GPUIterator: Bridging the Gap between Chapel and GPU Platforms

Akihiro Hayashi  
Department of Computer Science  
Rice University  
USA  
ahayashi@rice.edu

Sri Raj Paul  
College of Computing  
Georgia Institute of Technology  
USA  
sriraj@gatech.edu

Vivek Sarkar  
College of Computing  
Georgia Institute of Technology  
USA  
vsarkar@gatech.edu

Abstract
PGAS (Partitioned Global Address Space) programming models were originally designed to facilitate productive parallel programming at both the intra-node and inter-node levels in homogeneous parallel machines. However, there is a growing need to support accelerators, especially GPU accelerators, in heterogeneous nodes in a cluster. Among high-level PGAS programming languages, Chapel is well suited for this task due to its use of locales and domains to help abstract away low-level details of data and compute mappings for different compute nodes, as well as for different processing units (CPU vs. GPU) within a node.

In this paper, we address some of the key limitations of past approaches on mapping Chapel on to GPUs as follows. First, we introduce a Chapel module, GPUIterator, which is a portable programming interface that supports GPU execution of a Chapel forall loop. This module makes it possible for Chapel programmers to easily use hand-tuned native GPU programs/libraries, which is an important requirement in practice since there is still a big performance gap between compiler-generated GPU code and hand-tuned GPU code; hand-optimization of CPU-GPU data transfers is also an important contributor to this performance gap. Second, though Chapel programs are regularly executed on multi-node clusters, past work on GPU enablement of Chapel programs mainly focused on single-node execution. In contrast, our work supports execution across multiple CPU+GPU nodes by accepting Chapel’s distributed domains. Third, our approach supports hybrid execution of a Chapel parallel (forall) loop across both a GPU and CPU cores, which is beneficial for specific platforms.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CHIUW ’19, June 22, 2019, Phoenix, AZ, USA  
© 2019 Association for Computing Machinery.  
ACM ISBN 978-1-4503-6800-1/19/06 ... $15.00  
https://doi.org/10.1145/3329722.3330142

1 Introduction
Software productivity and portability is a profound issue for large scale systems. While conventional message-passing programming models such as MPI [11] are widely used in distributed-memory programs, orchestrating their low-level APIs imposes significant burdens on programmers. One promising solution is the use of PGAS (Partitioned Global Space) programming languages such as Chapel, Co-array Fortran, Habanero-C, Unified Parallel C (UPC), UPC++, and X10 [2, 8, 9, 12, 13, 16] since they are designed to mitigate productivity burdens by introducing high-level parallel language constructs that support globally accessible data, data parallelism, task parallelism, synchronization, and mutual exclusion.

However, there is a growing need to support accelerators, especially GPU accelerators, in heterogeneous nodes since they are now a common source of performance improvement in HPC clusters. According to the Top500 lists [17], 138 of the top 500 systems currently include accelerators. Thus, to keep up with the enhancements of hardware resources, a key challenge in the future development of PGAS programming models is to improve the programmability of accelerators.

For enabling GPU programming in PGAS programming models, past approaches focus on compiling and optimizing high-level data-parallel constructs for GPU execution. For example, Sidelnik et al. [3] and Chu et al. [7] compile Chapel’s
Listing 1. Problem: The user has to switch back and forth between CPU (forall) and GPU versions (an external C function call myGPUCode()) when exploring higher performance.

```csharp
forall i in 1..n {...}
// The user has to manually switch between CPU and GPU versions
myGPUCode(...);
```

Listing 2. Our Proposal: GPUIterator provides an appropriate interface between Chapel and accelerator programs.

```csharp
var GPUWrapper = lambda (lo: int, hi: int, n: int) {
  // The GPU portion (lo and hi) is automatically computed
  myGPUCode(lo, hi, n, ...);
  // even in multi-locale settings.
  GPUWrapper(lo, hi, n, ...);
}
forall i in 1..n {...}
// GPU version (invoking an external C function)
myGPUCode(...);
```

for all construct, to GPUs. Similarly, X10CUDA [10] compiles X10’s forall constructor to GPUs. While such compiler-driven approaches significantly increase productivity and portability, they often fall short in delivering the best possible performance on GPUs. Thus, it is important to provide an additional mechanism through which programmers can utilize low-level GPU kernels written in CUDA/OpenCL and highly tuned libraries like cuBLAS to achieve the highest possible performance.

Interestingly, Chapel inherently addresses this "performance" vs. "portability" issue through an approach that supports separation of concerns [1]. More specifically, Chapel’s multi-resolution concept allows programmers not only to stick with a high-level specification but also to dive into low-level details so that they can incrementally evolve their implementations with small changes. For example, [1] discusses multi-resolution support for arrays.

As for GPU programming with Chapel, typically programmers first start with writing forall loops and run these loops on CPUs as a proof-of-concept (see notional CPU version in Listing 1). If the resulting CPU performance is not sufficient for their needs, their next step could be to try the automatic compiler-based GPU code generation techniques discussed earlier. For portions that remain as performance bottlenecks, even after automatic compilation approaches, the next step is to consider writing GPU kernels using CUDA/OpenCL and invoking these kernels from the Chapel program using Chapel’s C interoperability (GPU version in Listing 1). More details of the C interoperability feature is discussed in Section 2.2.

However, we believe the current GPU development flow for three reasons. First, when using one version (either the CPU or GPU version), another version should be somehow removed or commented out, which is not very portable. Second, it is non-trivial or tedious to run the GPU version on multiple CPU+GPU nodes. Third, a choice of device is either the CPU or GPU version, while hybrid execution of these two versions can be more flexible and deliver higher performance. It is worth noting that the second and third points were not addressed in the past work on mapping Chapel on to GPUs [3, 7].

A primary goal of this paper is to provide an appropriate interface between Chapel and accelerator programs such that expert accelerator programmers can explore different variants in a portable way (i.e., CPU-only, GPU-only, X% for CPU and Y% for GPU on a single or multiple CPU+GPU node(s)). To address these challenges, we introduce a Chapel module, GPUIterator, which provides the capability of creating and distributing tasks across a single or multiple CPU+GPU node(s). As shown in Listing 2 our approach enables running the forall loop on CPU+GPU with minimal changes - i.e., wrapping the original loop range in GPU() (Line 9) with some extra code including a callback function (GPUWrapper(), Line 1-5) that eventually calls the GPU function with an automatically computed subrange (lo, and hi) for the GPU.

This paper makes the following contributions by addressing some of the key limitations of past work:

- Design and implementation of the GPUIterator module for Chapel, which
  1. provides a portable programming interface that supports CPU+GPU execution of a Chapel forall loop without requiring any modifications to the Chapel compiler.
  2. supports execution across multiple CPU+GPU nodes by accepting Chapel’s distributed domains to support multi-node GPUs.
  3. supports hybrid execution of a forall across both CPU and GPU processors.
- Performance evaluations of different CPU+GPU execution strategies for Chapel on three CPU+GPU platforms.

2 Chapel

The Chapel [2] language is a productive parallel programming language developed by Cray Inc. Chapel was initially developed as part of the DARPA High Productivity Computing Systems program (HPCS) to create highly productive languages for next-generation supercomputers. Chapel provides high-level abstractions to express multithreaded execution via data parallelism, task parallelism, and concurrency.

2.1 Iterators

An iterator [6] is a high-level abstraction that gives programmers control over the scheduling of the loops in a very
work stealing, we can create a new iterator using work_steal strategies can be found in [5].

Another example and use it just by changing the loop header as shown in the

Listing 3. Using an iterator to transform a program to express parallelism and vary scheduling

1 // Serial version
2 for i in 1..n { body(i) }
3 4 // Parallel version
5 forall i in 1..n { body(i) }
6 7 // Parallel work-stealing version
8 forall i in work_steal(1..n) { body(i) }
9 iter work_steal(r: range(?)) {
10 // work stealing implementation
11 }

productive manner. Iterators help to isolate iteration away from the computation. Thus loop iteration and loop body can be specified orthogonal of each other so as to minimize the impact of changes to one on the other. In general, many scientific applications have many complex loop nests which are optimized for maximizing performance. This large number of loop nests creates maintenance problems because each loop nest needs to be maintained separately as the system evolves through changes in parallelism, memory hierarchy and so on. The iterator can help to reduce this maintenance problem where a change in an iterator’s definition can affect all the loops using it rather than rewriting each of the loop nest individually.

An iterator is like a normal procedure which can yield multiple times as it executes whereas a procedure can return only once. They can be used to drive loops because each yielded value can correspond to a single loop iteration. Iterators can be used to represent complex iteration space, for example, perform yield on the vertices of a graph with some property.

Parallel Iterators: In Chapel, iterators can also help to perform parallel iteration. They can be used with both data-parallel (forall) and task-parallel (coforall) constructs. Listing 3 gives an example of a serial loop with an iterator and its parallel equivalents. In the example, while iterating over an integer range, the parallel version will evenly distribute the iterations on to available hardware parallelism, i.e., if there are \( p \) cores\(^1\), then each core will get a chunk with \( n/p \) iterations. In case we want to try a different scheduling strategy, say for example we want to distribute work using work-stealing, we can create a new iterator work_steal\(^2\), and use it just by changing the loop header as shown in the parallel work-stealing version in Listing 3. Another example usage of parallel iterators that simplifies complex scheduling strategies can be found in [5].

\(^1\)Assuming one thread per core.
\(^2\)Chapel provides work-stealing strategy using adaptive iterator. We are using work_steal iterator for demonstration purpose.

Listing 4. C interoperability in Chapel

1 // Caller.c (C file)
2 double func(int x) {...}

1 // Caller.chpl (Chapel file)
2 extern func(x: int) real;
3 b = func(a);

2.2 C Interoperability

Chapel allows programmers to refer external C functions, variables, and types thereby enabling Chapel to interoperate with C. Listing 4 shows a code example, where a C function (Line 2 in Caller.c) is invoked from a Chapel program (Caller.chpl). Note that an explicit extern declaration of the C function (Line 2 in Caller.chpl) is required so that it can be callable from the Chapel part. In this work, we use this feature for invoking hand-coded CUDA/OpenCL programs from the Chapel side.

3 Design

Overall, the basic strategy for enhancing GPU interfaces for PGAS programming is to introduce a new parallel iterator module called GPUIterator. The GPUIterator is supposed to be invoked in a forall loop (e.g., forall i in GPU(...)) to create and distribute tasks across a single or multiple CPU+GPU node(s), which gives more flexibility to explore different implementations. Specifically, the user can easily enable/disable this feature by just wrapping/unwrapping the original loop range/domain in GPU(). In the followings, we first discuss the problem of the conventional GPU programming in Chapel to motivate the importance of GPUIterator and then discuss the detailed design of it.

3.1 Motivation for Introducing GPUIterator

As discussed in Section 1, as a multi-resolution language, Chapel allows expert GPU programmers to manually develop manually prepared GPU programs that can be callable from a Chapel program. This can be done by invoking CUDA/OpenCL programs using the C interoperability feature discussed in Section 2.2. To understand this, consider the baseline forall implementation that performs vector copy shown in Listing 5. The equivalent Chapel+GPU code is shown in Listing 6. It is worth noting that Chapel enables seamless and intuitive data exchanges between the Chapel side and the native side due to the feature. The key difference is that the original forall loop (Line 4-6 in Listing 5) is replaced with the function call (Line 6 in Listing 6) to the native function that should include typical host and device operations including device memory allocations, data transfers, and kernel invocations (the C file in Listing 6).

Unfortunately, the source code is not very portable particularly when the user wants to explore different variants to get higher performance. Since GPUs are not always faster than...
Default Parallel iterator

1 forall i in 1..n {
2 // CPU Code
3 }

GPU Parallel iterator

1 use GPUIterator;
2 forall i in GPU(1..n, GPUWrapper, CPUPercent) {
3 // CPU Code
4 }

Listing 5. A baseline forall implementation

// Chapel file
var A: [1..n] real(32);
forall i in 1..n {
A(i) = B(i);
}

Listing 6. A Chapel+GPU program equivalent to Listing 5

// Chapel file
extern proc GPUfunc(A: [] real(32), B: [] real(32),
    lo: int, hi: int, N: int);

var A: [1..n] real(32);
var B: [1..n] real(32);
GPUfunc(A, B, 1, n);

// Separate C file
void GPUfunc(float *A, float *B, int start, int end) {
// GPU Implementation (CUDA/OpenCL)
// Note: A[0] and B[0] here corresponds to
// A(1) and B(1) in the Chapel part respectively
}

3.2.1 Single Locale Execution (ranges)

Figure 1 shows an overview of the GPUIterator. Let us first describe the behavior of Chapel’s default parallel iterator over a range of 1..n. When the range is invoked in a forall loop, the parallel iterator module divides the range (1 to n) into m chunks, where m is the number of CPUs, so the chunks can be executed by m CPUs in parallel.

Our GPUIterator module is essentially an extended version of the default iterator that is aware of GPUs. In summary, the module first divides the iteration space into two portions by looking at CPUPercent specified by the user, one is for CPUs, and another is for GPUs. Then, the module further divides the CPU portion to create m chunks and the GPU portion to create k chunks. More implementation details can be found in Section 4.

Also, Listing 7 illustrates a code example of the GPUIterator with Chapel’s range. To use it in the Chapel file, the user 1) imports the GPUIterator module (Line 2), 2) creates a wrapper function (Line 11-13) which is a callback function invoked after the module has created a task for the GPU portion of the iteration space (Line 11) and eventually invokes the GPU function (Line 12), 3) then wraps the iteration space using GPU() (Line 15) with the wrapper function GPUIterator. Note that the last argument (CPUPercent), the percentage of the iteration space will be executed on the CPU, is optional. The default number for it is zero, meaning the whole iteration space goes to the GPU side.

Figure 1. Overview of GPUIterator (Single Locale)
Listing 7. An example \textsc{GPUIterator} program (single locale)

\begin{verbatim}
1 // Chapel file
2 use GPU\texttt{ITERATOR};
3
4 extern proc GPUfunc(A: [] real(32), B: [] real(32),
5   lo:int, hi: int, N: int);

6 var A: [1..n] real(32);
7 var B: [1..n] real(32);

8 // Users need to prepare a callback function which is
9 // invoked after the \textsc{GPUIterator} has computed the GPU portion
10 var GPUWrapper = lambda (lo:int, hi: int, n: int) {
11   GPUfunc(A, B, lo, hi, n);
12   }
13
14 var GPU\texttt{Percent} = 50; // GPU\texttt{Percent} is optional
15 forall i in GPU[1..n, GPU\texttt{Wrapper}, GPU\texttt{Percent}) {
16   // CPU code
17   A(i) = B(i);
18 }
\end{verbatim}

// Separate C file

\begin{verbatim}
19 void GPUfunc(float *A, float *B, int start, int end, int n) {
20   // GPU Implementation (CUDA/OpenCL)
21   // Note: A[0] and B[0] here corresponds to
22   // (A[0] and B[0]) in the Chapel part respectively
23 }
\end{verbatim}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure2.png}
\caption{GPU locale model (CHPL\_LOCALE\_MODEL=gpu)}
\end{figure}

3.2.2 Multiple Locales Execution (\texttt{dmapped} domains)

The \textsc{GPUIterator} also accepts distributed domains to enable
distributed hybrid execution of CPUs and GPUs. It first divides
the domain 1..n into \texttt{l} chunks, where \texttt{l} is the number
of locales. Then, as with the single locale case, it further divides
each chunk into the CPU and GPU portions.

Listing 8 shows a code example of the \textsc{GPUIterator} with
distributed domains. Compared to Listing 7, there are two key differences. First, like typical distributed parallel for all
implementations, a distributed domain needs to be provided
instead of a range (Line 6 and Line 16), which preserves
portability very nicely. Second, the way of passing distributed
arrays to the native function is different from that of passing
local arrays. That is, in the callback function (Line 12), local sub-
arrays of the distributed array \( A \).\_\texttt{local}Slice(lo..hi) and
\( B \).\_\texttt{local}Slice(lo..hi)\(^3\) are passed. Figure 3 illustrates
how the local slice of the distributed array \( A \).\_\texttt{local}Slice(lo
..hi) corresponds to the native array \( \text{float}* A \).

For example, suppose \( n = 1024 \), \( l = 2 \) (two locales),
\texttt{CPU\texttt{Percent}} = 0. On Locale 0, \( 0 \) and \( \text{hi} \) computed by
the \textsc{GPUIterator} are 1 and 512 respectively. Similarly, on Locale
1, \( 1 \) and \( \text{hi} \) are 513, and 1024 respectively. Note that the
\texttt{\_\texttt{local}Slice()} is a Chapel Arrays API function.

\begin{verbatim}
Listing 8. An example \textsc{GPUIterator} program (multiple locales)

1 // Chapel file
2 use \textsc{GPUIterator};
3
4 extern proc GPUfunc(A: [] real(32), B: [] real(32),
5   lo:int, hi: int, N: int);

6 var D: domain(1) \texttt{dmapped} block(boundingBox={1..n}) = {1..n};
7 var A: [D] real(32);
8 var B: [D] real(32);
9
10 var GPU\texttt{Percent} = 50; // GPU\texttt{Percent} is optional
11 forall i in GPU(D, GPU\texttt{Wrapper}, GPU\texttt{Percent}) {
12   // CPU code
13   A(i) = B(i);
14 }
\end{verbatim}

\begin{verbatim}
19 void GPUfunc(float *A, float *B, int start, int end) {
20   // GPU Implementation (CUDA/OpenCL)
21   // Example Chapel-C array mapping with distributed setting
22   // Suppose \( n = 1024 \), the number of locales is 2
23   // and GPU\texttt{Percent} = 0,
24   // On Locale 0:
25   // \( A[0] \) and \( B[0] \) here are \( A(1) \) and \( B(1) \) in the Chapel part
26   // On Locale 1:
27   // \( A[0] \) and \( B[0] \) here are \( A(513) \) and \( B(513) \) in the Chapel part
\end{verbatim}

3.2.3 Supporting Zipper \texttt{forall}

The \textsc{GPUIterator} is also designed to support zippered for all
loops \([4]\) in addition to standalone (non-zippered) for all
loops. A code example is shown in Listing 9. It is worth noting
that the \textsc{GPUIterator} must be the first argument of zip
so it can be the leader iterator. Otherwise, GPUs are not used
because follower iterators are not supposed to create any
tasks and just follows what the leader iterator generates.

\begin{verbatim}
Listing 9. An example \textsc{GPUIterator} program (zippered)

1 forall (i, a, b) in zip(GPU(D, GPU\texttt{Wrapper}, GPU\texttt{Percent}), A, B) {
2   // CPU code
3   a = b;
4 }
\end{verbatim}
We implemented the GPUIterator. The actual implementation can be found at [15].

4.1 GPU Locale Model

In Chapel, a locale is a high-level abstraction of a physical node and may contain architectural descriptions (e.g., the number of processor cores, possibly the number of accelerators, and so on) that are visible from Chapel programs. Also, it may be hierarchical (sublocales). A target system can be virtually regarded as a network of locales, which is called locale models. Because there are currently only two homogeneous locale models (flat and NUMA), we created a new locale model that is aware of accelerators (CHPL_LOCALE_MODEL=gpu).

The GPU locale model consists of two sublocales. One is for a set of CPUs, and another is for a set of GPUs (Figure 2). This architectural information is used for creating and distributing tasks onto CPUs and GPUs.

4.2 Detailed GPUIterator Implementation

4.2.1 Single Locale Execution (ranges)

We begin with our standalone parallel iterator implementation that is used to implement non-zippered forall loops (i.e., forall i in GPU(...)). Listing 10 illustrates the implementation of the standalone GPUIterator. It first computes ranges for the CPU and GPU portions (Line 7-14) and then creates two tasks (Line 16), one is for sublocale 0 (CPUs), another is for sublocale 1 (GPUs). For sublocale 0, it further creates numTasks tasks to generate parallelism (Line 21) on the CPU side, where numTasks is the maximum task concurrency on this locale stored in the GPU local model. For sublocale 1, it essentially invokes a callback function given by the user with the computed GPU range. Note that the range is translated by -r.low to make it 0-origin. Also, the current

<table>
<thead>
<tr>
<th>Locale 0</th>
<th>Locale 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Portion</td>
<td>GPU Portion</td>
</tr>
<tr>
<td>A (Chapel)</td>
<td>A.localeSlice(lo..hi)</td>
</tr>
<tr>
<td></td>
<td>A[0] to A[hi-lo]</td>
</tr>
<tr>
<td>A (Native)</td>
<td>A.localeSlice(lo..hi)</td>
</tr>
<tr>
<td></td>
<td>A[0] to A[hi-lo]</td>
</tr>
</tbody>
</table>

Figure 3. Overview of GPUIterator (Multiple Locales)

Listing 10. GPUIterator implementation (single locale)

```
1  iter GPU(param tag: iterKind, 
2      r: range(?), 
3      CPUPercent: int = 0) 
4  where tag == iterKind.standalone { 
5    const CPUnumElements = (r.length * (CPUPercent*1.0/100.0));
6    // CPU portion
7    const GPUlo = GPUHi + 1;
8    // GPU portion
9    // Per each locale, create one task
10   coforall locDom in locDoms { 
11      // compute a subrange of CPUrange owned by each task,
12      const myIters = computeChunk(CPUrange, tid, numTasks);
13      for i in myIters do 
14          yield i;
15   } 
16   else if (subloc == 1) { 
17      // sublocale 1: GPUs
18      coforall tid in 0..numTasks { 
19          // invoke the GPUWrapper
20          GPUWrapper(GPUrange, translate(-r.low).first, 
21              GPUrange, translate(-r.low).last, 
22              GPUrange.length); 
23      } 
24 } 
```

Listing 11. GPUIterator implementation (multi locales)

```
1  iter GPU(param tag: iterKind, 
2      D: domain, 
3      CPUPercent: int = 0) 
4  where tag == iterKind.standalone 
5      && isRectangularDom(D) 
6      && D.dist.type <= Block { 
7      var locDoms = D.locDoms;
8      // Per each locale, create one task
9      coforall locDom in locDoms { 
10         const r = locDom.myBlock; 
11         // the same as the single locale implementation 
12         ... 
13      } 
14   }) 
15 } 
```
The implementation only supports a single GPU within a node, but it is trivial to support multiple GPUs.

The implementation of leader iterators to implement zippered forall loops is essentially the same as the standalone implementation.

4.2.2 Supporting Distributed Domains

Supporting distributed domains can be done on top of the above single locale version. Conceptually, like Figure 3, it first divides the domain \( 1 \ldots n \) into \( l \) chunks, where \( l \) is the number locales. Then, as with the single locale case, it further divides each chunk into the CPU and GPU portions.

Listing 11 includes the detailed implementation. In the current implementation, we only support Block-distributed domains.

5 Performance Evaluation

**Purpose:** The goal of this performance evaluation is to validate our GPUIterator implementation on different CPU+GPU platforms and conduct comparative performance evaluations across Chapel’s existing forall version, the GPU-Only version that just invokes a CUDA/OpenCL program, and the GPUIterator version.

**Machine:** We present the performance results on three platforms: two servers and one laptop. The first platform is the DAVinCI cluster at Rice University [14] consisting of multiple Intel Xeon CPU + NVIDIA Tesla M2050 (Fermi) nodes connected via QDR InfiniBand. The platform consists of a 12-core Intel Xeon X5660 running at 2.82GHz with a total main memory size of 48GB. The second platform is the PowerOmics cluster at Rice University [14] consisting of multiple IBM POWER8 CPU (S822L) + NVIDIA Tesla K80 nodes connected via InfiniBand\(^1\). The platform has two 12-core IBM POWER8 CPUs (3.02GHz with a total 256GB of main memory). The third platform is a MacBookPro that consists of 6-core Intel Core i7 running at 2.6GHz with 32GB memory, a built-in Intel UHD Graphics 630 with 1.5GB memory, and an AMD Radeon Pro 560X with 4GB memory.

**Benchmarks:** Table 1 lists five Chapel benchmarks that were used in the experiments: vector copy, matrix multiply, stream, blackscholes, and logistic regression. The implementation of these benchmark can be found at [15]. “Data Size” shows datasets that are used for each platform. For “LOC added/modified”, we report the lines of code added/modified to implement the hybrid version with the GPUIterator compared to the original forall implementation.

**Experimental variants:** Each benchmark was evaluated by comparing the following variants:

- **Original Forall (on CPUs):** Implemented in Chapel using a forall with the default parallel iterator \((1 \ldots n)\) that is executed on CPUs (CHPL_LOCALE_MODEL=Flat). In general, the GPUIterator requires \( 5 \times k + 1 \leftrightarrow k + 4 \times k + 1 \) lines, where \( k \) is the number of GPU cores.
- **GPU-Only (on GPUs):** Implemented using a manually implemented CUDA/OpenCL program, that is invoked from Chapel (CHPL_LOCALE_MODEL=Flat). Note that there is no forall loop in this variant.
- **Hybrid Execution (on CPUs + GPUs):** Implemented using a forall with the GPUIterator (GPU\((1 \ldots n, \ldots)\) and the same CUDA/OpenCL program as the GPU-Only variant, which is executed on CPU threads and/or a single GPU depending on the value of CPUPercent as a percentage \((0 \leq \text{CPUPercent} \leq 100)\) (CHPL_LOCALE_MODEL=GPU). A smaller value of CPUPercent indicates that fewer iterations should be executed on the CPU, and more iterations should be offloaded onto the GPU.

For all the variants, we used the latest Chapel compiler (1.20.0-pre) as of March 27, 2019 with the --fast option. For the Intel Xeon and NVIDIA Tesla M2050 platform (DAVinCI), we set CHPL_TASK=qthread and used the NVIDIA CUDA Compiler (nvcc) 7.0.27 with the --03 --arch sm_20 options for all CUDA variants. For the IBM POWER8 and NVIDIA Tesla K80 platform (PowerOmics), we also set CHPL_TASK=qthread and used nvcc 8.0.61 with the --03 --arch sm_50 options. For the Intel Core i7 and Intel UHD Graphics 630 + AMD Radeon Pro 560X laptop machine, we set CHPL_TASK=fifo\(^2\) and used Apple LLVM version 10.0.0 with the --03 option for all OpenCL variants.

The performance was measured in terms of elapsed milliseconds from the start of the parallel computation(s) to their completion. We ran each variant ten times and reported the mean value.

For the CPU variants (Original Forall and Hybrid Execution), the number of workers per node is the number of physical CPU cores — 12, 24, and 6 for the Intel Xeon, the IBM POWER8, and the Intel Core i7 platforms respectively. For the GPU variants (GPU-Only and Hybrid Execution), input data for kernels are prepared in the Chapel side, are then passed to the native side, and are eventually passed back to the Chapel side after GPU execution has completed, meaning that the native code does perform device memory allocation and data transfers (host-to-device and device-to-host) in addition to kernel invocation. Note that the performance numbers for the GPU variants include such overheads.

5.1 Lines of Code Added/Modified for Using the GPUIterator

Let us first discuss source code additions and modifications required for using the GPUIterator. As shown in Table 1, additions and modifications at the Chapel source code level are very small (\( \leq 11 \) lines\(^3\)). In general, the GPUIterator requires \( 5 \times k + 1 \leftrightarrow k + 4 \times k + 1 \) lines, where \( k \) is the number of GPU cores.

---

\(^1\)This cluster used to have two CPU+GPU nodes. However, due to a permanent hardware failure, only a single CPU+GPU node is currently available.

\(^2\)Even without Chapel, an OpenCL API gets SIGSEGV when it is invoked from a qthread task.

\(^3\)Our definitions of source code “lines” is based on common usage. In theory, all 11 lines could be combined into a single line of source code.
number of forall loops that needs to be wrapped by GPU()}.

4 \times k is for an external GPU function declaration (1 line for forall) plus a wrapper function (typically 3 lines per forall as shown in Listing 7 and Listing 8), and 1 is for "use GPUiterator".

Overall, the GPUiterator provides a portable way to explore different CPU/GPU variants by tweaking the CPU\per cent parameter with minimal source code changes.

### 5.2 Single Locale Performance Numbers

Figures 4, 5, 6, and 7 show speedup values relative to the original forall version on a log scale, respectively for the Intel Xeon CPUs with NVIDIA Tesla M2050 GPU and the IBM POWER8 CPUs with NVIDIA Tesla K80 GPU. In the figures, GPU Only refers to the GPU execution in which the native CUDA/OpenCL code is just invoked from the Chapel side, and Hybrid means the hybrid execution on CPU+GPU using the GPUiterator. Also, CX\%+GY\% means X\% of the whole iteration is executed on the CPU, and the Y\% is executed on the GPU (X+Y=100\%).

**How significant is the overhead of the GPUiterator?**

As shown in Figures 4-7, for all the benchmarks, there is no significant performance difference between the C100\%+G0\% and the forall variants (as well as the C0\%+G100\% and the GPU-Only variants), which indicates that the overhead of the GPUiterator is negligible.

**How fast are GPUs?** For blacksholes, logistic regression, and matrix multiplication, the kernels have enough workloads and the GPU variants significantly outperform the original forall. Specifically, the results show a speedup of up to 126.0x on the Intel Xeon + NVIDIA Tesla M2050 platform, 145.2x on the IBM POWER8 + NVIDIA K80 platform, 4.2x on the Intel Core i7 + Intel UHD 630, and 8.3x on the Intel Core i7 + AMD Radeon Pro 560X.

**Why are CPUs faster in some cases?** For vector copy, and stream, the original forall is the fastest because the kernels are too small to benefit from GPUs. The main bottleneck is obviously data transfers between the CPU and the GPU. Specifically, we further analyzed the stream case on the IBM POWER8 platform to understand why the forall is the fastest. In short, this is due to the overhead of host-to-device (H2D) and device-to-host (D2H) transfers. There are 3 Chapel arrays (A, B, C) and suppose the array size is N. The CUDA variant allocates dA, dB, dC arrays on device memory, each with size = N\%(100-CPUPercent). It performs H2D transfers for input arrays B and C, and D2H transfers for output array A after the kernel completed. The GPUiterator variant with CPUPercent=0 case includes all the data transfers and takes 1.7 sec, while it happens to be larger than the CPU Only time of 0.085. However, if we exclude the data transfer time, the time for running the kernel on the GPU (with CPUPercent=0) is only 0.009s, which is about 9.4x faster than the CPU.

**When is hybrid execution beneficial?** As shown in Figure 6, for blacksholes, C50\%+G50\% exhibits the best performance on the Intel Core i7 + Intel UHD Graphics 630. We further analyzed to understand why the half-half execution is faster than the forall and GPU-Only variants. The primary reason for that is communication costs between the Core i7 CPUs and the UHD GPUs are relatively small compared to the discrete GPU (Radeon Pro). As shown in Table 2, H2D/D2H costs for using UHD are a few orders of magnitude faster than these for using Radeon Pro. Secondly, the cost of the CPU portion and that of the GPU portion (the kernel + H2D/D2H transfers + device memory allocation) are very close, thereby exploiting the full capability of the CPU and the GPU.

### 5.3 Multiple Locales Performance Numbers

Figure 8 shows strong scaling speedup values for blacksholes relative to the original forall version on a single node of the Intel Xeon CPUs with NVIDIA Tesla M2050 GPU. While the original forall variant has good scalability, the GPUiterator variants give further performance improvements due to GPU execution. The results show a speedup of up to 14.9x with 100% GPU execution on four nodes. Also, as with the single locale cases, the overhead of the GPUiterator is negligible.

---

**Table 1.** Benchmarks used in our evaluation (S1: the Intel Xeon + NVIDIA Tesla M2050 platform, S2: the IBM POWER8 + NVIDIA Tesla K80 platform, S3: the Intel Core i7 + Intel UHD Graphics 630 + AMD Radeon Pro 560X).

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Data Size</th>
<th>LOC added/modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector Copy</td>
<td>A Simple Vector Kernel</td>
<td>S1: n = 200 \times 2^{20}, S2: n = 2^{20}, S3: n = 2^{40}</td>
<td>6</td>
</tr>
<tr>
<td>Stream</td>
<td>A Simple Vector Kernel</td>
<td>S1: n = 200 \times 2^{20}, S2: n = 2^{20}, S3: n = 2^{40}</td>
<td>6</td>
</tr>
<tr>
<td>BlackScholes</td>
<td>The Black-Scholes Equation</td>
<td>S1: n = 200 \times 2^{20}, S2: n = 2^{20}, S3: n = 2^{40}</td>
<td>6</td>
</tr>
<tr>
<td>Logistic Regression</td>
<td>A Classification Algorithm</td>
<td>S1, S2: f = 2^{10}, s = 32, S3: f = 2^{10}, s = 32</td>
<td>11</td>
</tr>
<tr>
<td>Matrix Multiplication</td>
<td>Matrix-Matrix Multiply</td>
<td>S1, S2: n = 2^{10}, S3: n = 2^{10}</td>
<td>6</td>
</tr>
</tbody>
</table>

**Table 2.** Breakdown for the GPU-Only execution of blacksholes with a size of n = 2^{24}.

<table>
<thead>
<tr>
<th></th>
<th>UHD 630</th>
<th>Radeon Pro 560X</th>
</tr>
</thead>
<tbody>
<tr>
<td>H2D transfer (sec)</td>
<td>1.4 \times 10^{-2}</td>
<td>9.8 \times 10^{-3}</td>
</tr>
<tr>
<td>Kernel (sec)</td>
<td>5.3 \times 10^{-2}</td>
<td>4.3 \times 10^{-3}</td>
</tr>
<tr>
<td>D2H transfer (sec)</td>
<td>2.4 \times 10^{-5}</td>
<td>1.8 \times 10^{-4}</td>
</tr>
</tbody>
</table>
Figure 4. Performance improvements (log scale) over the original forall (12 workers) on the Intel Xeon + NVIDIA Tesla M2050 platform.

Figure 5. Performance improvements (log scale) over the original forall (24 workers) on the IBM POWER8 + NVIDIA Tesla K80 platform.

Figure 6. Performance improvements (log scale) over the original forall (6 workers) on the Intel Core i7 + Intel UHD Graphics 630 platform.

Figure 7. Performance improvements (log scale) over the original forall (6 workers) on the Intel Core i7 + AMD Radeon Pro 560X platform.
6 Related Work

Chapel is designed to express parallelism as part of language rather than include it as libraries or language extensions such as compiler directives or annotations. Therefore it contains constructs to express parallelism and locality as first-class citizens of the language. Thus it enables users to express parallelism for a wide range of platforms without the need for code specializations. This expressiveness helps the programmers to create portable applications thereby improving programmer productivity.

Sidelnik [3] explores the use of GPU from Chapel applications using the forall work-sharing construct. It provides a new GPU distribution based on Chapel’s user-defined distributions which when used with forall offloads the work on to GPU. Based on the distribution provided, the chapel compiler analyzes the forall loop and either generates C code if the target is CPU or C + CUDA code if the target is GPU. But if the user needs any GPU specific feature such as shared memory, constant cache memory or thread block barriers, they need to invoke them explicitly from the Chapel code thereby exposing many GPU low-level constructs to the programmer. This work does not support multi-node GPUs or multiple GPUs on a single node.

Chu [7] generates OpenCL code instead of CUDA code so that Chapel programs can use more than NVIDIA GPUs. They use Chapel’s hierarchical locales to enable GPU computation by exposing a new GPU locale. They also propose extensions to expose various GPU features such as local memory, grid size and so on. The experiments do not show how well it can scale on multi-node clusters.

7 Conclusions

In this paper, we implemented the GPUIterator, which is a portable programming interface that supports 1) GPU-only execution, 2) execution on multiple CPU+GPU nodes, and 3) hybrid execution of a Chapel forall loop, assuming hand-tuned GPU kernels are available. Performance evaluation is conducted on a wide range of CPU+GPU platforms -i.e., an Intel CPU + an NVIDIA GPU, an IBM POWER8 CPU + an NVIDIA GPU, an Intel CPU + an Intel GPU, and an Intel CPU + an AMD GPU and the results show the GPUIterator allows Chapel programmers to explore different CPU/GPU configurations for achieving high performance with minimal source code changes. In future work, we plan to explore further the possibility of hybrid execution on recent platforms with faster CPU-GPU interconnects (e.g., AMD’s APUs and NVIDIA’s NVLink), and also plan to add the capability of automatically deciding the best CPUPercent.

References


Figure 8. Strong scaling speedups over the original forall on a single node of the Intel Xeon + NVIDIA Tesla M2050 platforms. (1, 2, and 4 nodes, 12 workers/node, \( n = 2^{25} \), blackscholes, CHPL,RCOMM=gasnet)