf Eigenmann

School of Electrical and Computer Engineering
Purdue University

Abstract. This work extends shared address programming to accelerator clusters by pursuing a simple form of shared-address programming, named HYDRA, where the programmer only specifies the parallel regions in the program. We present a fully automatic translation system that generates an MPI + accelerator program from a HYDRA program. Our mechanism ensures scalability of the generated program by optimizing data placement and transfer to and from the limited, discrete memories of accelerator devices. We also present a compiler design built on a high-level IR to support multiple accelerator architectures. Evaluation results demonstrate the scalability of the translated programs on five well-known benchmarks. On average, HYDRA gains a 24.54x speedup over single-accelerator performance when running on a 64-node Intel Xeon Phi cluster and a 27.56x speedup when running on a 64-node NVIDIA GPU cluster.

1 Introduction

The last decade has seen a steady rise in the use of accelerators towards high-performance computing. Many supercomputers rely on devices such as NVIDIA or AMD GPUs and Intel Xeon Phis to accelerate compute-intensive workloads. Various programming models and frameworks [8, 14] have so far been proposed to effectively use accelerators on individual compute nodes. As the productivity of these frameworks has risen over the years, there is growing interest in programming systems that can efficiently use accelerators on *all* nodes of a cluster.

Writing a program to exploit CPU clusters in itself is a tedious and error-prone task. The need for accelerator programming adds further to this difficulty, as the involved programming models differ substantially from those of common CPUs. To achieve greater productivity, high-level programming models for accelerator clusters are needed.

In response to such requirements, this paper presents compiler and runtime techniques required for a shared address programming model for accelerator clusters. In our research, we pursue a simple model, called *HYDRA*, where programmers only specify parallel regions and shared data in the program. From our observation, most parallel applications in well-known benchmark suites, such as Rodinia [4], can be implemented using only this construct. To demonstrate the effectiveness of our techniques, we developed a source-to-source translation

system that converts a HYDRA program into an MPI + accelerator program (referred to as accelerated MPI program hereafter).

There are two important performance factors for accelerator cluster programs: single-accelerator speed and scalability across nodes. Researchers have previously proposed advanced techniques for generating optimized single-accelerator code from shared address programs [8, 14]. By contrast, this paper focuses on the scalability aspect, which is crucial, as large clusters are expected to efficiently process increasingly large problem sizes. Optimization techniques for single accelerators are insufficient. Realizing shared address programming with high scalability on accelerator clusters poses the following three challenges. These challenges do not exist on CPU clusters. Their solution represent the specific contributions of this paper.

1. The first challenge comes from the fact that, unliked CPUs, current accelerators have discrete and limited memories. Full data allocation of today's typical problem sizes on accelerator memories could exceed available capacities. This limitation would result in failure of single-accelerator execution and an inability to scale to multiple nodes. Programmers of accelerated MPI code avoid this problem by allocating only the part of the data accessed by each process of a distributed program. By contrast, shared address programming hides the access distribution from programmers and, instead, relies on the compiler or runtime support to extract such information. The distribution of the data accesses is related to the partitioning of the program computation. The system must be aware of such partitioning to precisely allocate memory on accelerators. Without advanced analysis, a compiler may allocate the entire shared data on the accelerator memory, which could result in the said failure. Our first contribution overcomes this issue by introducing a precise compile-time memory allocation method.

2. A second critical issue is related to the data transfer between accelerator and host memory. Minimizing this transfer is critical for scalability. The challenge lies in the single machine image of the shared address space, where programmers do not specify data movements between CPU and accelerator memories. The compiler, having to derive such transfer from the program, might send entire shared data structures to/from accelerator memory, introducing excessive overhead. Our second contribution introduces a compile-time solution to minimize such transfers.

3. Both proposed techniques are architecture-agnostic. We show results on two common accelerators: NVIDIA's GPUs and Intel Xeon Phis (referred to as MIC hereafter). Our compiler design includes support for multiple architectures. Our third contribution lies in this design, which separates passes that are common across architectures and specialized passes for the target architectures. The compiler takes HYDRA programs as input, and translates them into accelerated MPI programs, using CUDA or OpenCL, depending upon the underlying architecture.

We demonstrate the efficacy of the proposed techniques by experimenting with five common applications on two clusters of 64 nodes each; one has

NVIDIA GPUs, the other has Intel MICs. The speedup against optimized single-accelerator performance is as high as 43.81x on a 64-node GPU cluster and 45.18x on a MIC cluster.

The remainder of this paper is organized as follows. Section 2 describes the baseline system on which HYDRA is built. Section 3 discusses the requirements for the translation and our solutions. Section 4 describes the implementation of the HYDRA translation system. Section 5 presents experimental results on five benchmarks. We discuss related work in Section 6 and present conclusions in Section 7.

2 Background

2.1 OMPD Baseline System

Our work builds on the OMPD [10] hybrid compiler-runtime system, which enables OpenMP programs to utilize nodes of a distributed system.

The compiler is responsible to partition the program computation and to perform the static part of the communication analysis.The compilation process of OMPD consists of two phases: (1) program partitioning and (2) static communication analysis. In program partitioning, the compiler divides the program into sections, referred to as *program blocks*, each containing either serial code or a parallel loop. The serial program blocks are replicated across processes while the parallel blocks are work-shared. The parallel loop's iterations are partitioned and distributed across MPI processes. A barrier is placed at the end of each program block, representing a potential communication point. The static communication analysis performs array data flow analysis, described in Section 2.2, determining local uses and local definitions of each program block. The compiler transfers this information to the runtime system for complete communication analysis.

All inter-node communication is generated and executed at runtime. At each barrier, the runtime system analyzes *global uses*, which determines future uses of all data at any needed communication point. The communication messages are determined by intersecting local definitions and global uses. The runtime system uses this information to schedule communication and generate MPI messages.

2.2 Array Data Flow Analysis

Array data flow analysis [11] enables the compiler to analyze the precise producer and consumer relationships between program blocks. The result of the analysis is a set of local uses and local definitions of each program block, at each barrier in the program. Every process will have its own local definitions and local uses. For shared array A at barrier i, the local use is denoted by $LUSE_i^A$ and local definition by $LDEF_i^A$, defined as

$$LUSE_i^A = \{use_{(i,j)}^A | 1 \leq j \leq n\} \tag{1}$$

$$LDEF_i^A = \{def_{(i,k)}^A | 1 \leq k \leq m\} \tag{2}$$

where each *use* represents a read access of array A in the program block after barrier i and each *def* represents a write access of array A in the program block before barrier i. n and m are the number of read accesses in the program block after barrier i and the number of write accesses in the program block before barrier i of array A, respectively. For a p-dimensional array A, each *use* and *def* is defined as a pair of lower bound and upper bound accesses in each dimension of the array. For dimension d, the lower and upper bound are represented as $[lb_d : ub_d]$. An example of *use* and *def* for a p-dimensional array is as follows

$$use^A_{(i,j)} = [lb_{p-1} : ub_{p-1}]...[lb_1 : ub_1][lb_0 : ub_0]$$
$$def^A_{(i,j)} = [lb_{p-1} : ub_{p-1}]...[lb_1 : ub_1][lb_0 : ub_0]$$

We extend this array data flow analysis framework for the new optimizations described in Section 3.

3 Extending Shared Address Programming Beyond CPU Clusters

Extending CPU-based shared address programming to support accelerator clusters poses a number of challenges. While the model is convenient for users, the programs abstraction hides information that is relevant for the translator. Thus, the compiler needs sophisticated techniques to extract this information. The need for such techniques is critical in our HYDRA programming model, as programmers only specify parallel regions and do not include such information as data transfer and communication. Section 3.1 describes the model in more detail.

Our techniques deal with the fact that accelerators are independent computational components with separate address spaces, reduced memory capacities, and diverse architectures. Section 3.2 explains these threee challenges in more detail and presents our solutions.

3.1 HYDRA Programming Model

HYDRA is a directive-based shared address programming model offering a single parallel loop construct

```
#pragma hydra parallel for [clauses]
```

The clauses are syntactically optional but might be needed for program semantics. Table 1 lists all available clauses for the HYDRA parallel loop directive. The `shared`, `private`, and `firstprivate` clauses specify characteristics of variables. Variables not listed explicitly are `shared` by default. The `reduction` clause indicates that the annotated loop performs a reduction operation on variables in `varlist` using operator `op`.

Despite HYDRA's simplicity, many parallel applications can be implemented using only this single HYDRA construct. All of our evaluation benchmarks were available in the form of OpenMP programs. We generated HYDRA versions by

Table 1. Parallel Loop Directive Clauses

Clause	Format	Description
`shared`	`shared(varlist)`	List of shared variables.
`private`	`private(varlist)`	List of private variables.
`firstprivate`	`firstprivate(varlist)`	List of private variables, whose value must be initiated before the start of the parallel loop.
`reduction`	`reduction(op:varlist)`	List of variables to perform reduction with operator `op`.

a simple, syntactic translation. We chose HYDRA instead of available models, such as OpenACC and OpenMP, for research purposed, which are to explore the concepts of the translation and the generic characteristic of shared-address models.

3.2 Compiler Analyses for Accelerator Data Management

In distributed programming, the computation is partitioned and distributed across processes. The programmer is responsible for doing so. HYDRA instead holds the underlying compiler and runtime responsible for these tasks. Programmers do not need to express any information about access ranges of shared data.

The lack of such information may tell the compiler that each process is accessing the entire data, although in reality, only a portion of the data is being accessed. This problem is not critical in CPU clusters because of large physical memory space and virtual address systems; however, accelerator memory is much smaller and does not have a virtual memory system. As the typical problem sizes used on clusters are much larger than a single accelerator's memory, allocating the entire data required by the computation on each accelerator would result in program failure due to insufficient memory. Even if the data fits in the accelerator memory, another issue would arise: accelerator memory is discrete and input data must be transferred to it before being used. Transferring the entire data would introduce excessive overhead. Therefore, data access information is crucial to the scalability of accelerator cluster programs.

Data Transfer Analysis To minimize data transfers, a compiler analysis must precisely identify the data accessed by each program block. The precise access information can be identified by the union of read and write sections of live data. The details of the analysis are as follows: The first part of our data transfer analysis identifies the shared data that are live-in and live-out of a given program block executing on the accelerators, B_i. This information can be derived from the $LUSE$ information, generated by the array data flow analysis described in Section 2.2.

Let Br_i denote the barrier before B_i, Br_f denote the future barriers that the program will reach after B_i, and $ShareVar(B_i)$ denote the set of shared variables accessed in the program block B_i. Let $A \in ShareVar(B_i)$. If there exists $LUSE^A_{Br_i}$, array A will be used in B_i and a data transfer from host to accelerator is required. On the other hand, if there exists $LUSE^A_{Br_f}$ array A will be used in the future, requiring a data transfer from accelerator to host.

If it is determined that a data transfer is required for an array A at barrier Br_i, the next step is to identify the section of array A that will be transferred. The required section of an array A on each dimension can be obtained as $[lb_{min,Br_i} : ub_{max,Br_i}]$ where lb_{min,Br_i} is the minimum lower bound of all local accesses of array A and ub_{max,Br_i} is the maximum upper bound of all local accesses of array A at barrier Br_i in that dimension. Note that the upper and lower bounds can be symbolic expressions. The analysis obtains lb_{min,Br_i} and ub_{max,Br_i} by using the symbolic analysis capabilities of Cetus [1].

Memory Allocation Optimization Memory allocation/deallocation could be done at the beginning/end of each kernel, based on the data size computed for the transfer. However, as the same array may be accessed in multiple kernels, one can do better. Our method performs global analysis to summarize all accesses of the shared array in the program and allocates/deallocates only once, saving costs and improving re-use of the allocated memory. There is a small sacrifice in precision, in terms of the memory size allocated, which however is always conservatively larger. Such sacrifice does not affect the correctness of the program and is outweighed by the saved costs of repeated allocation and possible re-transfer of data.

The optimization is based upon global array dataflow analysis for precise array sections. The implementation also makes use of the advanced array dataflow framework and symbolic analysis capabilities available in the Cetus compiler infrastructure. The memory space requirement of an array A is extracted from the union of $LDEF^A$ and $LUSE^A$, where $LUSE^A$ represents all read accesses and $LDEF^A$ represents all write accesses of an array A in the program. $LDEF^A$ is the union of all $LUSE^A_{Br_i}$ and $LDEF^A$ is the union of all $LDEF^A_{Br_i}$ in the program. Thus, $LUSE^A \cup LDEF^a$ represents all accesses of array a in the program. The memory requirement for each dimension of the array can be defined as $[lb_{min} : ub_{max}]$ where $lb_{min} \in (LUSE^A \cup LDEF^A)$ is the minimum lower bound of all accesses of array A and $ub_{max} \in (LUSE^A \cup LDEF^A)$ is the maximum upper bound of all accesses of array A. $[lb_{min} : ub_{max}]$ indicates the bounds of any access to array A in the local process. Thus, it also defines the memory allocation for array A. The size of the new array is different from the original. The compiler must incorporate this change into all accesses of the new array by subtracting lb_{min} from all indices. The size and offset information is also utilized while generating the data transfers.

The analysis does not require array sections to be contiguous and can support arrays with any number of dimensions. In our current implementation, if the analysis results in multiple array sections, the algorithm will conservatively merge them together. Further analysis can be done to determine whether the sections should be merged or not, which we leave to future work.

4 Translation System Implementation

The HYDRA translation system consists of a compiler and a runtime system. The compiler performs source-to-source translation to generate accelerated MPI

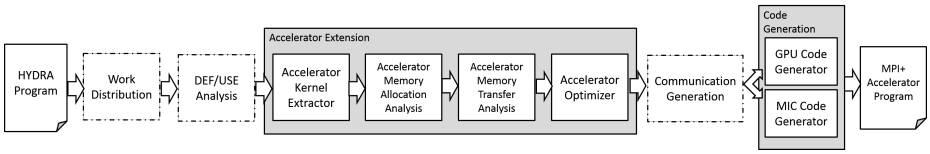


Fig. 1. HYDRA Compiler Translation Process: Grey boxes represent the new passes in the HYDRA compiler.

code from input HYDRA programs. Section 4.1 explains the compiler design to support multiple accelerator architectures. Section 4.2 presents the overall translation process of the HYDRA compiler. The HYDRA runtime system is responsible for remote accelerator-to-accelerator communication in the compiler-translated, accelerated MPI programs. The implementation of the runtime system is described in Section 4.3.

4.1 Supporting Multiple Accelerator Architectures

To support a wide-range of accelerator clusters, the compiler must be able to target different accelerator architectures. This requirement poses a challenge to the compiler design as different architectures have different features, some of which are common while others are unique to the specific architecture.

In the HYDRA compiler, most compilation passes are architecture-agnostic with no specialization needed. The design defers specialization to as late as possible in the translation process. In this way, only the last compilation pass of code generation is architecture specific. The key to realizing such design is the internal representation (IR).

From our observation the following four operations are sufficient to express any accelerator program : (1) Memory Allocation, (2) Data Transfer, (3) Accelerator Kernel Execution, and (4) Memory Deallocation. By using these operations as IR constructs, the compiler can represent programs in an architecture-independent form. To generate architecture-specific code, the compiler converts architecture-independent constructs to their architecture-specific equivalents during the code generation pass.

4.2 HYDRA Translation Process

Fig. 1 shows the overall translation process from the input HYDRA program to the accelerated MPI program. Accelerator extensions are highlighted using grey boxes. The dashed boxes represent existing CPU passes.

The compilation process starts with the CPU passes, which perform work partitioning and array dataflow analysis. The partitioned program is then passed to HYDRA's accelerator extension. The passes in the extension perform accelerator kernel generation, memory transfer analysis, memory allocation optimization and further architecture-independent optimization (e.g. hoisting memory transfers, prefetching, etc.). After the accelerator code is added, the compiler

analyzes and adds communication code to the program. The compilation process completes with the code generation pass, which produces the accelerated MPI program with accelerator kernels specific to the target architecture.

The current implementation of the HYDRA compiler supports two accelerator types: NVIDIA CUDA GPUs and Intel MIC. As target languages, we choose CUDA for NVIDIA GPUs and OpenCL for Intel MICs. One might argue that different architectures could be supported by using OpenCL as the target language for all accelerator architectures; the compiler just needs to generate OpenCL + MPI programs, allowing the generated code to run on any accelerator cluster. However, OpenCL does not support accelerator-specific features, e.g. using warp-level functions in CUDA. Thus, the translated code cannot fully utilize the accelerator capabilities. Further, some architectures have limited support for OpenCL features [6].

The HYDRA compiler faces similar limitations as the baseline OMPD system: irregular programs are handled inefficiently for lack of compile-time information about data accesses. Such accesses may lead to conservative memory allocations and data transfers.

4.3 HYDRA Runtime System

The HYDRA runtime system is responsible for remote accelerator communication. In contrast to CPUs, accelerators cannot directly perform remote communication. The communication must be handled by the host CPU. Thus, additional data transfer between host and accelerator memories is required before and after the communication. We refer to such data transfer as *message relay*.

We designed a new runtime extension (ACC-RT), whose interaction with the host-side runtime system (HOST-RT) enables remote accelerator communication. The HOST-RT system is responsible for generating communication messages and executing host-to-host communication, while the ACC-RT system is responsible for managing host-accelerator data mapping and message relays. The ACC-RT system uses communication information from the HOST-RT system to generate message relays. The transfers are computed from the communication messages generated by the HOST-RT system, and the mapping information provided by the HYDRA compiler. A runtime interface is designed for the compiler to provide mapping information between host and accelerator data. The mapping information includes the host address, accelerator address, accelerator data size, and accelerator data offset. The accelerator offset is necessary in order to align accelerator and host data. The overhead of the ACC-RT system is negligible. In our experiments, we found this overhead to be less than 0.1% of the total execution time on 64-node accelerator clusters.

5 Evaluation

This section evaluates the effectiveness of the proposed techniques on two accelerator clusters, one with NVIDIA GPUs and another with Intel MICs.

Table 2. Experimental Setup for Strong Scaling

Benchmark	class-A Problem Size	class-B Problem Size	Number of Iterations
Jacobi	20000×20000	24000×24000	1000
Heat3D	$768 \times 768 \times 768$	$800 \times 800 \times 800$	1000
Blackscholes	67,000,000 options	400,000,000 options	1000
Bilateral Filter	12280×12280	20000×20000	1
Filterbank	67,000,000	134,000,000	32

5.1 Experimental Setup

We used the Keeneland cluster [20] to evaluate the GPU versions of the HYDRA programs. Keeneland consists of 264 compute nodes, connected by an FDR Infiniband network. Each node has two 8-core Xeon E5-2670 running at 2.6 Ghz, 32GB of main memory, and three NVIDIA Tesla M2090 GPUs. Each GPU has 6 GB of device memory available for computation. We evaluated the MIC program versions on a community cluster, where each node contains two 8-core Xeon E5-2670 CPUs, 64 GB of main memory, and two Intel Xeon Phi P5110 accelerators. Each Xeon Phi has 6 GB of device memory available for computation. The nodes are connected by an FDR-10 Infiniband network. Our evaluation uses up to 64 nodes with one MPI process and one accelerator per node.

We present the results for five representative benchmarks: Bilateral Filter, Blackscholes, Filterbank, Jacobi, and Heat3D. Bilateral Filter and Blackscholes are from the NVIDIA CUDA SDK. The benchmarks are implemented in HYDRA by converting their OpenMP counterparts. Bilateral Filter is a non-linear and edge-preserving filter used for noise reduction and image recovery. It uses a weighted average of intensity values from nearby pixels to update the intensity value of each individual image pixel. Blackscholes is a financial formula to compute the fair call and put prices for options. Filterbank is from StreamIt [19] benchmark suite. The benchmark creates a filter bank to perform multi-rate signal processing. Jacobi is a two-dimensional 5-point stencil computation that solves Laplace equations using Jacobi iterations. Heat3D is a three-dimensional 7-point stencil computation that solves a heat equation. Both Jacobi and Heat3D are common computations in scientific applications. These benchmarks represent a class of applications and computations that perform well on single-accelerator systems, and thus can be expected to take advantage of accelerator clusters.

5.2 Scalability

Strong scaling In the strong-scaling test, the problem size is kept fixed and the number of processes is varied. We use two problem sizes for each benchmark: *class-A* and *class-B*. A class-A problem is small enough to fit the entire computation data in a single accelerator's memory. A class-B problem requires more than one accelerator to execute, since the memory requirement exceeds the capacity of a single accelerator. Table 2 shows the setting of each problem class.

Fig. 2 shows the results for both MIC and GPU clusters. HYDRA programs with class-A problems achieve an average of 24.54x speedup on the 64-nodes MIC

Table 3. Experimental Setup for Weak Scaling

Benchmark	MIC Problem Size	GPU Problem Size	Number of Iterations
Jacobi	8192×8192	8192×8192	100
Heat3D	$512 \times 512 \times 512$	$450 \times 450 \times 450$	100
Blackscholes	67,000,000 options	32,000,000 options	100
Bilateral Filter	5500×5500	5500×5500	1
Filterbank	4,000,000	4,000,000	32

cluster and 27.56x speedup on the GPU cluster. The maximum speedup is 45.18x on the MIC cluster and 43.81x on the GPU cluster. The speedup is calculated against a single accelerator execution time. We show the average speedup only on class-A problems because they can correctly execute on a single node. For class-B problems, the performance is compared against the performance of a configuration with the smallest number of accelerators that allow the program to be executed successfully. Our result shows that Jacobi, Heat3D, and Blackscholes have good scalability on both MIC and GPU clusters.

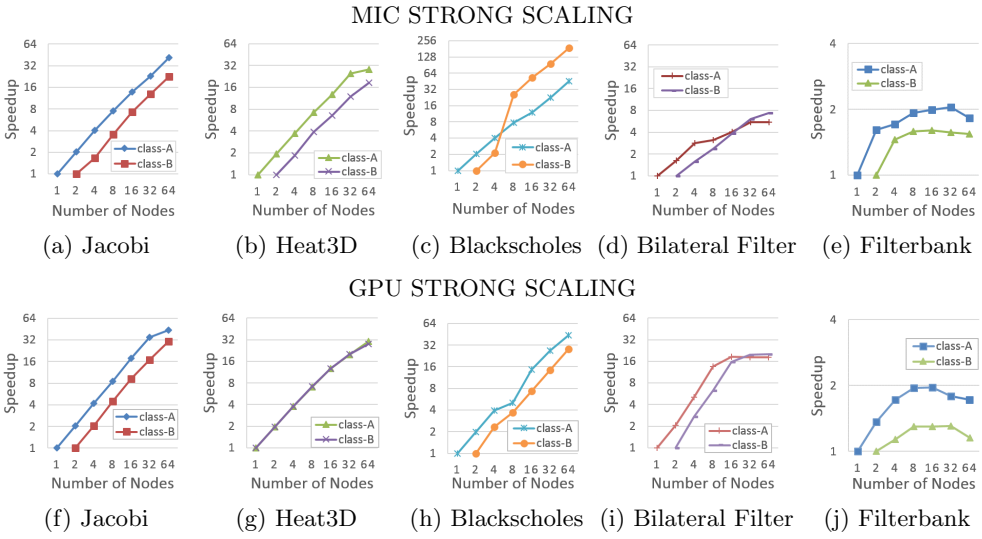
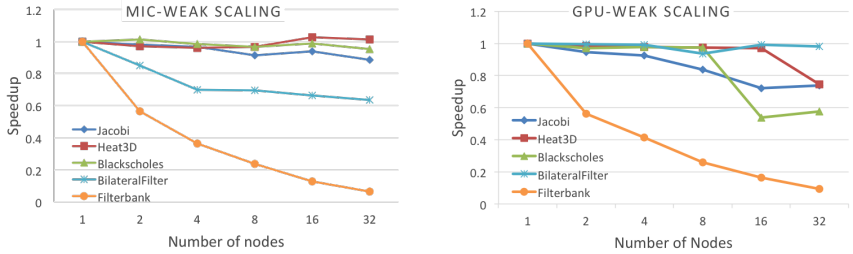


Fig. 2. Strong scaling experimental results of five benchmarks on MIC cluster(a-e) and GPU cluster (f-j). The speedup of the class-A problem is relative to a single-node performance. The speedup of class-B problem is relative to the performance of a configuration with the smallest number of accelerators that allow the program to be executed successfully.

Bilateral Filter shows limited scalability on both MIC and GPU clusters. The lack of coalesced memory accesses inside the accelerator kernel leads to inefficient execution, limiting performance gained by node-level parallelism. With 64 nodes, the speedup is 5.49x on MICs and 18.24x on GPUs. More advanced compiler analysis may enable coalesced memory accesses, thus improving the scalability of the generated program. Filterbank also exhibits scalability limita-



(a) Weak scaling-MIC (b) Weak scaling-GPU

Fig. 3. Weak scaling results of five benchmarks on MIC cluster(a) and GPU cluster(b). The speedup shown is against the execution of a single-accelerator single-node setup.

tion on both MIC and GPU clusters. In contrast to Bilateral Filter, the cause of the limitation is the conservative methods of the array data flow analysis. The analysis summarizes memory accesses by all paths of conditional branches inside the parallel loops, resulting in extra broadcast communications.

On the MIC cluster, Blackscholes with class-B problem size shows super-linear speedup when the number of nodes increases from 4 to 8. The reason lies in the data transfers inside the iteration loop. The transfer on 4 nodes is 22.14x slower than on 8 nodes due to MIC's driver issue. This difference in data transfer time contributes to the super-linear speedup. This transfer could have been hoisted out of the iteration loop, however, automatic compiler hoisting did not take place in this case due to implementation limitations. We tried hoisting this transfer out of the loop manually, and observed that the achieved performance showed linear scaling, as in the class-A problem.

Weak scaling In the weak scaling test, the problem size is increased as the number of processes increases. The problem size per process is fixed. Table 3 shows the problem sizes per compute node used in the weak scaling experiment on the GPU and MIC clusters. We performed this experiment using up to 32 accelerators. Fig. 3 shows the weak scaling results of both MIC and GPU clusters. The speedup is calculated over the execution time of a single node with one accelerator.

Jacobi, Heat3D, Bilateral Filter, and Blackscholes achieve high scalability in the weak scaling test. Filterbank performs the worst in terms of the scalability owing to excessive broadcast communication caused by the conservative array data flow analysis. Note that the achieved scalability is better on the MIC cluster than on the GPU one. This is because, on average, the accelerator execution time is greater on MICs than that for GPUs. Therefore, the communication overhead has a bigger impact on the scalability in the GPU cluster.

5.3 Memory Allocation

In this experiment, we show only weak-scaling results on the MIC cluster. The other tests exhibited similar trends. Fig. 4 shows the memory allocation require-

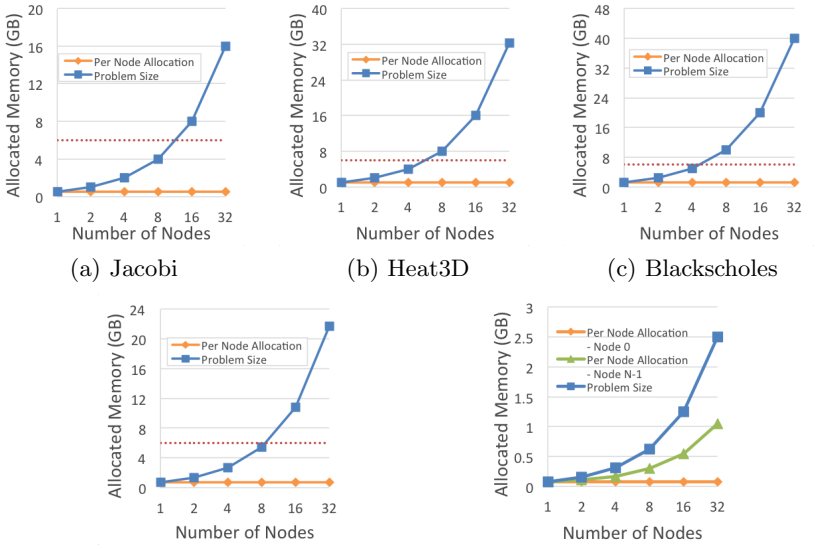


Fig. 4. Accelerator memory allocation in weak-scaling experiments on MIC cluster.

ment for each benchmark in the weak scaling experiment on the MIC cluster. Each chart shows the total amount of memory required by the entire problem and the amount of memory actually allocated on the accelerator for each benchmark. For all benchmarks, except Filterbank, the size of allocated memory on the accelerator memory is fixed as the number of nodes increases. The dotted line indicates the single accelerator memory limitation. It shows the scaling limit if the memory allocation optimization is not implemented. Without memory allocation optimization, Jacobi cannot exploit more than 8 nodes, while Heat3D, Blackscholes, and Bilateral Filter benchmarks cannot run beyond 4 nodes.

Unlike other benchmarks, the accelerator memories allocated by each process are different for Filterbank. We report the minimum memory (required by process 0) and the maximum memory (required by process N-1) in Fig. 4e. For process 0, the accelerator memory requirement remains the same for any problem size. For other processes (1 to N-1), however, the memory requirement grows with the problem size. This behavior is explained by the conservative array data flow analysis employed by HYDRA that results in over-allocation in the presence of conditional branches.

6 Related Work

Programming Models for Accelerator Clusters Several previous efforts proposed programming models for accelerator clusters. OmpSs [2, 3] considers a directive-based shared address programming model. This model requires the users to provide extra information to the compiler about computation offload-

ing and data transfers. Programmers use data region to specify accessed regions of shared data; the underlying runtime system then manages the allocations and transfers of these regions. Several other approaches [13, 16] extend a PGAS (Partitioned Global Address Space) language to support GPU clusters. PGAS languages require programmers to specify thread and data affinity explicitly. In contrast to this work, the HYDRA compiler derives the required information automatically from HYDRA programs, which are easier to write than PGAS programs. SnuCL [9] extends an OpenCL framework to run on CPU/GPU clusters. The SnuCL runtime system provides a single machine image, which allows single-node OpenCL programs to run on the cluster. In contrast to our work, SnuCL programmers still face the programming complexity of accelerator programming. Moreover, as we discussed earlier, OpenCL is not fully portable. Programmers need to customize OpenCL programs for each architecture to fully utilize CPUs and accelerators.

Memory Management and Communication Memory management and communication are innate to any shared address programming model. Several previous efforts proposed shared address programming models for CPU clusters. There are two major approaches for memory allocation management in these contributions. The first approach is to rely on the underlying operating system or runtime system. For example, in OMPD [10], each process allocates the entire shared data in every process and lets the virtual memory system allocate the required physical memory when the data is accessed. This solution is not feasible on accelerators because of the lack of virtual address space on GPUs and the lack of swap space on MICs. Another example is Software Distributed Shared Memory (SDSM) [7]. The SDSM runtime system provides a shared address abstraction of the distributed system. The performance of this approach has remained far below that of MPI programming. Another approach relies on information provided by the programmers. In High Performance Fortran (HPF) [18], programmers explicitly provide data partitioning information through directives. In PGAS languages, such as UPC [5], Co-array Fortran [15], and Titanium [21] , the programmers explicitly specify the affinity between processes and data. In contrast to these systems, HYDRA neither requires additional directives nor relies on the operating system. On the GPU side, Ramashekar and Bondhugula [17] proposed a hybrid compiler-runtime analysis, based upon the polyhedral model, to automate data allocation on multi-GPU machines. In contrast to their work, HYDRA uses symbolic analysis to perform compile-time memory allocation and transfer analyses targeting accelerator clusters and provides a complete translation system for multiple accelerator types. NVIDIA introduced Unified Memory Access to simplify memory management on GPUs; however, this system incurs high overhead [12].

7 Conclusion

We have introduced compile-time and runtime techniques for extending shared address programs for execution on accelerator clusters of multiple types.

The paper presented two novel, architecture-agnostic compile-time analyses, which ensure scalability of the translated program. We also presented a runtime system to support accelerator communication. To show the effectiveness of these analyses, we developed a source-to-source translation system that generated an accelerated MPI program from a simple shared address programming model called HYDRA. To support the architecture-agnostic nature of the proposed technique, a careful compiler design was presented. We demonstrate this design for two common accelerators: NVIDIA GPUs and Intel Xeon Phi. With the proposed techniques, we showed that the simple form of shared address programming can be extended to accelerator clusters without additional involvement of programmers.

HYDRA can achieve an average speedup of 24.54x against a single-accelerator performance when running on a 64-node cluster with Intel Xeon Phis and a 27.56x speedup when running on 64 nodes with NVIDIA GPUs. We also showed that our single-node performance is comparable to, or better than, a state-of-the-art OpenMP-to-CUDA translation system. There are additional opportunities for performance enhancements in our system for both computation and communication. Ongoing work is exploring these opportunities.

Acknowledgments

This work was supported, in part, by the National Science Foundation under grants No. 0916817-CCF and 1449258-ACI. This research used resources of the Keeneland Computing Facility at the Georgia Institute of Technology and the Extreme Sciene and Engineering Discovery Environment (XSEDE), which are supported by the National Science Foundation under awards OCI-0910735 and ACI-1053575, respectively.

References

1. Bae, H., Mustafa, D., Lee, J.W., Aurangzeb, Lin, H., Dave, C., Eigenmann, R., Midkiff, S.: The Cetus source-to-source compiler infrastructure: Overview and evaluation. International Journal of Parallel Programming pp. 1–15 (2012)
2. Bueno, J., Planas, J., Duran, A., Badia, R., Martorell, X., Ayguade, E., Labarta, J.: Productive programming of GPU clusters with OmpSs. In: Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International. pp. 557–568 (May 2012)
3. Bueno, J., Martorell, X., Badia, R.M., Ayguadé, E., Labarta, J.: Implementing OmpSs support for regions of data in architectures with multiple address spaces. In: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing. pp. 359–368. ACM, New York, NY, USA (2013)
4. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.H., Skadron, K.: Rodinia: A benchmark suite for heterogeneous computing. In: Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC). pp. 44–54. IISWC '09, IEEE Computer Society, Washington, DC, USA (2009)
5. UPC Consortium: UPC language specifications, v1.2. Tech Report LBNL-59208, Lawrence Berkeley National Lab (2005)

6. Intel Corporation: Intel® SDK for OpenCL applications XE 2013 R3, https://software.intel.com/sites/products/documentation/ioclsdk/2013XE/UG/index.htm
7. Dwarkadas, S., Cox, A.L., Zwaenepoel, W.: An integrated compile-time/run-time software distributed shared memory system. In: Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 186–197. ASPLOS VII, ACM, New York, NY, USA (1996)
8. Han, T.D., Abdelrahman, T.S.: hiCUDA: High-level GPGPU programming. IEEE Transactions on Parallel and Distributed Systems 22, 78–90 (2011)
9. Kim, J., Seo, S., Lee, J., Nah, J., Jo, G., Lee, J.: SnuCL: An OpenCL framework for heterogeneous CPU/GPU clusters. In: Proceedings of the 26th ACM International Conference on Supercomputing. pp. 341–352. ICS '12, ACM, New York, NY, USA (2012)
10. Kwon, O., Jubair, F., Eigenmann, R., Midkiff, S.: A hybrid approach of OpenMP for clusters. In: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 75–84 (2012)
11. Kwon, O., Jubair, F., Min, S.J., Bae, H., Eigenmann, R., Midkiff, S.: Automatic scaling of openmp beyond shared memory. In: Rajopadhye, S., Mills Strout, M. (eds.) Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science, vol. 7146, pp. 1–15. Springer Berlin Heidelberg (2013)
12. Landaverde, R., Zhang, T., Coskun, A.K., Herbordt, M.: An investigation of unified memory access performance in cuda. In: Proceedings of the IEEE High Performance Extreme Computing Conference (2014)
13. Lee, J., Tran, M., Odajima, T., Boku, T., Sato, M.: An extension of XcalableMP PGAS lanaguage for multi-node GPU clusters. In: Euro-Par 2011: Parallel Processing Workshops, Lecture Notes in Computer Science, vol. 7155, pp. 429–439. Springer Berlin Heidelberg (2012)
14. Lee, S., Eigenmann, R.: OpenMPC: Extended OpenMP programming and tuning for GPUs. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–11 (2010)
15. Numrich, R.W., Reid, J.: Co-array fortran for parallel programming. SIGPLAN Fortran Forum 17(2), 1–31 (Aug 1998)
16. Potluri, S., Bureddy, D., Wang, H., Subramoni, H., Panda, D.: Extending OpenSHMEM for GPU computing. In: 2013 IEEE 27th International Symposium on Parallel Distributed Processing (IPDPS). pp. 1001–1012 (May 2013)
17. Ramashekar, T., Bondhugula, U.: Automatic data allocation and buffer management for multi-GPU machines. ACM Trans. Archit. Code Optim. 10(4), 60:1–60:26 (Dec 2013)
18. High Performance Fortran Forum: High performance fortran language specification. SIGPLAN Fortran Forum 12(4), 1–86 (Dec 1993)
19. Thies, W., Karczmarek, M., Gordon, M.I., Maze, D.Z., Wong, J., Hoffman, H., Brown, M., Amarasinghe, S.: Streamit: A compiler for streaming applications. Technical Report MIT/LCS Technical Memo LCS-TM-622, Massachusetts Institute of Technology, Cambridge, MA (Dec 2001)
20. Vetter, J., Glassbrook, R., Dongarra, J., Schwan, K., Loftis, B., McNally, S., Meredith, J., Rogers, J., Roth, P., Spafford, K., Yalamanchili, S.: Keeneland: Bringing heterogeneous gpu computing to the computational science community. Computing in Science Engineering 13(5), 90–95 (Sept 2011)
21. Yelick, K., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P., Graham, S., Gay, D., Colella, P., Aiken, A.: Titanium: A high-performance Java dialect. In: In ACM. pp. 10–11 (1998)