LLVM-based Communication Optimizations for PGAS Programs

Akihiro Hayashi
Rice University
ahayashi@rice.edu

Jisheng Zhao
Rice University
jisheng.zhao@rice.edu

Michael Ferguson
Cray Inc.
mferguson@cray.com

Vivek Sarkar
Rice University
vsarkar@rice.edu

ABSTRACT

While Partitioned Global Address Space (PGAS) programming languages such as UPC/UPC++, CAF, Chapel and X10 provide high-level programming models for facilitating large-scale distributed-memory parallel programming, it is widely recognized that compiler analysis and optimization for these languages has been very limited, unlike the optimization of SMP models such as OpenMP. One reason for this limitation is that current optimizers for PGAS programs are specialized to different languages. This is unfortunate since communication optimization is an important class of compiler optimizations for PGAS programs running on distributed-memory platforms, and these optimizations need to be performed more widely. Thus, a more effective approach would be to build a language-independent and runtime-independent compiler framework for optimizing PGAS programs so that new communication optimizations can be leveraged by different languages.

To address this need, we introduce an LLVM-based (Low Level Virtual Machine) communication optimization framework. Our compilation system leverages existing optimization passes and introduces new PGAS language-aware runtime dependent/independent passes to reduce communication overheads. Our experimental results show an average performance improvement of 3.5× and 3.4× on 64-nodes of a Cray XC30^{TM} supercomputer and 32-nodes of a Westmere cluster respectively, for a set of benchmarks written in the Chapel language. Overall, we show that our new LLVM-based compiler optimization framework can effectively improve the performance of PGAS programs.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Compilers

Keywords

Chapel, PGAS Languages, LLVM, Communication Optimizations

LLVM-HPC2015, November 15-20, 2015, Austin, TX, USA
© 2015 ACM. ISBN 978-1-4503-4005-2/15/11...$15.00
DOI: http://dx.doi.org/10.1145/2833157.2833164

1. INTRODUCTION

PGAS (Partitioned Global Address Space) programming languages such as Chapel, Co-array Fortran, Habanero-C, Unified Parallel C (UPC), UPC++, and X10 [9, 24, 11, 13, 33, 10] support highly productive programming models. PGAS programming languages aim to reduce the complexity of writing distributed-memory parallel programs compared to conventional message-passing programming models such as MPI [15]. They also aim to simplify parallel programming by introducing global operations on distributed arrays, distributed task parallelism, directed synchronization and mutual exclusion. These high-level language constructs offer many opportunities for enabling compiler developers to build parallelism-aware program optimization. For example, some prior work [4] focused on reducing communication overheads across multiple nodes for distributed X10 programs. The UPC compiler also introduced optimizations to reduce communication overheads by recognizing language-specific runtime API calls [6].

However, these past efforts have been specific to different languages and runtimes. In contrast, we believe it is feasible to build a common compiler parallel intermediate representation (PIR [32]) that supports new optimizations for multiple PGAS languages because these languages have semantically similar features for expressing parallelism, synchronization, and communication. To support the PIR based compiler infrastructure that can uniformly optimize PGAS programs, we chose LLVM (Low Level Virtual Machine) [21] since LLVM is widely used and provides modern language- and target-independent optimization passes.

In this work, we demonstrate LLVM-based optimizations for Chapel programs, with an initial focus on communication optimizations. However, these optimizations are applicable to any PGAS language that supports translation to LLVM. We built an LLVM-based Chapel compiler that takes Chapel programs, creates LLVM IRs, applies LLVM optimization passes, and generate target code. Our compilation framework uses the address space feature in LLVM to distinguish between remote and local data, which corresponds to a PIR representing remote/local communications, thereby providing a language and runtime-independent representation for the local/remote data communications. The remote data accesses can be expressed as LLVM load/store/memcpy instructions that involve a designated address space (e.g. address space 100). This means that classic LLVM optimization passes (e.g. loop invariant code motion) can be applied to eliminate redundant data access even for remote data. Additionally, the compiler performs several PGAS-aware communication optimizations. Finally, instructions involving remote address space pointers that remain are lowered to runtime communication API calls such as *get* and *put* that operate on

```
1 // before LICM (Chapel)
2  var remoteData : int;   // On Node X
3  ...
4  for i in 1..N { // On Node Y
5    data = remoteData;   // from Node X
6    ...;
7  }
```

Figure 1: Communication optimization example for Chapel pseudo-code. For each Chapel example, the loop is executed on node Y and reads *remoteData* from Node X. In the case that *remoteData* is a loop invariant, the loop invariant code motion (LICM) can be applied to reduce redundant communications.

the remote pointers.

This paper makes the following contributions:

1. A uniform LLVM-based communication optimization framework for PGAS programs.
2. An optimization strategy using the address space feature in LLVM to enable existing LLVM optimizations to apply to operations on both local and remote data.
3. New LLVM-based optimizations built on top of our framework.
4. Performance evaluation of LLVM-based Chapel compiler on a Intel Xeon Cluster and a Cray XC30^{TM} system.

The rest of paper is organized as follows. In Section 2, we summarize background on PGAS languages. Section 3 describes our LLVM-based communication optimization framework and how it can be used to leverage existing LLVM optimizations. Section 4 summarizes new LLVM optimizations that can be used to reduce communication operations in PGAS programs. Section 5 presents an experimental evaluation. We discuss related work in Section 6 and conclude in Section 7.

2. PGAS LANGUAGES

Parallel computing is now moving to the *Extreme Scale* [28] computing era and this poses challenging problems in software development. In the context of parallel programming, message-passing programming models with communication APIs such as MPI (Message Passing Interface) [15] are ubiquitous. These models adopt an SPMD (Single Program Multiple Data) programming paradigm[1], which simplifies the runtime support needed to manage large number of processors. The use of MPI, however, adds complexity since programmers have to orchestrate data transfers across large number of nodes by using MPI's inter-node communication library, which does not support a global address space[2].

In contrast, PGAS (Partitioned Global Address Space) programming languages such as Co-Array Fortran, UPC, UPC++, Titanium, Chapel and X10 [24, 13, 33, 16, 9, 10] aim to increase productivity and portability by providing high-level language features that include explicit task parallelism, data distribution, operations on global or distributed arrays, and synchronizations. For example, each array and variable can be either remote or local, and remote data operations can be expressed as just an assignment statement (e.g "data = remoteData;" in Figure 1). The compiler will translate the assignment statement to appropriate *put* / *get* API calls in communication layers like GASNet [7], OpenSHMEM [27], PAMI [20], or MPI [29].

[1]MPMD was also introduced in MPI 2.0.
[2]Recent extensions to the MPI standard now included support for one-sided communications, but that it's still not the same as support for a global address space.

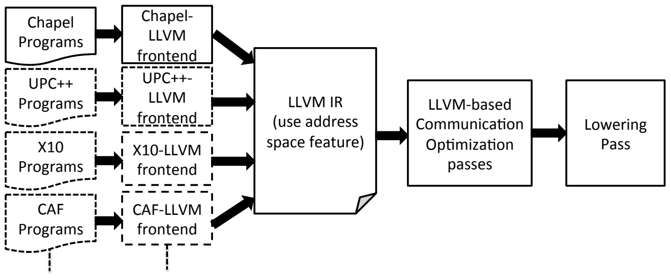

Figure 2: LLVM-based Communication Optimization Framework.

3. ENABLING EXISTING LLVM PASSES FOR COMMUNICATION OPTIMIZATIONS

In this section, we introduce the communication optimization framework to reduce redundant remote data accesses, and thereby improves the efficiency of data communication in PGAS programs.

The structure of our compiler optimization framework is presented in Figure 2. First, a language frontend generates LLVM IR with some properties (see Section 3.1 and Section 3.2). The LLVM IR is referred to as the PGAS LLVM IR in the following. Then our framework applies LLVM-based optimization passes to perform communication optimization (see Section 3.3). Finally, remote references are lowered to communication API calls such as GASNet get/put (see Section 3.4). Our current implementation supports the Chapel language.

3.1 PGAS LLVM IR

Since PGAS language constructs have explicit information on parallelism, PGAS programming languages offer opportunities for enabling the compiler to perform parallel-aware optimizations. Unlike prior studies [4, 6] which focus on optimizations for specific languages, we believe it is feasible to build a compiler infrastructure that can uniformly optimize PGAS programs. The key idea is to build a common compiler backend for PGAS program languages which enables compiler writers to build language- and target- independent compiler optimizations.

To motivate our framework, we show how the classical Loop Invariant Code Motion (LICM) optimization can be used to reduce communication overheads in PGAS programs. Figure 1 shows an example where LICM hoists communication out of a loop. To enable this optimization, we use the *address space* feature of LLVM. In LLVM, each pointer type has an *address space*, which can differentiate between different kinds of memory in a system [22]. In this work, we use the *address space* feature to distinguish possibly-remote and definitely-local references (see Section 3.2). For example, a read from a remote node (Line 5 in Figure 1) can be expressed as a LLVM load instruction that involves a special address space:

```
%1 = load i64 addrspace(100)* %remoteData
```

In our convention, address space 100 indicates a possibly-remote address. Now that a remote reference can be represented as a standard load instruction, the existing LICM optimization in LLVM can move the remote read to the outside of loop (e.g. Line 3 in Figure 1) provided that the remote reference is a loop invariant. Similarly, other existing LLVM optimizations can remove redundant communications that are described as accesses with the address space 100. After these optimizations have completed, operations on pointers with address space 100 will be lowered to a runtime communication API. It is worth noting that we do not extend the LLVM IR. We just utilize the existing address space feature of LLVM as a PIR to enable communication optimizations for PGAS programs.

3.2 Code generation for PGAS LLVM

In order to enable existing LLVM optimizations to apply to communication in PGAS programs, a PGAS programming language front-end needs to generate LLVM IR with the address space feature. In particular, the PGAS LLVM IR needs to include *global pointers*, which are an alternative way of representing a pointer to possibly remote memory and are recognizable by LLVM as pointer types, instead of the typical structure representation of wide pointers that include at least a node number and a local address. The language front-end needs to use normal memory operations - such as load and store - for operations involving *global pointers* instead of calling runtime functions implementing GET or PUT. In other words, a GET/PUT can be modeled as a special kind of load/store operation.

In our current implementation, *global pointers* are pointers in address space 100. Address space 0 pointers continue to refer to local data. Encoding the possibly-remote nature of a pointer type in this manner allows existing LLVM optimizations to work with remote memory operations as if they were local. After the optimizations have completed, all global pointers will be converted to a standard wide pointer representation for use by the runtime system.

Operations on *global pointers* need to be generated as load or store instructions instead of calls to the runtime functions implementing GET and PUT. In addition to load and store with *global pointers*, our framework also handles memcpy and memset operations, where at least one of the pointer arguments is a *global pointer*. Thus, a GET can be generated as a load of a *global pointer* or a memcpy with a local destination and a *global pointer* source. Likewise, a PUT can be generated as a store to a *global pointer* or a memcpy with a *global pointer* destination and a local source.

It is also sometimes necessary to extract information from a global pointer. In order to support these operations, the LLVM code generated can include calls to the following placeholder functions which only exist during optimization:

- *addr* extracts the address from a *global pointer* argument
- *loc* extracts the *locale* from a *global pointer* argument
- *node* extracts the node number portion of the *locale* from a *global pointer* argument
- *make* constructs a *global pointer* from a locale and an address
- *global-to-wide* converts a *global pointer* to a wide pointer
- *wide-to-global* converts a wide pointer to a *global pointer*

These functions, along with the *global pointers* themselves and the operations on them, will be replaced with the usual runtime functions (e.g. PUT and GET) when all global pointers are lowered into wide pointers by the `global-to-wide` pass which is described below.

3.3 Existing LLVM optimization passes

Existing LLVM optimization passes are applied to the PGAS LLVM IR that now uses loads and stores on *global pointers* instead of GETs and PUTs. For example, the existing Loop Invariant Code Motion pass might hoist a GET - now represented as an invariant load from a *global pointer* - out of a loop (as discussed in Figure 1). In this way, existing LLVM optimization passes that eliminate redundant memory operations can remove communication in the PGAS program.

Note that if the *global pointer* technique was not applied, the existing LLVM optimization passes would not be able to optimize the communication in a PGAS program. In the Loop Invariant Code Motion example, the GET inside the loop would be a call to a runtime function that reads from the network. Without additional knowledge, existing optimization passes will treat runtime calls as an unknown function calls, and will not be able to perform any optimization on the runtime calls.

3.4 Lowering remote accesses (global-to-wide)

After running the LLVM-based optimizations, our optimization framework performs a `global-to-wide` pass to lower the *global pointers* to wide pointers and to translate operations on *global pointers* into operations on wide pointers supported by the language runtime.

The `global-to-wide` pass operates in two phases: an interprocedural phase and a function body transformation phase. An important assumption of this pass is that the *global pointers* marked with address space 100 have the same size as the wide pointers to which they will be translated, though their formats may be different.

In the interprocedural phase, this pass updates all function signatures and globals to translate from global types to wide types. Each function body is updated to call the placeholder function *wide-to-global* on its now-wide arguments to provide the value to replace the old argument. Similarly, each function body updated to call *global-to-wide* on any *global pointer* return value in order to convert it to a wide pointer type. In this phase, call sites are also updated by inserting calls to *global-to-wide* for all of the arguments by adding a call to *wide-to-global* for the result.

In the function body transformation phase, the pass replaces all operations on *global pointers* with the equivalent operations on the corresponding wide pointers. It replaces loads of a *global pointer* with a call to the runtime function to perform a GET; stores with calls to PUT; memcpys with calls to a runtime function to perform a GET, PUT, or GET and PUT. Pointer arithmetic operations, such as `getelementptr`, are replaced with the corresponding arithmetic on wide pointers. In addition, the placeholder functions described above for working with *global pointers*, including *addr*, *loc*, *node*, and *make* are replaced with their equivalent versions that generate wide pointers. Lastly, calls to the placeholder functions *global-to-wide* and *wide-to-global* are removed since all *global pointers* have been converted to wide pointers.

Note that in the current implementation, the `global-to-wide` pass requires that the wide pointer representation be 64 bits. In order to work with this limitation, we use a packed pointer format - where the first 16 bits are the node number and the remaining 48 bits form an address. This limitation is due to the fact that LLVM 3.3 did not support differently sized pointers for different address spaces - and address space 0 pointers are 64 bits on platforms of interest. We look forward to removing this limitation when using newer versions of LLVM.

3.5 Supporting various PGAS languages

This section discusses how our compilation framework enables the existing LLVM optimizations passes for various PGAS languages.

3.5.1 Generation of Global Pointers

In our framework, a load/store/memcpy/memset instruction that involves a specific address space is used to model an access through a pointer to possibly-remote data. Since our initial implementation works with Chapel, our *global pointers* only include a locale and an address; however, the LLVM global pointer optimizations do not need to see the internal implementation details of global pointers.

OpenSHMEM/UPC++ could also be easily optimized with our framework since its communication API calls take node ID and an address, as in Chapel's wide pointer. However, a shared pointer in the UPC language contains thread number, local address of block, and phase (position in the block)[3]. That might require the front-end to insert a sequence of statements that calculate node ID and an address within a node when processing a read/write statement from/to shared pointer. Alternatively, an implementation could adjust the PUT and GET calls to work with shared pointers containing phase - so that each PUT or GET might result in communication to more than one other node. In any case, there is a Clang-based UPC compiler under development which would integrate more easily with our framework. In CAF, a data access with a co-dimension indicates a possibly-remote access that could be optimized by our framework. In X10, the remote data transfers are performed by task level, (e.g. the *at* construct). The optimization can be done by applying LLVM pass to the special code blocks that correspond to those remote tasks.

3.5.2 Memory Consistency Model

Since existing LLVM optimizations will now be used to optimize communication, these optimizations will need to be subject to the PGAS language memory model. In this section, we briefly summarize the memory model assumptions made in this paper. We observe that LLVM provides flexibility for a front-end designer to support a wide range of memory models [22]. Therefore, we believe that our approach can support a specific language's memory consistency model by inserting appropriate fence (or equivalent) instructions when generating LLVM IR.

For Chapel in particular, the LLVM optimizations normally applied to C programs are safe to run because the memory consistency requirements for remote accesses are no more strict than those for local accesses. Since a commonly-used implementation of Chapel is via generation of C code, that implementation implicitly assumes that Chapel's memory model will be compatible with the C11 memory model.

Chapel's memory model has recently been formalized and is similar to C11 [9].

An interesting aspect of the UPC memory model is that it allows for some variables to have relaxed memory model semantics and others to have strict memory model semantics. UPC's relaxed memory model semantics are compatible with C11's memory model, whereas an implementation of UPC's strict memory model will need to add fence operations (or equivalent) for accesses to strict shared variables.

4. NEW LLVM-BASED COMMUNICATION OPTIMIZATIONS

In addition to the existing LLVM-based optimization passes, our framework allows a compiler designer to implement their own optimization passes built on top of our PGAS LLVM IR. A compiler designer may want to build an optimization pass which is specialized to runtime independent or runtime dependent features. This section introduces the three new LLVM-based communication optimizations introduced in this work: aggregating remote accesses, locality analysis and optimization, and coalescing of data transfers.

[3]UPC++'s global pointers no longer contains the block and phase.

```
 1 record Point { var x,y,z:int; }
 2 var A:[1..N] Point; // an array of Points
 3 on Locales[X] { // On another Locale
 4   for i in 1..N {
 5     sum += A[i].x + A[i].y + A[i].z;
 6   }
 7   for i in 1..N {
 8     part += A[i].x + A[i].z;
 9   }
10 }
```

Figure 3: Example Chapel program which benefits from the `aggregate-global-ops` pass.

4.1 Aggregating remote accesses

Our new `aggregate-global-ops` pass identifies operations on *global pointers* that can be combined into a single memcpy in order to aggregate PUTs and GETs. This pass works on the PGAS LLVM IR to find sequences of loads or sequences of stores on adjacent memory locations sharing a *global pointer* base address and turns them into a single memcpy. In this manner, loads from adjacent *global pointers* will generate a single GET and stores to adjacent *global pointers* will generate a single PUT. This optimization is important to our framework because existing LLVM optimizations can break apart a small memcpy data transfer into individual loads and stores, which can result in performance improvements for local accesses due to elimination of the memcpy() function call overheard. However, when the loads and stores access remote memory, it actually hurts performance to split a memcpy() into multiple loads and stores, since each load/store call will translate to a separate get/put communication call. A single aggregated get/put call would be much better instead. Figure 3 shows an example Chapel program working with records containing three fields - x, y, and z. It would not be unusual for the program to access all three of these fields in sequence as is shown on line 5. When that happens, the current Chapel front-end will generate three different GET requests - one for each field access. These GETs can be combined into a single request by the `aggregate-global-ops` pass.

The `aggregate-global-ops` pass is implemented in a similar manner to the existing memset optimization in LLVM. The memset optimization combines stores of a constant byte pattern to provably adjacent memory regions into memsets. Both of these optimizations look for sequences of memory operations that are working with addresses computed from the same base pointer and that have compile-time known offsets. The `aggregate-global-ops` optimization considers only memory operations through *global pointers* and considers both loads and stores. It replaces these operations with memcpy to or from a stack-allocated communication buffer and loads or stores on the communication buffer. One interesting feature of this optimization is that it can choose to aggregate non-contiguous GETs if the gap along them is small enough. This case can be seen in the example in Figure 3 on line 8. Fields x and z are requested but the optimization will request the intervening data (the y field) even though it will not be used. This ability helps in reducing the number of messages.

4.2 Locality analysis and optimization

The special address space (addrspace 100 in this paper) indicates a *possibly-remote* reference at compile-time, which means that the accessed data may be local or remote at runtime for dif-

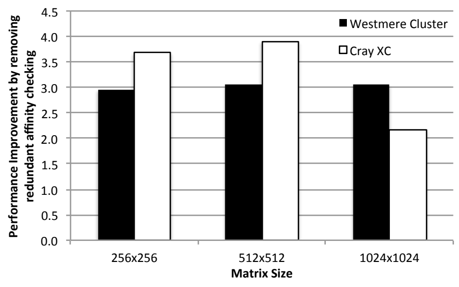

Figure 4: Performance improvements by removing runtime affinity checking by compiler on a single node of the Westmere cluster and a single node of the Cray XC.

```
var A[1..N]: int; // locale-local array
if (x == 0) {
  p = y;
} else {
  local { p = z; } // local stmt
}
z = A(0) + z; // definitely local
```

Figure 5: Example Chapel program which benefits from the `locality optimization` pass.

ferent instances of the same static reference. This variability is a consequence of the dynamic communication features of PGAS languages, which simplify the code needed to support data distributions, place-shifting, local/remote actual arguments for the same formal parameter, etc. To enable this flexibility, the runtime will check the affinity of data and perform either intra-node or inter-node communications. This check adds non-trivial overhead to an intra-node memory access. Figure 4 demonstrates the performance gain from eliminating runtime affinity checking on a single node of a Westmere cluster and a single node of a Cray XC30^{TM} system with matrix multiplication in Chapel (The detailed information on these platforms is shown in Section 5). The results show significant performance improvements (3.1x on average) by eliminating redundant runtime affinity checking on both platforms. This motivates us to infer an appropriate address space to avoid runtime affinity checking as much as possible at compile time. This technique is particularly effective for a language which permits implicit accesses to remote locations such as Chapel.

We introduce a novel locality analysis for identifying which *possibly-remote* accesses are guaranteed to be *definitely-local*. This analysis problem is challenging because it can require bidirectional dataflow analysis in general.

In Figure 5, consider an assignment statement `p = z`, where `p` and `z` are global pointers. If these variables are known to be definitely-local (e.g. by Chapel's *local* statements), the def/use of `p` and `z` in the other statements (Line 3 and 7) must be definitely-local. Similarly, the declaration of array `A` in Line 1 implies that `A` should be *definitely-local* since it is not distributed. Space limitations prevent us from including more details on the locality analysis and optimization. However, the impact of this optimization is included in the results presented in Section 5.

4.3 Runtime-Specific Data transfer Coalescing

Coalescing multiple remote data accesses into a single remote read/write operation has been widely used to optimizing distributed applications [4]. In our framework, we implement a data transfer coalescing that buffers remote read/write operations on top of PGAS LLVM IR. This pass is runtime specific since it uses the parallel programming language's runtime support for bulk data transfer. We use scalar expansion to replace the remote data access with local load/store operations on a memory buffer. This algorithm works on the call graph in a bottom-up manner. For each loop nest in each function, the optimization performs coalescing from inner most loop to outer most loop. The May-Happen-In-Parallel analysis [1] is applied to detected possible data races, and thereby ensures the correctness of the transformation.

5. PERFORMANCE EVALUATIONS

This section presents experimental results for our extended LLVM-based Chapel compiler on two platforms.

The first platform is an Intel Westmere cluster. This platform has multiple Intel Westmere nodes connected over a Quad Data Rate InfiniBand interconnect at 40 Gb/s. Each node has two 6-core Intel Xeon X5660 CPUs at 2.83GHz. There are 4GB of RAM per CPU core, with a total 48GB of RAM inside a single node.

The second platform is a Cray XC30^{TM} supercomputer. The platform has multiple Intel E5 nodes connected over the Cray Aries interconnect with Dragonfly topology with 23.7 TB/s global bandwidth. Each node has two 12-core Intel Xeon E5-2695 v2 CPUs at 2.40GHz and 64GB of RAM.

We use a modified version of Chapel compiler 1.9.0 with LLVM 3.3. In particular, we took the standard v 1.9.0 release of the Chapel compiler, and added the use of the LLVM-based locality optimization and coalescing passes reported in this paper. Note that our work on enabling existing LLVM optimizations and the aggregation pass is already implemented as the *–llvm-wide-opt* option in the standard Chapel compiler[4]. The *–fast* option is used for this evaluation. The Chapel runtime use the GASNet library 1.22.0 [7] for inter-node communication via Infiniband[5] and Cray Aries.

The six benchmarks shown in Table 1 are executed across multiple nodes. **Stream-EP**, **NPB EP**, and **SSCA2** were obtained from the Chapel repository [30]. **Smith-Waterman**, **Cholesky** and **Sobel** were ported to Chapel from their original UPC implementation. In these experiments, we use up to 32 nodes on the Westmere cluster and 64 nodes on the Cray XC. For Chapel, qthreads 1.10 [31] is used for enabling light-weight worker thread creation within a single node. 12 qthread workers and 48 qthread workers are running on the Westmere cluster and the Cray XC respectively.

We measured performance on multiple nodes using Chapel compiler's code generation in the following three LLVM-based modes:

- **Baseline** (**LLVM-unopt**) : uses the LLVM-backend with a packed wide pointer representation and without activating the LLVM communication optimizations. LLVM optimizations run but cannot optimize communication.
- **Global Pointer Optimization** (**LLVM-gopt**) : Code generation with *global pointers* + Existing LLVM passes + Aggregation pass + Lowering pass, which corresponds to *–llvm-wide-opt* in the standard Chapel compiler
- **All LLVM-based Communication Optimizations** (**LLVM-allopt**) : Global Pointer Optimization + Locality optimization + Coalescing.

[4]This paper is the first to publish the internal details of the *–llvm-wide-opt option*

[5]OpenMPI 1.6.3 is used for GASNet initialization

Benchmark	Lang	Summary	Data Size
Smith-Waterman	Chapel	Sequence alignment algorithm for DNA/RNA	N = 185,600×192,000, Tile Size = 2,900×3,000
Cholesky Decomp	Chapel	Decomposition of a Hermitian matrix	N = $10{,}000^2$ Tile Size = 500^2
NPB EP	Chapel	Generation of independent Gaussian random variates using the Marsaglia polar method from NAS Parallel benchmark, ported to Chapel	CLASS = D
Sobel	Chapel	Sobel edge-detection algorithm from UPC benchmark, ported to Chapel	N = $48{,}000^2$ for Westmere, $24{,}000^2$ for Cray XC, Tile Size = $3{,}000^2$ for Westmere, $1{,}200^2$ for Cray XC
SSCA2 Kernel 4	Chapel	Graph Theory Benchmark from [3], ported to Chapel	SCALE = 16, 4D Torus Graph, it is necessary to limit the number of tasks per locale to around 4 using the flag –dataParTasksPerLocale=4 to throttle the nested parallelism inherent in the algorithm.
Stream EP	Chapel	Simple vector kernel from HPC Challenge Benchmark	2^{30}

Table 1: Information on the benchmarks used to evaluate the Chapel compiler.

In the following sections, these three variants are referred to as **LLVM-unopt** , **LLVM-gopt** , and **LLVM-allopt** respectively.

The following four metrics were used to show the impact of our LLVM-based optimizations. Note that *Locales* is a Chapel's term which corresponds to 1 physical node.

- **Performance Improvement** : Figure 6 shows the speedup numbers of the six benchmarks on the Westmere cluster (up to 32 locales) and the Cray XC (up to 64 locales) relative to the **LLVM-unopt** version.
- **Breakdown for performance improvements** : Figure 6 also shows breakdown for average performance improvement by the existing LLVM passes, aggregation pass, locality optimization pass, and coalescing on the Cray XC[6].
- **Analysis of generated code by LLVM** : We analyze how our LLVM-based optimization passes transform each benchmark. (Figure 7)
- **Number of Chapel Communication API calls** : Table 2 shows the amount of Chapel communication API calls REMOVED by **LLVM-gopt** and **LLVM-allopt** relative to **LLVM-unopt** on the Cray XC[6].

5.1 Summary of Results

As shown in Figure 6, **LLVM-gopt** (existing LLVM passes + aggregation pass + lowering pass) and **LLVM-allopt** (**LLVM-gopt** + locality optimization + coalescing) are faster than **LLVM-unopt** (LLVM optimization run but cannot optimize communications). Specifically, **LLVM-gopt** and **LLVM-allopt** are 2.0× and 4.4× faster than **LLVM-unopt** on average on the Westmere cluster. For the Cray XC30 supercomputer, **LLVM-gopt** and **LLVM-allopt** are 1.9 × and 5.1× times faster than **LLVM-unopt** on average.

Smith-Waterman and **SSCA2 Kernel 4** do not show scalability on the Cray XC due to high-overheads in Chapel's communication runtime which is built on top of GASNet. An experiment with a synthetic benchmark indicated that Chapel programs have higher communication cost with our GASNet configuration on the Cray XC than on the Westmere cluster.

5.2 Breakdown for Performance Improvement

We first analyze the dynamic number of Chapel communication APIs made by each applications. As shown in Table 2, we observed

[6]We omit results on the Westmere cluster because the results are quantitively similar

Benchmark	Comm Kind	Cray XC	
		LLVM-gopt	**LLVM-allopt**
Smith-Waterman	LOCAL_GET	63.6%	75.5%
Note : obtained with	REMOTE_GET	36.4%	36.7%
18,560x19,200 input	LOCAL_PUT	58.0%	58.0%
	REMOTE_PUT	0.0%	0.0%
Cholesky	LOCAL_GET	77.6%	87.9%
Note : obtained with	REMOTE_GET	84.7%	99.8%
2,000x2,000 input	LOCAL_PUT	10.3%	10.8%
	REMOTE_PUT	0.0%	0.0%
NPB EP	LOCAL_GET	58.6%	58.6%
	REMOTE_GET	39.7%	39.7%
	LOCAL_PUT	29.5%	58.8%
	REMOTE_PUT	-	-
Sobel	LOCAL_GET	74.6%	95.2%
Note : obtained with	REMOTE_GET	0.0%	0.0%
CLASS=B	LOCAL_PUT	35.8%	68.3%
	REMOTE_PUT	-	-
SSCA2	LOCAL_GET	55.6%	56.2%
	REMOTE_GET	60.9%	60.8%
	LOCAL_PUT	5.6%	3.8%
	REMOTE_PUT	0.0%	0.0%
Stream-EP	LOCAL_GET	70.6%	70.6%
	REMOTE_GET	35.7%	35.7%
	LOCAL_PUT	17.3%	17.3%
	REMOTE_PUT	0.0%	0.0%

Table 2: The amount of Chapel Comm API calls REMOVED by LLVM-gopt and LLVM-allopt relative to LLVM-unopt (Cray XC, 16 locales).

decreases in the number of PUTs and GETs in addition to reductions in runtime. In particular, some amount of LOCAL_GETs and LOCAL_PUTs are removed by **LLVM-allopt** since it removes redundant affinity checking with locality optimization. (see Section 4.2)

In the following, we show how our framework optimizes each program by analyzing generated code by our LLVM-based optimizations. Detailed information on code transformation is presented in Figure 7.

5.2.1 Smith-Waterman

Since there are common array subscripts (e.g. A(ii,jj) and A(ii,jj-1)) in a main computation loop in **Smith-Waterman**[7], one of the existing LLVM passes replaces some of GETs with a variable hold-

[7]In **Smith-Waterman**, **LLVM-gopt** and **LLVM-allopt** on 2 locales on the Westmere cluster are in dead-lock. But this does not mean a problem in LLVM-backend because the conventional C-Backend also has this problem.

Platform	# of Locales	Backend	**Smith-Waterman**	**Cholesky**	**NPB EP**	**Sobel**	**SSCA2**	**Stream-EP**
Westmere	1	**LLVM-unopt**	756.2 sec	982.9 sec	3745.3 sec	7488.5 sec	2569.0 sec	6.6 sec
		LLVM-allopt	298.6 sec	292.9 sec	3005.1 sec	7231.0 sec	2449.2 sec	2.4 sec
		C-backend	206.6 sec	241.5 sec	2407.8 sec	6746.9 sec	2196.7 sec	2.56 sec
	32	**LLVM-unopt**	408.9 sec	408.9 sec	116.2 sec	232.1 sec	444.0 sec	0.26 sec
		LLVM-allopt	163.5 sec	79.3 sec	92.8 sec	222.2 sec	443.1 sec	0.106 sec
		C-backend	142.0 sec	730.9 sec	77.7 sec	206.9 sec	607.1 sec	0.107 sec
Cray XC	1	**LLVM-unopt**	480.11 sec	533.1 sec	920.5 sec	3927.6 sec	368.4 sec	1.8 sec
		LLVM-allopt	190.1 sec	201.7 sec	686.0 sec	3640.6 sec	326.1 sec	0.8 sec
		C-backend	115.6 sec	188.9 sec	548.3 sec	3796.4 sec	194.7 sec	0.8 sec
	32	**LLVM-unopt**	683.3 sec	1527.4 sec	30.1 sec	121.9 sec	2112.7 sec	0.09 sec
		LLVM-allopt	327.3 sec	119.8 sec	21.8 sec	93.2 sec	1943.0 sec	0.03 sec
		C-backend	359.0 sec	730.9 sec	18.1 sec	99.0 sec	2221.97 sec	0.03 sec
	64	**LLVM-unopt**	693.9 sec	1514.7 sec	15.5 sec	56.4 sec	2097.5 sec	0.04 sec
		LLVM-allopt	309.0 sec	118.0 sec	11.0 sec	50.9 sec	2032.9 sec	0.016 sec
		C-backend	347.7 sec	1356.8 sec	12.7 sec	40.7 sec	2234.9 sec	0.017 sec

Table 3: Absolute performance numbers for each benchmark.

ing the same value obtained by a predecessor GET by load elimination. To explain how it works, it is worth mentioning how the generated code accesses an element of Chapel array. The steps are 1) GET a pointer to head of array data, 2) GET per-dimension multiplier which is used for calculating an offset for multi-dimensional access, 3) calculate an address of the element with 1) and 2), and 4) GET the array element. In this case compiler reuses the per-dimension multiplier. The aggregation pass combines a sequence of GETs for Chapel array accesses into memcpys (see Section 4.1). Overall, the existing LLVM passes and the aggregation pass reduce 33 GETs in the main loop in Line 3 to 12 GETs (see Line 5), which contributes 181% (= 134% + 47%) of performance improvement relative to **LLVM-unopt** in average on the Cray XC. Additionally, the locality optimization pass detects locale-local array accesses and converts 4 GETs which are *possibly-remote* at compile-time but are *actually-local* at runtime into load access without communication calls in the main loop, which contributes additional 25% of performance improvement in average on the Cray XC. These optimizations actually eliminate significant number of GETs and PUTs at runtime. For example, **LLVM-gopt** and **LLVM-allopt** remove 63.6% and 75.5% of LOCAL_GET API calls respectively (see Table 2).

5.2.2 *Cholesky*

Figure 7 shows program transformation of update_nondiagonal() function by our optimizations, which is one of main computation parts in **Cholesky**. For example, LLVM loop invariant code motion pass moves 1 loop invariant GET of the 4 GETs in Line 12 to the outside of the second innermost loop. Similarly, the pass moves 2 GETs out of 9 GETs in Line 14 to the outside of the loop. Overall, the existing optimizations without the aggregation pass contributes 381% of performance improvement relative to **LLVM-unopt** in average on the Cray XC. The aggregation pass does not contribute performance improvement but Locality optimization contributes 4%, which is invisible in Figure 6 though). Coalescing shows significant performance improvement (1,565%).

5.2.3 *NPB EP*

There are two loops in the gaussPairsBatch() function, which is a main computation of **NBP EP**. **LLVM-gopt** applies LICM to 1 GET in the first loop in Line 31 and aggregates a sequence of GETs in the body of the second loop in Line 32. This contributes 132 (=127% + 5%) performance improvement in average on the the Cray XC. Locality optimization eliminates an affinity check for 1 PUT in the body of the first loop in Line 34, which eventually shows 6% performance improvement in average on the the Cray XC. We also observed decreases in the number of puts and gets in addition to reductions in runtime (See Table 2).

5.2.4 *Sobel*

Sobel shows similar trends to **Smith-Waterman**, since **Sobel** has a number of array accesses for neighbor elements like **Smith-waterman**, **LLVM-gopt** reduces 60 GETs to 15 GETs in the body of main computation loop in Line 41, which is 104% performance improvement in average on the Cray XC (the aggregation pass did not find opportunities for optimization). These 15 GETs are eventually reduced to 3 GETs by detecting a local array in locality optimization, leading to an additional 3% performance improvement in average on the Cray XC.

5.2.5 *SSCA2 Kernel 4*

The existing LICM optimization pass hoists a GET out of an important loop. Overall, **LLVM-gopt** without aggregation shows 117% performance improvement. Additionally, the other part in **SSCA2** benefits from the aggregation pass, which shows 11% performance improvement. Locality optimization attains additional 2% performance improvement in average on the Cray XC.

5.2.6 *Stream-EP*

LICM hoists 6 out of 8 GET calls out of the body of the main computation loop in Line 54. These hoisted GETs retrieve a pointer to array data and per-dimensional multiplier. This shows 234% performance improvement in average on the Cray XC but the aggregation pass does not help in this case. Additionally, Locality optimization shows additional 4% performance improvement.

5.3 Comparison with the conventional Chapel C-backend

Table 3 shows the execution time of the conventional C-backend and our LLVM-backend (**LLVM-unopt** and **LLVM-allopt**). Note that the numbers for LLVM-backend (**LLVM-unopt** and **LLVM-allopt**) corresponds to the results shown in Figure 6.

While **LLVM-allopt** always outperforms **LLVM-unopt** , the performance of the LLVM-backend is slower than that of the C-backend in some cases. One potential problem with the LLVM backend is the use of packed pointers. The C backend uses a structure representation of wide pointers, where each wide pointer is actually a structure containing a node ID and a local pointer. The LLVM version has to pack wide pointers into 64-bit quantities in order to work around the limitations in LLVM 3.3 that every address space have the same pointer size (For more details, see Section 3.4). Packing wide pointers into 64-bit values adds two primary sources of overhead. First, it uses more instructions such as bit shifting and masking in order to create and use wide pointers. Second, since the pointer values are now computed from integers, existing compiler optimization is less effective because alias analysis is much more difficult. Once this limitation is addressed and the LLVM backend uses a structure format for wide pointers, we expect that the LLVM backend will consistently produce faster distributed programs than the C backend because the C-backend cannot perform these communication optimizations. In particular, while the Chapel compiler does include communication optimization at the AST level, the low-level communication optimizations provided by our framework have no analogue when using the C backend. Finally, note that the number of communication API calls made in the generated code is similar for the C backend and for LLVM-unopt. Thus, the number of communication calls removed in Table 2 represent an improvement over using the C backend.

6. RELATED WORK

6.1 Communication optimizations for PGAS languages

Communication optimization is important technique for distributed memory applications. Compared with message-passing, the PGAS programming model provides a more user friendly interface, but relies on compiler optimizations for improved performance, especially for inter-node communications.

There is a lot of past work on optimizing communication in PGAS programs. X10 [10] is one of the HPCS languages with language-based notation for distributed arrays, global pointers, and locality exploitation. In [4], Barik et al. focus on reducing communication overheads across multiple nodes for distributed X10 programs. This work uses program transformation techniques to enable message aggregation, reuse and eliminate redundant communication. The relevant transformations includes scalar replacement, object splitting, and loop transformations, such as loop distribution, scalar expansion, loop tiling, and loop splitting. Chandra et.al. introduced a frontend approach [14] for enabling locality optimization. They extended X10 with s special dependent type system that provides *place types* which captures fine-grain locality information. In compiler, they provided a type inference algorithm for helping developer add type annotations.

UPC (Unified Parallel C) [13] is another mature PGAS system. In [5], Barton et al. introduced the compiler technique that performs affinity test of the `upc_forall` loop and eliminates accesses to shared pointers proven by the compiler analysis to be local. In [2], Alvanos et al. propose a communication optimization techniques that utilizes both the static coalescing optimization and the inspector-executor model.

The Fortran D compiler [17] introduces several program transformation and communication optimizations that reduces communication overhead, such as message vectorization, message pipelining for distributed systems. In [12], the High Performance Fortran Compiler merges communication events for different remote references into a single event for regular applications.

For OpenSHMEM optimization, the compiler [23] replaces function calls of remote memory access with load / store operations with `shmem_ptr` to enable vectorization in Xeon Phi processor.

The ZPL compiler [8] detects array constructs and performs several communication optimizations at the array-level.

All of above works are language-specific and compiler-specific optimizations, while in our proposed compiler optimization framework, we demonstrate that using LLVM as a general platform and plug-in program optimizations for distributed memory application (i.e. PGAS program) is a flexible approach and easy to be adapted to existing/new program languages.

6.2 LLVM Utilization for High-Performance Computing

LLVM is widely used as compiler infrastructure in both academia and industry. There have been several projects that explore the use of LLVM as a program optimization tool for high performance computing. For example, Intel SPMD Program Compiler [18] aims to generate vectorized code for vector units. LLVM CUDA compiler [25] compiles CUDA C/C++, Fortran, and Domain Specific Languages to NVVM IR [26], which is an extended version of LLVM IR for GPGPU. This compiler then applies the LLVM-based optimizations and generates a GPU binary. It is worth mentioning that this compiler can distinguish several types of memory such as shared memory, texture memory, and constant memory with the address space feature in LLVM. One of prior approaches makes use of NVVM IR to generate GPU code from Java 8 programs [19].

In contrast, our work is working toward building a general program optimization framework for distributed memory applications, which can support multiple programming languages, especially for PGAS programming system for both runtime independent/dependent program optimizations, especially for optimizing communication overhead.

7. CONCLUSION

In this paper, we proposed an LLVM-based optimization framework for PGAS programs. Our compilation system uses the address space feature in LLVM to differentiate between definitely-local and possibly-remote memory accesses, and leverages the existing optimization passes to do further elimination of communication overheads. Our experimental results show an average performance improvement of 3.5× and 3.4× on 64 nodes of a Cray XC30 supercomputer and 32 nodes of a Westmere cluster, respectively. These experiments shows that our approach effectively accelerates the execution of PGAS programs.

This work shows that the use of LLVM is a promising way to enhance the performance of PGAS programs. In the future, we plan to extend our compilation system to support a wide range of PGAS language features such as task parallel constructs and synchronizations by introducing uniform parallel intermediate representations (PIRs) [32].

Acknowledgment

We would like to thank members of the Chapel team at Cray and the Habanero group at Rice University for valuable discussions related to this work. The Westmere results in this paper were obtained on the Rice DAVinCI system, which was partially supported in part by the Data Analysis and Visualization Cyberinfrastructure funded by NSF under grant OCI-0959097 and Rice University. In addition, the Cray XC results were obtained on the Edison supercomputer at the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

8. REFERENCES

[1] S. Agarwal, R. Barik, V. Sarkar, and R. K. Shyamasundar. May-happen-in-parallel analysis of X10 programs. PPoPP '07, pages 183–193, New York, NY, USA, 2007. ACM.

[2] M. Alvanos, M. Farreras, E. Tiotto, J. N. Amaral, and X. Martorell. Improving communication in PGAS environments: Static and dynamic coalescing in UPC. ICS '13, pages 129–138, New York, NY, USA, 2013. ACM.

[3] D. A. Bader and K. Madduri. Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors. HiPC'05, pages 465–476, Berlin, Heidelberg, 2005. Springer-Verlag.

[4] R. Barik, J. Zhao, D. Grove, I. Peshansky, Z. Budimlic, and V. Sarkar. Communication optimizations for distributed-memory X10 programs. In *IPDPS'11*, pages 1101–1113, 2011.

[5] C. Barton, C. Casçaval, G. Almási, Y. Zheng, M. Farreras, S. Chatterje, and J. N. Amaral. Shared memory programming for large scale machines. PLDI'06, pages 108–117, New York, NY, USA, 2006. ACM.

[6] C. M. Barton. Improving Access to Shared Data in a Partitioned Global Address Space Programming Model. 2009. Ph.D. Thesis.

[7] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick. Optimizing bandwidth limited problems using one-sided communication and overlap. In *IPDPS'06*, pages 10 pp.–, April 2006.

[8] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder, and D. Weathersby. Factor-join: A unique approach to compiling array languages for parallel machines. LCPC '96, pages 481–500, London, UK, UK, 1997. Springer-Verlag.

[9] Chapel. The Chapel language specification version 0.98. http://chapel.cray.com/spec/spec-0.98.pdf, Oct. 2015.

[10] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. OOPSLA'05, pages 519–538, New York, NY, USA, 2005. ACM.

[11] S. Chatterjee, S. Tasirlar, Z. Budimlic, V. Cave, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan. Integrating asynchronous task parallelism with MPI. IPDPS '13, pages 712–725, Washington, DC, USA, 2013. IEEE Computer Society.

[12] D. Chavarría-Miranda and J. Mellor-Crummey. Effective communication coalescing for data-parallel applications. PPoPP '05, pages 14–25, New York, NY, USA, 2005. ACM.

[13] T. El-Ghazawi, W. W. Carlson, and J. M. Draper. UPC Language Specification v1.1.1, October 2003.

[14] S. C. et al. Type inference for locality analysis of distributed data structures. In *PPoPP '08*, pages 11–22, New York, NY, USA, 2008. ACM.

[15] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA, 1994.

[16] P. Hilfinger, D. Bonachea, D. Gay, S. Graham, B. Liblit, G. Pike, and K. Yelick. Titanium Language Reference Manual. Technical Report CSD-01-1163, University of California at Berkeley, Berkeley, Ca, USA, 2001.

[17] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Evaluation of compiler optimizations for Fortran D on MIMD distributed memory machines. ICS '92, pages 1–14, New York, NY, USA, 1992. ACM.

[18] Intel. Intel SPMD program compiler. https://ispc.github.io/.

[19] K. Ishizaki, A. Hayashi, G. Koblents, and V. Sarkar. Compiling and Optimizing Java 8 Programs for GPGPU Execution. PACT '15, 2015.

[20] S. Kumar, A. Mamidala, D. Faraj, B. Smith, M. Blocksome, B. Cernohous, D. Miller, J. Parker, J. Ratterman, P. Heidelberger, D. Chen, and B. Steinmacher-Burrow. Pami: A parallel active message interface for the Blue Gene/Q supercomputer. In *IPDPS '12*, pages 763–773, May 2012.

[21] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[22] LLVM. LLVM language reference manual. http://llvm.org/docs/LangRef.html.

[23] N. Namashivayam, S. Ghosh, D. Khaldi, D. Eachempati, and B. Chapman. Native mode-based optimizations of remote memory accesses in OpenSHMEM for intel xeon phi. PGAS '14, pages 12:1–12:11, New York, NY, USA, 2014. ACM.

[24] R. W. Numrich and J. Reid. Co-Array Fortran for parallel programming. *ACM SIGPLAN Fortran Forum Archive*, 17:1–31, Aug. 1998.

[25] NVIDIA. CUDA LLVM compiler. https://developer.nvidia.com/cuda-llvm-compiler.

[26] NVIDIA. NVVM IR specification 1.1. http://docs.nvidia.com/cuda/nvvm-ir-spec/index.html.

[27] S. Poole, O. Hernandez, J. Kuehn, G. Shipman, A. Curtis, and K. Feind. Openshmem - toward a unified RMA model. In D. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1379–1391. Springer US, 2011.

[28] V. Sarkar, W. Harrod, and A. Snavely. Software Challenges in Extreme Scale Systems. *Special Issue on Advanced Computing: The Roadmap to Exascale.*, 2010.

[29] A. Skjellum, E. Lusk, and W. Gropp. *Using MPI: Portable Parallel Programming with the Message Passing Iinterface*. MIT Press, 1999.

[30] sourceforge.net. Chapel repository. http://sourceforge.net/p/chapel/code/HEAD/tree/trunk/test/.

[31] K. Wheeler, R. Murphy, and D. Thain. Qthreads: An API for programming with millions of lightweight threads. In *IPDPS '08*, pages 1–8, April 2008.

[32] J. Zhao and V. Sarkar. Intermediate Language Extensions for Parallelism. *VMIL'11*, October 2011.

[33] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick. Upc++: A PGAS extension for C++. IPDPS '14, pages 1105–1114, Washington, DC, USA, 2014. IEEE Computer Society.

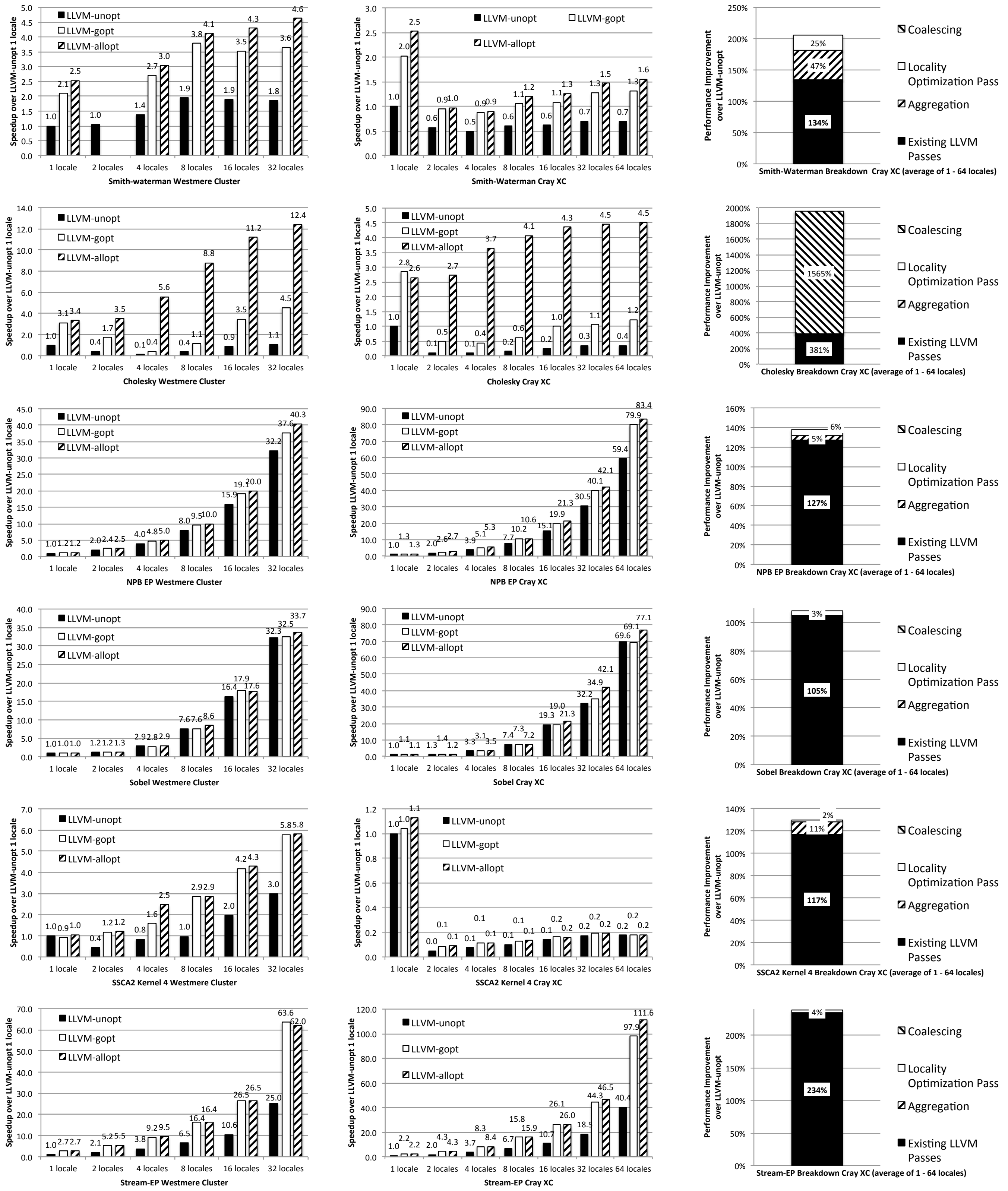

Figure 6: Comparison of performance improvement relative to LLVM-unoptimized on 1 locale on the Westmere Cluster and the Cray XC supercomputer and Breakdown for performance improvement. LLVM-gopt : existing optimizations + aggregation (*–llvm-wide-opt* in the standard Chapel compiler), LLVM-allopt : LLVM-gopt + locality optimization + coalescing

```
 Smith-Waterman
// LLVM-unopt
for (ii, jj) in tile {  33 GETs + 1 PUT    }
// LLVM-gopt
for (ii, jj) in tile {  12 GETs + 1 PUT    }
//  LLVM-allopt
for (ii, jj) in tile {  8 GETs + 1 PUT    }
 Cholesky
// LLVM-unopt
for jB in zero..tileSize-1 do {
  for kB in zero..tileSize-1 do {
    4 GETs
    for iB in zero..tileSize-1 do {
      9 GETs + 1 PUT   }}}
// LLVM-gopt
for jB in zero..tileSize-1 do {
  1 GETs
  for kB in zero..tileSize-1 do {
    3 GETs
    for iB in zero..tileSize-1 do {
      2 GETs + 1 PUT   }}}
// LLVM-allopt
  bulk_transfer();
for jB in zero..tileSize-1 do {
  for kB in zero..tileSize-1 do {
    1 GET
    for iB in zero..tileSize-1 do {
      1 GET + 1 PUT   }}}
 NPB EP
// LLVM-unopt
for i in pairs {  2 GETs + 2 PUTs    }
for i in pairs {  8 GETs + 1 PUT    }
// LLVM-gopt
for i in pairs {  1 GET + 2 PUTs    }
for i in pairs {  3 GETs + 1 PUT    }
// LLVM-allopt
for i in pairs {  1 GETs + 1 PUT    }
for i in pairs {  3 GETs + 1 PUT    }
 Sobel
// LLVM-unopt
for (ii, jj) in tile {  60 GETs + 2 PUTs    }
// LLVM-gopt
for (ii, jj) in tile {  15 GETs + 2 PUTs    }
// LLVM-allopt
for (ii, jj) in tile {  3 GETs + 2 PUTs    }
 SSCA2 Kernel 4
// LLVM-unopt
forall v in G.FilteredNeighbors(...) {  3 GETs    }
// LLVM-gopt, LLVM-allopt
 1 GET
forall v in G.FilteredNeighbors(...) {  2 GETs    }
 Stream-EP
// LLVM-unopt
forall (a, b, c) in zip(A, B, C) {  8 GETs + 1PUT    }
// LLVM-gopt, LLVM-allopt
 6 GETs
forall (a, b, c) in zip(A, B, C) {  2GETs + 1PUT    }
```

Figure 7: How optimizations work for six benchmarks.