GPUAPI: Multi-level Chapel Runtime API for GPUs

Akihiro Hayashi
ahayashi@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

Sri Raj Paul
sriraj@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

Vivek Sarkar
vsarkar@gatech.edu
Georgia Institute of Technology
Atlanta, Georgia, USA

Abstract

Chapel is inherently well suited not only for homogeneous nodes but also heterogeneous nodes because they employ the concept of locales, distributed domains, forall/reduce constructs, and implicit communications. However, it is unfortunate that there is room for further improvements in supporting GPU in Chapel.

This paper addresses some of the key limitations of past approaches on mapping Chapel on to GPUs as follows. We introduce the `GPUAPI` module, which provides multi-level abstractions of existing low-level GPU API such as CUDA runtime API. This module allows Chapel programmers to have the option of explicitly manipulating device memory (de)allocation and data transfer API at the Chapel level while maintaining good performance and productivity. The `GPUAPI` module is useful particularly when they dive into lower-level details to incrementally evolve their GPU implementations for improved performance on multiple heterogeneous nodes. We provide two tiers of GPU API: the MID-LOW-level API and the MID-level API. The MID-LOW-level API offers thin wrappers for raw GPU API routines, whereas the MID-level API provides Chapel programmer-friendly interface - i.e., allocating device memory using the `new` keyword. Also, the module allows the coexistence of different levels of API even with the prototype GPU code generator in Chapel 1.24.

Our preliminary performance and productivity evaluations show that the use of the `GPUAPI` module significantly simplifies the manipulation of GPU API in Chapel for multiple CPUs+GPUs nodes while achieving the same performance.

CCS Concepts: • **Software and its engineering → Distributed programming languages**.

Keywords: Chapel, GPUs, GPUIterator, Parallel Iterators, GPU API

CHIUW '21, June 04, 2021, Virtual Format
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06...$15.00
https://doi.org/10.1145/1122445.1122456

ACM Reference Format:
Akihiro Hayashi, Sri Raj Paul, and Vivek Sarkar. 2021. GPUAPI: Multi-level Chapel Runtime API for GPUs. In *CHIUW '21: The 8th Annual Chapel Implementers and Users Workshop, June 04, 2021, Virtual Format*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/1122445.1122456

1 Introduction

There has been a growing interest in accelerators, especially GPU accelerators, in large-scale systems. In the Top 500 list [14], one can see that a significant number of systems consist of heterogeneous nodes with GPUs. As with homogeneous systems, software productivity and portability is still a profound issue for heterogeneous systems. We believe that the use of PGAS (Partitioned Global Address Space) languages [2, 6–9, 13] including Chapel, is a scalable and portable way to achieve high-performance without sacrificing productivity.

As for GPU support in Chapel, some of the past approaches [4, 5] aim at compiling Chapel's `forall` loop to GPUs. Also, in Chapel 1.24, an LLVM-based prototype GPU code generator is introduced. However, to achieve significant performance improvements on GPUs, the current best practice is to prepare a low-level program with CUDA/HIP/OpenCL/SYCL and invoke it from Chapel using the C interoperability feature, which is not always very productive. In our past work, we introduced the `GPUIterator` module [11], which facilitates the invocation of a user-written low-level GPU program across multiple CPU+GPU nodes. However, regardless of the use of the `GPUIterator`, it is unfortunate that the user needs to write a full GPU program that includes the host part -i.e., GPU memory (de)allocation, host-device/device-host data transfer, and the device part - i.e., GPU kernels.

Our key observation is that the complexity comes not only from writing GPU kernels in the device part, but also from writing the host part. In particular, interfacing Chapel objects to raw C/C++ pointers is tedious and error-prone, especially because Chapel itself has a well-defined type system with type inference.

In this paper, we propose the `GPUAPI` module, which includes a wide variety of Chapel-level GPU API that facilitates device memory (de)allocation and device-to-host/host-to-device data transfer. The `GPUAPI` module is designed to comply with Chapel's multi-resolution concept, where the user has the option of providing a high-level specification and also of diving into lower-level details to incrementally evolve

Listing 1. A Chapel program with the `GPUIterator` module.

```
use GPUIterator;
proc GPUCallBack(lo: int, hi: int, nElems: int) {
  // The GPU portion (lo, hi, nElems) is automatically computed
  // even in multi-locale + multi-GPUs settings.
  // Also, hi-lo+1 == nElems
  myGPUCode(...);
};
var CPUpercent = x; // X% goes to the CPU
                    // (100 - X)% goes to the GPU
// D can be a distributed domain
forall i in GPU(D, GPUCallBack, CPUPercent) {...}
```

their implementations for improved performance on multiple CPUs+GPUs nodes. Specifically, we propose to introduce a spectrum of GPU programming abstraction in Chapel [12]:

- **HIGH-level/HIGH-MID-level**: The compiler compiles `forall`/`reduce` constructs to GPUs and generates all the host part required for GPU execution (HIGH). For the host part, the user may want to use explicit GPU API to optimize data transfer (HIGH-MID).
- **MID-level/MID-LOW-level**: The user writes 1) GPU kernels in a low-level GPU language, and 2) the host part in Chapel in either/both of two levels of abstraction: a Chapel programmer-friendly version (MID), and a thin wrapper version of raw GPU API routines (MID-LOW).
- **LOW-level**: The user writes a full GPU program in a low-level GPU language. The `GPUIterator` module may be used to facilitate the distributed execution and/or hybrid execution.

To the best of our knowledge, this paper is the first paper that discusses the design and implementation of MID-level and MID-LOW level GPU API. We also discuss the composability of different levels of GPU abstraction, including the `GPUIterator` module and even the HIGH-level abstraction, which is the prototype GPU code generator introduced in Chapel 1.24.

This paper makes the following contributions:

- The design and implementation of multi-level yet composable Chapel GPU API.
- Performance evaluations and productivity discussion using different distributed mini applications and a real-world application [1] on different CPU+GPU systems.

2 Background

2.1 The `GPUIterator` module

In our past work [11], we introduced the `GPUIterator` module, which facilitates the invocation of a user-written low-level GPU program. The module provides a parallel iterator for a `forall` loop, in which the iteration space is divided into two spaces: a CPU and GPU space. The original `forall` iterating over the CPU space is executed on the CPUs. Similarly, for the GPU space, it invokes a user-written callback function where a low-level GPU program is invoked with the divided GPU space.

Listing 1 shows an example of a Chapel program with the module.The domain `D` is wrapped in the `GPU()` iterator. The `GPUCallBack()` is invoked once the module has computed a CPU and GPU space, and the user is supposed to write the invocation of low-level GPU code (`myGPUCode()`) in it. Also, the user can tweak the CPU/GPU percentage by changing the `CPUPercent` (100% goes to the GPU if the user omits the argument).

Let us emphasize that the module is designed to facilitate multi-locale, multi-GPUs, plus hybrid execution in a portable way. This feature is significant because many of the past approaches that tackle GPU execution in Chapel do not support such a feature. To handle multi-GPUs per locale, the module automatically computes a subspace for each GPU and implicitly calls the callback function multiple times - i.e., the number of GPUs per locale × the number of locales. Because the module implicitly sets the device ID for each GPU, all the user has to do is 1) to write a code snippet that gets a local portion of a distributed array in the Chapel part, 2) to make the device part flexible to change in iteration spaces -i.e., making it aware of `lo`, `hi`, `nElems`, and 3) not to put a device setting call.

Listing 2 and Listing 3 illustrate an example distributed implementation of the STREAM benchmark (`A = B + alpha*C`) that enables distributed hybrid execution on multple CPUs+GPUs locales. On line 17 in Listing 2, in the `GPUCallBack` function, it obtains a local portion of the distributed array `A`, `B`, and `C` using the `localSlice()` API, which is fed into the external C function `cudaSTREAM()` along with a subspace for each GPU (`lo`, `hi`, and `nElems`). The GPU part in Listing 3 includes a typical host program including device memory (de)allocation, data transfer, and kernel invocation. Note that the kernel (line 3 in Listing 3) is flexible to change in iteration space because it only iterate over `0..#nElems` that is given by the Chapel part. Also, since `localSlice(lo..hi)` returns a pointer to the head of the local slice, it is safe to assume that `&A[0]`, `&B[0]`, and `&C[0]` in the host part point to `A[lo]`, `B[lo]`, and `C[lo]` in the Chapel part respectively.

2.2 The GPU code generator in Chapel

In version 1.24, the Chapel compiler includes an LLVM-based GPU compiler. While it is advertised as a "prototype" feature, it compiles a Chapel function to a CUDA binary using the NVPTX backend. In the current implementation, the user is supposed to write boilerplate code that loads the CUDA binary as well as the host part, which would be simplified in future releases. For example, Listing 4 shows a part of `gpuAddNums` example from ([3]). The `launchKernel()` function on line 6 is a C function that performs 1) host operations that are similar to `cudaSTREAM()` in Listing 3, and 2) an operation that loads a CUDA binary (a `fatbin` file) into the

Listing 2. An example distributed implementation of STREAM(The Chapel part).

```
/* stream.chpl */
use BlockDist; use GPUIterator; use GPUAPI;

extern proc cudaSTREAM(A: [] real(32), B: [] real(32), C: [] real(32),
                       alpha: real(32), lo: int, hi: int, nElems: int);

config const n = 1024: int;
config const CPUPercent = 0: int;
var D: domain(1) dmapped Block(boundingBox={0..#n}) = {0..#n};
var A: [D] real(32);
var B: [D] real(32);
var C: [D] real(32);
var alpha: real(32) = 0.5;

proc GPUCallBack(lo: int, hi: int, nElems: int) {
  // lo, hi, nElems plus device ID is automatically set here
  cudaSTREAM(A.localSlice(lo..hi), B.localSlice(lo..hi),
             C.localSlice(lo..hi), alpha, lo, hi, nElems);
};
...
forall i in GPU(D, GPUCallBack, CPUPercent) {
  A[i] = B[i] + alpha * C[i]; // CPU Version
}
```

Listing 3. An example distributed implementation of STREAM (The GPU part)

```
/* stream.cu */
// the kernel part
__global__ void stream(float *dA, float *dB, float *dC,
                       float alpha, int nElems) {
  int id = blockIdx.x * blockDim.x + threadIdx.x;
  if (id < nElems) dA[id] = dB[id] + alpha * dC[id];
}
// the host part
extern "C" {
void cudaSTREAM(float* A, float *B, float *C, float alpha,
                int64_t start, int64_t end, int64_t nElems) {
  assert((end-start+1) == nElems);
  float *dA, *dB, *dC;
  cudaMalloc(&dA, sizeof(float) * nElems);
  cudaMalloc(&dB, sizeof(float) * nElems);
  cudaMalloc(&dC, sizeof(float) * nElems);
  cudaMemcpy(dB, B, sizeof(float) * nElems, cudaMemcpyHostToDevice);
  cudaMemcpy(dC, C, sizeof(float) * nElems, cudaMemcpyHostToDevice);
  stream<<<ceil(((float)nElems)/)1024, 1024>>>(dA, dB, dC,
                                               alpha, nElems);
  cudaDeviceSynchronize();
  cudaMemcpy(A, dA, sizeof(float) * nElems, cudaMemcpyDeviceToHost);
  cudaFree(dA);
  cudaFree(dB);
  cudaFree(dC);
}}
```

device that is generated from the add_nums function on line 11.

3 Design

3.1 Motivation

While the GPUIterator module provides a portable way to perform distributed, hybrid, and multi-GPU execution, in terms of productivity, there is room for further improvements. As shown in Listing 3, most of the host part includes

Listing 4. A part of the gpuAddNums example introduced in Chapel 1.24

```
 1 // A part of gpuAddNums.chpl
 2 extern {
 3   // The path to the GPU binary
 4   #define FATBIN_FILE "tmp/chpl__gpu.fatbin"
 5   // The host part (load GPU binary, data (de)allocation and transfer)
 6   static double launchKernel() { ... }
 7 }
 8
 9 pragma "codegen for GPU"
10 pragma "always resolve function"
11 export proc add_nums(dst_ptr: c_ptr(real(64))){
12   dst_ptr[0] = dst_ptr[0]+10;
13 }
14 launchKernel();
```

device memory (de)allocation and host-to-device/device-to-host transfer, which is relatively larger than the kernel invocation and the kernel itself. Note that the complexity of the host part can significantly grow as the kernel part grows. More importantly, in this low-level program, the user has to deal with raw C pointers and the size of the allocated memory regions, which is abstracted away in the main Chapel program. This motivates us to design and implement a set of Chapel-level GPU API which mitigates the complexity of handling the low-level host part, thereby improving productivity.

In this paper, we mainly focus on the design and implementation of MID-level/MID-LOW-level explicit GPU API discussed in Section 1. We believe this level of abstraction is still important even when fully automatic approaches (the HIGH-level abstraction) are available because 1) compiler-generated kernels would not always outperform user-written kernels or highly-tuned GPU libraries, and 2) it would not be always trivial for the compiler to perform data transfer optimizations such as data transfer hoisting. Therefore, MID-level/MID-LOW-level GPU API comes in portions that remain as performance bottlenecks even after automatic compilation approaches, which should comply with Chapel's multi-resolution concept.

3.2 MID-LOW-level API: Thin wrappers for raw GPU routines

At the MID-LOW-level, most of the low-level 1) device memory allocation, 2) device synchronization, and 3) data transfer can be written in Chapel. This level of abstraction only provides thin wrapper functions for the CUDA/HIP/OpenCL-level API functions, which requires the user to directly manipulate C types like c_void_ptr and so on. The MID-LOW level API is helpful, particularly when the user wants to fine-tune the use of GPU API but still wants to stick with Chapel.

Listing 5 is an example program written with the MID-LOW-level API. On line 2, use GPUAPI; is added to use the GPUAPI module. Also, since this version manipulates raw C

Listing 5. An example distributed implementation of STREAM (The MID-LOW version).

```
 1 /* steram-mid-low.chpl */
 2 use BlockDist; use GPUIterator; use GPUAPI; use SysCTypes;
 3 proc GPUCallBack(lo: int, hi: int, nElems: int) {
 4   var dA, dB, dC: c_void_ptr; // device memory pointers
 5   ref lA = A.localSlice(lo..hi);
 6   ref lB = B.localSlice(lo..hi);
 7   ref lC = C.localSlice(lo..hi);
 8   const size: size_t = (lA.size:size_t * c_sizeof(lA.eltType));
 9   Malloc(dA, size);
10   Malloc(dB, size);
11   Malloc(dC, size);
12   Memcpy(dB, c_ptrTo(lB), size, H2D);
13   Memcpy(dC, c_ptrTo(lC), size, H2D);
14   cudaSTREAM_kernel(dA, dB, dC, alpha,
15                         lo, hi, nElems);
16   DeviceSynchronize();
17   Memcpy(c_ptrTo(lA), dA, size, D2H);
18   Free(dA);
19   Free(dB);
20   Free(dC);
21 };
22 ...
23 /* stream-kernel.cu */
24 void cudaSTREAM_kernel(float* dA, float *dB, float *dC, float alpha,
25                  int start, int end, int nElems) {
26      // the kernel code remains the same
27      stream<<<ceil(((float)nElems)/1024), 1024>>>(dA, dB, dC,
28                                        alpha, start, end, nElems);
29 }
```

pointers, use `SysCTypes;` is also required. From line 9 to line 20, there is a sequence of the host code including `Malloc()`, `Memcpy()`, a kernel invocation, `DeviceSynchronize()`, and `Free()`. Each GPU API routine is essentially a thin wrapper for the corresponding CUDA API (e.g., `cudaMalloc()`, `cudaMemcpy()`, `cudaDeviceSynchronize()`, and `cudaFree()`). Note that the first argument to `Malloc()` is a `ref` variable, and a pointer to allocated device memory is assigned once the allocation is done.

Now that all of the host part except for the kernel invocation is done at the Chapel level, the low GPU program part only includes a CUDA kernel invocation (see line 24). While this MID-LOW-level abstraction simplifies the host code compared to the original host part in Listing 3, notice that the user still needs to handle C pointers explicitly (e.g., `c_void_ptr`, `c_sizeof`, and `c_ptrTo()`).

Pitched Memory Allocation and 2D Data Transfer: In addition to `Malloc()` and `Memcpy()`, which are linear memory allocation and data transfer, the `GPUAPI` module also supports pitched memory allocation (`MallocPitch()`) and 2D data transfer (`Memcpy2D()`). The pitched memory allocation API takes 2D shape information - i.e., `width` and `height`, and the underlying raw routine may add a fixed pad (`pitch`) to ensure high memory bandwidth on the device. The 2D data transfer API is a variant of `Memcpy()`, which is aware of the pad information.

Listing 6 shows a standalone example program with the pitched memory allocation and 2D data transfer. First, the 2D domain (`D`) on line 1 is used to construct the 2D array

Listing 6. Allocating pitched memory and perform 2D memcpy

```
1 var D = {0..255, 0..255};
2 var A: [D] real(32) = 1.0;
3 var widthInBytes: size_t = D.dim(1).size:size_t * c_sizeof(A.eltType);
4 var spitch = widthInBytes;
5 var dA: c_void_ptr;
6 var dpitch: size_t;
7 MallocPitch(dA, dpitch, widthInBytes, D.dim(0).size:size_t);
8 Memcpy2D(dA, dpitch, c_ptrTo(A), spitch, widthInBytes,
9          D.dim(0).size:size_t, 0);
```

Listing 7. An example distributed implementation of STREAM. (The MID version)

```
 1 use BlockDist; use GPUIterator; use GPUAPI; /* use SysCTypes; */
 2 proc GPUCallBack(lo: int, hi: int, nElems: int) {
 3   // nElems * sizeof(int) will be automatically allocated onto the device
 4   var dA = new GPUArray(A.localSlice(lo..hi));
 5   var dB = new GPUArray(B.localSlice(lo..hi));
 6   var dC = new GPUArray(C.localSlice(lo..hi));
 7   dB.toDevice();
 8   dC.toDevice();
 9   cudaSTREAM_kernel(dA.dPtr(), dB.dPtr(), dC.dPtr(), alpha,
10                         lo, hi, nElems);
11   DeviceSynchronize();
12   dA.fromDevice();
13   // allocate GPU memory automatically deallocated
14 }
```

(`A`) on line 2. The arguments to `MallocPitch()` on line 7 are as follows: `dA` is a `ref` variable that stores a pointer to allocated device memory, `dpitch` is also a `ref` variable that stores pitch on the device, `hpitch` is the width of the Chapel array in bytes, and the last argument is the height of the Chapel array (# of elements).

3.3 MID-level API: A Chapel programmer friendly GPU API

At the MID-level, as with the MID-LOW-level, most of the low-level 1) device memory allocation, 2) device synchronization, and 3) data transfer can be written in Chapel. The key difference between the MID-LOW and the MID levels is that the MID-level API is more Chapel programmer-friendly. The user can allocate GPU memory using the `new` keyword and no longer need to manipulate C types explicitly.

Listing 7 shows an example program written with the MID-level API. As shown on line 4-6, device memory allocation can be done using `new GPUArray()`. The corresponding device pointer can be obtained by invoking `dPtr()` (line 9). Host-to-device and device-to-host transfer can be done by using `toDevice()` and `fromDevice()` respectively (line 7, 8, and 12) Note that no device memory deallocation is required because the deinitializer of `GPUArray` is automatically invoked to handle the deallocation as with typical Chapel class objects. In case the user wants to manually manage device memory, this can be done by doing `var dA = new unmanaged GPUArray(A);` and `delete dA;`.

Comparing Listing 7 with Listing 5 and Listing 3, one can see that the use of the MID-level API significantly simplifies the host part.

The following discusses the details of API provided at the MID level.

`class GPUArray`: This class encapsulates the allocation, deallocation, and transfer of device memory. It can accept a multi-dimensional Chapel array and internally allocates linear memory for it. For 2D Chapel arrays, the user has the option of using pitched memory by adding `pitched=true` to the constructor call, and the allocated pitch can be obtained using `pitch()` method.

`class GPUJaggedArray`: This class encapsulates the allocation, deallocation, and transfer of jagged device memory. We introduce this class because a real-world but proprietary Chapel program heavily uses this pattern. Instead of discussing the proprietary application, let us discuss our motivation using a simple Chapel program. Consider the Chapel code shown in Listing 8. There is a declaration of `class C` (line 1-5), which includes an array (`x`). Also, on line 7, an array of C, namely `Cs`, is created. When mapping `Cs` onto the device, since `Cs` is a heterogeneous array, it is required to create an array of an array using `Malloc()`. Line 10 shows an example implementation using the MID-LOW level API. Essentially, it first performs `Malloc()` and `Memcpy()` for each `Cs[0].x` and `Cs[1].x`, then performs another `Malloc()` and `Memcpy()` for allocating a device memory region that stores pointers to the device counterpart of `Cs[0].x` and `Cs[1].x`. On the other hand, the MID-level version (line 24) saves a lot of lines. Essentially like the `GPUArray` class, all the user has to do is put `Cs.x` into the constructor of `GPUJaggedArray`. Thanks to the promotion feature of Chapel, `Cs.x` is promoted to `Cs[0..#2].x` and the jagged array class internally performs the same thing as the MID-LOW version does.

Listing 8. A jagged array example.

```
 1 class C {
 2   var n: int;
 3   proc init(_n: int) { n = _n; }
 4   var x: [0..#n] int;
 5 }
 6
 7 var Cs = [new C(256), new C(512)];
 8 const N = Cs.size;
 9
10 // MIDLOW
11 {
12     var dA: [0..#N] c_void_ptr;
13     var dAs: c_ptr(c_void_ptr);
14     for i in 0..#N {
15        const size = Cs[i].x.size:size_t*c_sizeof(int);
16        Malloc(dA[i], size);
17        Memcpy(dA[i], c_ptrTo(Cs[i].x), size, 0);
18     }
19     const size = N: size_t * c_sizeof(c_ptr(c_void_ptr));
20     Malloc(dAs, size);
21     Memcpy(dAs, c_ptrTo(dA), size, 0);
22     // kernel invocation
23 }
24 // MID
25 {
26     var dAs = new GPUJaggedArray(Cs.x);
27     dAs.toDevice();
28     // kernel invocation
29 }
```

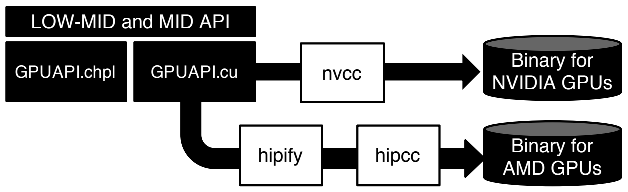


Figure 1. The implementation of the `GPUAPI` module.

4 Implementation

We implemented the `GPUAPI` module as an external Chapel module. The module can be used either standalone or with the `GPUIterator` module. The actual implementation and the detailed documentation can be found at [15, 16].

The MID-LOW-level API routines are implemented as thin wrappers for raw CUDA routines to minimize the overhead. Also, the MID-level API classes are implemented on top of the MID-LOW-level API, which enables multiple levels of GPU API code can coexist, thereby improving productivity and composability. It is worth noting that MID-level and MID-LOW-level API work well with the prototype GPU code generator in Chapel 1.24, which can significantly facilitate the host part.

In the current implementation, the module mainly supports NVIDIA CUDA-supported GPUs and AMD ROCm-supported GPUs. One of the interesting aspects of our implementation is that there is only a CUDA implementation of the `GPUAPI` module. We utilize the `hipify` tool from AMD to convert the CUDA implementation to a HIP version. More specifically, at the time of installation, our cmake-based build system does the conversion and generates a binary if a target system has an AMD GPU (Figure 1).

Also, it is worth noting that the OpenCL backend is also available and we are currently developing a SYCL backend.

5 Performance and Productivity Evaluations

Purpose: In this evaluation we validate our `GPUAPI` implementation on different CPU+GPU platforms. We mainly discuss the performance and productivity of different levels of GPU API (LOW, MID-LOW, MID) with the `GPUIterator` module. The goal is to demonstrate 1) there is no significant performance difference between the LOW, MID-LOW, and MID versions, and 2) the use of a higher-level API improves the productivity in terms of lines of code.

Machine: We present the performance results on three platforms: a GPU cluster, a supercomputer and a server. The first platform is the Cori GPU nodes at NERSC, each node of which consists of two sockets of 20-core Intel Xeon Gold 6148 running at 2.40 GHz with a total main memory size of 384GB and 8 NVIDIA Tesla V100 GPUs, each with 16 GB HBM2 memory, connected via PCIe 3.0[1]. The second platform is the Summit supercomputer at ORNL, which consists of the IBM Power System AC922 nodes. Each node contains two IBM POWER9 running at 3.45GHz with a total main memory size of 512GB and 6 NVIDIA Tesla V100 GPUs, each with 16GB HBM2 memory, connected via NVLink. The third platform is a single-node AMD server, which consists of 12-core Ryzen9 3900X running at 3.8GHz and a Radeon RX570 GPU with 8GB memory.

Benchmarks: We use four distributed mini-applications (Stream, BlackScholes, Matrix Multiplication, and Logistic Regression) and a distributed Tree Search implementation as a real-world example. We use an input data size of $n = 2^{30}$ (Stream, BlackScholes), $n \times n = 4096 \times 4096$ (MM), $nFeatures = 2^{18}, nSamples = 2^4$ (Logistic Regression), and $n = 2^{18}$ (Tree Search). We report the average performance number from 5 runs.

Experimental variants: Each benchmark is evaluated by comparing the following variants:

- **Chapel-CPU**: Implemented in Chapel using a `forall` with the default parallel iterator that is executed on CPUs.
- **Chapel-GPU**: Implemented using a `forall` with the `GPUIterator` module.
  - **MID-level**: All the GPU part except for GPU kernels is implemented using the MID-level API, which is a Chapel class based abstraction of GPU arrays.
  - **MID-LOW-level**: All the GPU part except for GPU kernels is implemented using the MID-LOW-level API, which is a set of thin wrappers for raw GPU API routines.
  - **LOW-level**: The GPU part is fully implemented in CUDA (on NVIDIA GPUs) or HIP (on AMD GPUs).

5.1 Distributed Mini Applications

Figure 2, 3, and 4 show speedup values relative to the Chapel-CPU version on a log scale. In the figures *GPU(M)*, *GPU(ML)*, *GPU(L)* refers to MID-level, MID-LOW-level, and LOW-level respectively. While we use the Chapel compiler version 1.20 with the `--fast` option, `CHPL_COMM=gasnet`, `CHPL_COMM_SUBSTRATE=ibv`, and `CHPL_TASK=qthreads` in this evaluation, we believe the performance trend will not change when the latest Chapel version is used.

As shown in Figure 2-4, for all the benchmarks, there is no significant performance difference between the MID, MID-LOW, and LOW versions, which indicates that the overhead of the `GPUAPI` module can be ignored.

[1]Interconnection network between the GPUs is NVLink.

Application	Level	Chapel	Host (CUDA)	Kernel (CUDA)
Stream	LOW	4	13	6
	MID-LOW	16	1	6
	MID	8	1	6
BlackScholes	LOW	4	13	68
	MID-LOW	16	1	68
	MID	8	1	68
Matrix Multiplication	LOW	3	12	10
	MID-LOW	14	1	10
	MID	8	1	10
Logistic Regression	LOW	2	15	13
	MID-LOW	16	1	13
	MID	10	1	13
Tree Search	LOW	2	16	71
	MID-LOW	13	4	71
	MID	9	4	71

Table 1. Source code additions and modifications required for using the `GPUAPI` module in terms of source lines of code (SLOC).

Table 1 shows source code additions and modifications required for using the GPUAPI. We measure the productivity in term of souece lines of code[2]. The goal of this productivity experiment is to demonstrate SLOC for both the Chapel part and the host part are reduced when the MID-level API is used. Note that the CUDA kernel part is out of the scope of this paper. The results show 1) the MID-LOW level version requires almost the same lines of code as the LOW-level version, and 2) the use of the MID-level API significantly decreases the lines of code. Let us reiterate that the MID-level simplifies the host part more than what it appears as the lines of code reduction because it avoids the explicit manipulation of raw C pointers.

In terms of performance improvements over Chapel-CPU, for Blackscholes, Matrix Multiplication, and Logistic Regression, the kernels have enough workloads, and the GPU variants significantly outperform the Chapel-CPU. Specifically, the results show a speedup of up to 21k × on the Cori supercomputer, 20k × on the Summit supercomputer, and 211 × on the AMD server over the 1 node execution of Chapel-CPU. For Stream, the Chapel-CPU outperforms the GPU variants because the data transfer time is significantly larger than the kernel time. Note that if we only compare the kernel times, the GPU kernel is faster. However, let us reiterate that our primary focus is to prove that there is no significant performance difference between the three Chapel-GPU variants. Also, the use of the `GPUIterator` can help the user to easily switch back and forth between the Chapel-CPU and the Chapel-GPU versions.

5.2 Real-world Example: Distributed Tree Search

Here we present the performance and productivity of the `GPUAPI` module using a real-world application: distributed tree search [1]. In this evaluation, we use the latest Chapel compiler version 1.24 with the `--fast` option, `CHPL_COMM=gasnet`,

[2]Our definitions of source code "lines" is based on common usage.

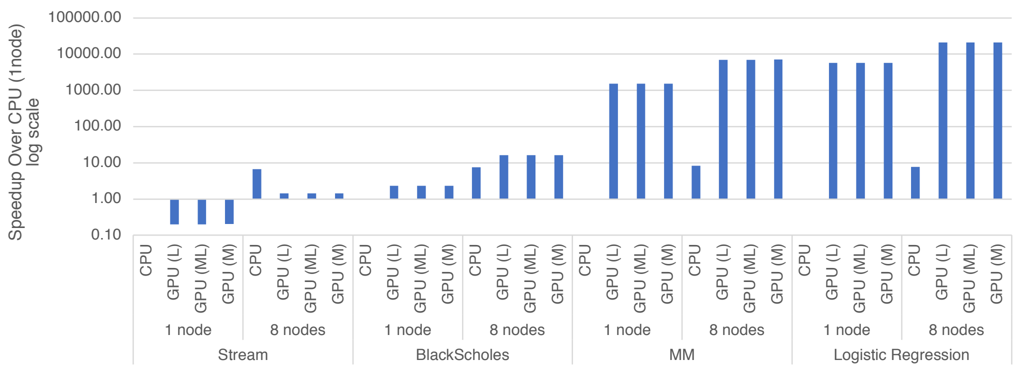


Figure 2. Performance improvements of mini applications on the Cori supercomputer (log scale, multi-nodes: 1GPU/node)

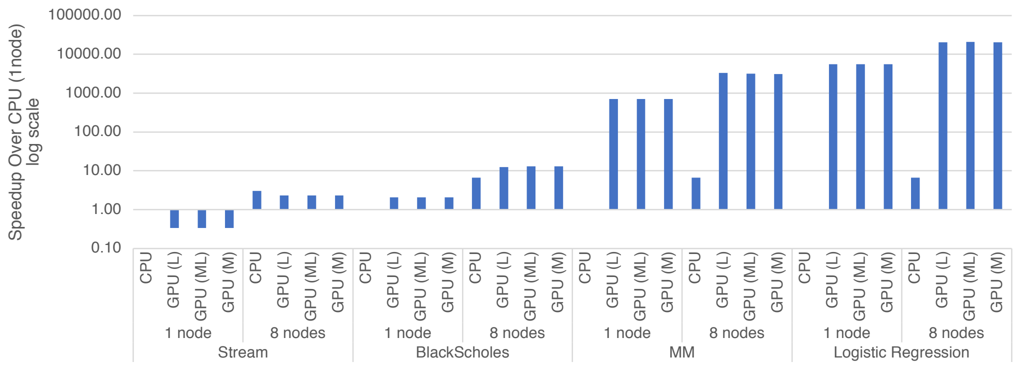


Figure 3. Performance improvements of mini applications on the Summit supercomputer (log scale, multi-nodes: 1GPU/node)

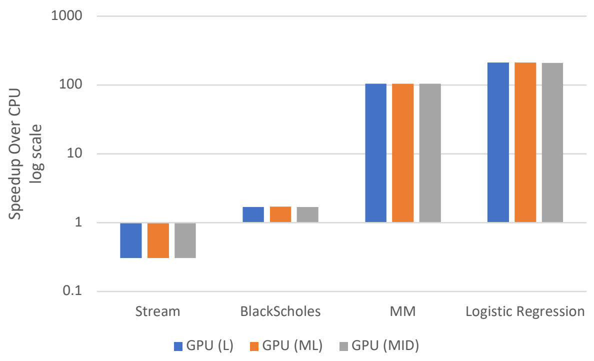
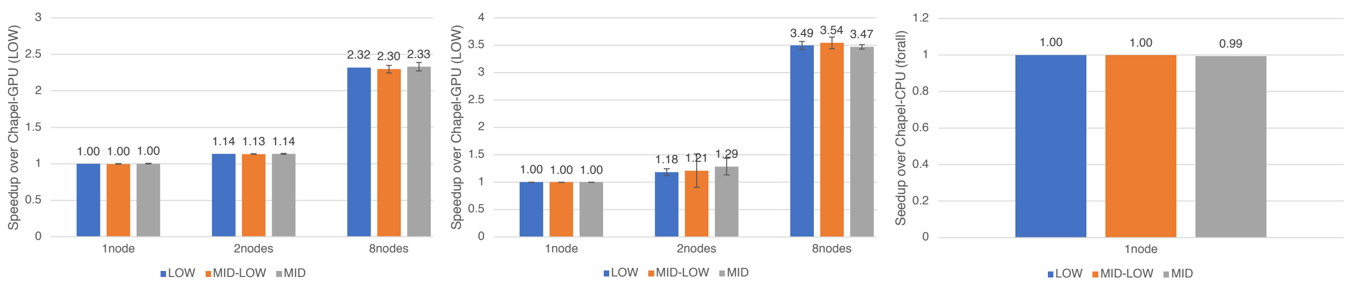


Figure 4. Performance improvements of mini applications on the AMD server(log scale, single-node:1GPU/node)

`CHPL_COMM_SUBSTRATE=ibv`, and `CHPL_TASK=qthreads`. Note that there is no Chapel-CPU version of this application.

Figure 5a, 5b, and 5c, show speedup values relative to the LOW version on a single node with the 95% confidence intervals. Note that, on the Summit supercomputer, 6 GPUs / locale are used without any modifications to the source code thanks to the `GPUIterator` module, while the use of multiple GPUs gives an error that is unrelated to our modules on the Cori supercomputer. As with the mini applications discussed in Section 5.1, while there are slight performance differences, the use of the 95% confidence intervals indicates that there is no statistically significant performance difference between the LOW, MID-LOW, and MID versions. Because this application is highly irregular, the strong scalability is not as good as that of the mini applications. However, improving the scalability is orthogonal to this work.

Also, the last row of Table 1 shows source code additions and modifications required for this application. The results also show the same trends as the other mini-applications, where a higher-level GPU API simplifies the Chapel and host parts.



(a) Cori (Multi-nodes: 1GPU/node) **(b)** Summit (Multi-nodes: 6GPUs/node) **(c)** The AMD server(Single-node:1GPU/node)

Figure 5. Performance improvements of the distributed tree search application.

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. Due to this design, many of the constructs that support parallelism are treated as first-class citizens of the language. Since locality is also important in achieving performance in parallel programs, the locality constructs are also included as a first-class citizen in the Chapel language. Chapel allows expressing parallelism at various granularity for a wide range of platforms without the need for code specialization. This expressiveness of parallelism helps programmers to create portable parallel programs, thereby improving their productivity.

Sidelnik [4] used the `forall` work-sharing construct to explore the use of GPU for chapel applications. Using Chapel's user-defined distributions, they expose a new GPU distribution that offloads the work to GPU when `forall` is used. The chapel compiler analyzes the `forall` loop, and based on the distribution provided, it either generates C code if the target is CPU or C + CUDA code if the target is GPU. The user needs to explicitly invoke any GPU-specific feature such as shared memory, constant cache memory, or thread block barriers. This exposes the GPU low-level constructs to the Chapel programmer. In their work, they do not support multi-node GPUs or multiple GPUs on a single node.

Chu [5] generates OpenCL for chapel constructs using a Chapel-to-OpenCL pass in the compiler that runs along with the Chapel-to-C code generation. They use OpenCL code instead of CUDA code so that Chapel programs can use more than NVIDIA GPUs. They use the Radeon Open Compute Platform (ROCm) for creating and launching kernels for execution. They expose a new GPU locale using Chapel's hierarchical locales to enable GPU computation and also propose extensions to expose various other GPU features such as local memory, grid size, and so on. They have no results that scale to multi-node clusters or GPUs other than that of AMD.

Ghangas [10] extended support to accommodate complex expressions. In this case, a Chapel statement containing multiple arrays is offloaded to the GPU with a single kernel. They also extended support to multiple vendors other than AMD. However, performance results have not been demonstrated yet.

7 Conclusions

In this paper, we implemented the `GPUAPI` module, which allows Chapel programmers to have the option of explicitly manipulating device memory (de)allocation API, and data transfer API at the Chapel level. While it can be used standalone, when it is used with the `GPUIterator` module, it significantly facilitates distributed and hybrid execution on multiple CPU+GPU nodes.

Our preliminary performance evaluation using mini-applications and a real-world application is conducted on a wide range of CPU+GPU platforms - i.e., Intel Skylake CPUs + NVIDIA V100 GPUs, IBM POWER9 CPUs + NVIDIA V100 GPUs, and AMD Ryzen9 CPUs + an AMD Radeon RX GPU. The results show that the use of the `GPUAPI` and `GPUIterator` modules is a promising approach for Chapel programmers to easily utilize multiple CPU+GPU node(s) while maintaining portability.

In future work, we plan to explore further the possibility of using our modules in different real-world Chapel applications.

Acknowledgement

The authors would like to thank Tiago Carneiro and Nouredine Melab for giving feedback on the `GPUIterator` and using it in their distributed tree search application.

This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. Also, this research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] Tiago Carneiro, Nouredine Melab, Akihiro Hayashi, and Vivek Sarkar. 2021. Towards Chapel-based Exascale Tree Search Algorithms: dealing with multiple GPU accelerators. In *HPCS 2020 - The 18th International Conference on High Performance Computing & Simulation*. Barcelona / Virtual, Spain. https://hal.archives-ouvertes.fr/hal-03149394

[2] Bradford L. Chamberlain. 2011. Chapel (Cray Inc. HPCS Language). In *Encyclopedia of Parallel Computing*. 249–256. https://doi.org/10.1007/978-0-387-09766-4_54

[3] Chapel. 2021. The gpuAddNums example. https://github.com/chapel-lang/chapel/blob/master/test/gpu/native/gpuAddNums/gpuAddNums.chpl.

[4] Albert Sidelnik et al. 2012. Performance Portability with the Chapel Language *(IPDPS '12)*. IEEE Computer Society, Washington, DC, USA, 582–594. https://doi.org/10.1109/IPDPS.2012.60

[5] Michael L. Chu et al. 2017. GPGPU support in Chapel with the Radeon Open Compute Platform (Extended Abstract) *(CHIUW'17)*.

[6] Philippe Charles et al. 2005. X10: an object-oriented approach to non-uniform cluster computing *(OOPSLA'05)*. ACM, New York, NY, USA, 519–538.

[7] Sanjay Chatterjee et al. 2013. Integrating Asynchronous Task Parallelism with MPI *(IPDPS '13)*. IEEE Computer Society, Washington, DC, USA, 712–725. https://doi.org/10.1109/IPDPS.2013.78

[8] William W Carlson et al. 1999. *Introduction to UPC and language specification*. Technical Report.

[9] Yili Zheng et al. 2014. UPC++: A PGAS Extension for C++ *(IPDPS'14)*. 1105–1114. https://doi.org/10.1109/IPDPS.2014.115

[10] Rahul Ghangas and Josh Milthorpe. 2020. Chapel on Accelerators. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2020, New Orleans, LA, USA, May 18-22, 2020*. IEEE, 679. https://doi.org/10.1109/IPDPSW50202.2020.00121

[11] Akihiro Hayashi, Sri Raj Paul, and Vivek Sarkar. 2019. GPUIterator: Bridging the Gap between Chapel and GPU Platforms. In *Proceedings of the ACM SIGPLAN 6th on Chapel Implementers and Users Workshop* (Phoenix, AZ, USA) *(CHIUW 2019)*. Association for Computing Machinery, New York, NY, USA, 2–11. https://doi.org/10.1145/3329722.3330142

[12] A. Hayashi, S. Raj Paul, and V. Sarkar. 2020. Exploring a multi-resolution GPU programming model for Chapel. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 675–675. https://doi.org/10.1109/IPDPSW50202.2020.00117

[13] Robert W. Numrich and John Reid. 1998. Co-array Fortran for Parallel Programming. *SIGPLAN Fortran Forum* 17, 2 (Aug. 1998), 1–31. https://doi.org/10.1145/289918.289920

[14] The TOP500 project. 2021. TOP500 LISTS. https://www.top500.org/lists/top500/.

[15] Akihiro Hayashi Vivek Sarkar, Sri Raj Paul. 2019. GPUIterator and GPUAPI Implementation. https://github.com/ahayashi/chapel-gpu.

[16] Akihiro Hayashi Vivek Sarkar, Sri Raj Paul. 2020. GPUIterator and GPUAPI Documentation. https://ahayashi.github.io/chapel-gpu/.