A Multi-Level Platform-Independent GPU API for High-Level Programming Models

Akihiro Hayashi[1(✉)], Sri Raj Paul[2], and Vivek Sarkar[1]

[1] Georgia Institute of Technology, Atlanta, GA 30332, USA
{ahayashi,vsarkar}@gatech.edu
[2] Intel Corporation, Austin, USA
sriraj.paul@intel.com

Abstract. While there has been a growing interest in supporting accelerators, especially GPU accelerators, in large-scale systems, the user typically has to work with low-level GPU programming models such as CUDA along with the low-level message passing interface (MPI).

We believe higher-level programming models such as Partitioned Global Address Space (PGAS) programming models enable productive parallel programming at both the intra-node and inter-node levels in homogeneous and heterogeneous nodes. However, GPU programming with PGAS languages in practice is still limited 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. Thus, it is not rare that the user eventually writes a fully external 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, and calls it from their primary language, which is not very productive.

Our key observation is that the complexity of writing the external GPU program comes not only from writing GPU kernels in the device part, but also from writing the host part. In particular, interfacing objects in the primary language to raw C/C++ pointers is tedious and error-prone, especially because high-level languages usually have a well-defined type system with type inference.

In this paper, we introduce the `GPUAPI` module, which offers multiple abstraction levels of low-level GPU API routines for high-level programming models with a special focus on PGAS languages, which allows the user to choose an appropriate abstraction level depending on their tuning scenarios. The module is also designed to work with multiple standard low-level GPU programming models: CUDA, HIP, DPC++, and SYCL, thereby significantly improving productivity and portability.

We use Chapel as the primary example and our preliminary performance and productivity evaluations show that the use of the GPUAPI module significantly simplifies GPU programming in a high-level programming model like Chapel, while targeting different multi-node CPUs+GPUs platforms with no performance loss.

Keywords: GPUs · Chapel · PGAS languages · Distributed programming model · GPU API library

H. Anzt et al. (Eds.): ISC High Performance 2022 Workshops, LNCS 13387, pp. 90–107, 2022.
https://doi.org/10.1007/978-3-031-23220-6_7

1 Introduction

There has been a growing interest in accelerators, especially GPU accelerators, in large-scale systems. In the Top 500 list, 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 [2,5,15] including Chapel, is a scalable and portable way to achieve high-performance without sacrificing productivity.

As for GPU support in PGAS languages, some of the past approaches [6, 13] aim at compiling high-level parallel constructs (e.g., Chapel's `forall`) to GPUs. Also, from Chapel 1.24 onwards, a preliminary full automatic approach is available [4,12]. However, in general, there is still a big performance gap between compiler-generated GPU code and hand-tuned GPU code. Thus, it is possible that the user ends up writing a low-level 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, and call it from their primary language.

Our key observation is that there are only two ultimate GPU programming approaches in PGAS languages: fully automatic and fully manual, and there is no "intermediate" approach. Also, our another key observation is that the complexity of the fully manual approach comes not only from writing GPU kernels in the device part, but also from writing the host part. In particular, interfacing objects in the primary language to raw C/C++ pointers is tedious and error-prone, especially because PGAS languages have a well-defined type system with type inference.

In this paper, we propose the `GPUAPI` module, which offers "medium-level (MID-level)" abstraction of low-level GPU API routines for high-level programming models with a special focus on PGAS languages, which fills the gap between the fully automatic approach (we call it HIGH-level) and fully manual approach (we call it LOW-level). In our design, MID-level includes two sub-levels:

- **MID-level:** Provides GPU API that is more natural to the user of the primary language -i.e., use the `new` keyword to allocate GPU memory.
- **MID-LOW-level:** Provides simple wrapper functions for raw GPU API functions -i.e., use the `Malloc` function to allocate GPU memory.

This multi-level design allows the user to choose an appropriate one depending on their tuning scenarios. Specifically, the user has the option of 1) providing a high-level specification (HIGH-level) and letting the compiler do the job, and 2) diving into lower-level details to incrementally evolve their implementations for improved performance (MID-level → MID-LOW-level → LOW-level). Also, the module is designed to work with multiple standard low-level GPU programming models: CUDA, HIP, DPC++, and SYCL, thereby significantly improving productivity and portability.

To the best of our knowledge, this paper is the first paper that discusses the design and implementation of "intermediate-level" GPU API for multiple CPUs+GPUs platforms.

This paper makes the following contributions:

- The design and implementation of multi-level platform-independent GPU API for high-level languages.
- Performance evaluations and productivity discussion using different distributed mini applications and a real-world application [1] on different CPU+GPU systems.

While we use Chapel as the primary language, our discussion should apply to other PGAS languages.

2 Background

2.1 Chapel

Chapel has been one of the most active PGAS languages for decades. 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.

Also, Chapel's "global-view" programming model allows the user to easily write a multi-node program as if they are writing a program for a single-node. For example, suppose `D` is a distributed domain, which is an iteration space that is distributed across multiple nodes, one can write the following code to create a distributed array `A` with the length of `n` and assign 1 to it:

```
1 // D is a block distributed domain, n is a big number
2 var D: domain(1) dmapped Block(boundingBox = {1..n}) = {1..n};
3 var A[D]: int;
4 forall i in D {
5   A[i] = 1;
6 }
```

Space limitations prevent us from including more details on Chapel. For more details, see [3].

2.2 Chapel's `GPUIterator` Module

In our past work [9], 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

Listing 1.1. A Chapel program with the `GPUIterator` module.

```
1 use GPUIterator;
2 proc GPUCallBack(lo: int, hi: int, nElems: int) {
3   // The GPU portion (lo, hi, nElems) is automatically computed
4   // even in multi-node + multi-GPUs settings.
5   // Also, hi-lo+1 == nElems
6   myGPUCode(...);
7 };
8 var CPUpercent = x; // X% goes to the CPU
9                     // (100 - X)% goes to the GPU
10 // D can be a distributed domain
11 forall i in GPU(D, GPUCallBack, CPUPercent) {...}
```

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 1.1 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 the callback. 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-node, multi-GPUs, plus hybrid execution in a portable way. This feature is significant because many of the past approaches that tackle GPU execution in PGAS languages do not support such a feature. To handle multi-GPUs per node, the module automatically computes a subspace for each GPU and implicitly calls the callback function multiple times - i.e., the number of GPUs per node $\times$ the number of nodes. 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 1.2 and Listing 1.3 illustrate an example distributed implementation of the STREAM benchmark (`A = B + alpha*C`) that enables distributed hybrid execution on multple CPUs+GPUs nodes. On line 16 in Listing 1.2, 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 1.3 includes a typical host program including device memory (de)allocation, data transfer, and kernel invocation. Note that the kernel (line 3 in Listing 1.3) is flexible to change in iteration space because it only iterate over `0` to `nElems-1` that is given by the Chapel part. Also, since `localSlice(lo..hi)`[1] returns a pointer to the head of the local slice, it is safe to assume that `&A[0]`,

[1] In Chapel, `lo..hi` means a range starting with `lo` and ending in `hi` (inclusive).

Listing 1.2. 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};
// distributed arrays (A, B, C) with the domain D
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]; }
```

Listing 1.3. 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; size_t nBytes = sizeof(float) * nElems;
  cudaMalloc(&dA, nBytes);
  cudaMalloc(&dB, nBytes);
  cudaMalloc(&dC, nBytes);
  cudaMemcpy(dB, B, nBytes, cudaMemcpyHostToDevice);
  cudaMemcpy(dC, C, nBytes, cudaMemcpyHostToDevice);
  stream<<<ceil(((float)nElems)/)1024, 1024>>>(dA, dB, dC, alpha, nElems);
  cudaDeviceSynchronize();
  cudaMemcpy(A, dA, nBytes, cudaMemcpyDeviceToHost);
  cudaFree(dA);
  cudaFree(dB);
  cudaFree(dC);
}}
```

`&B[0]`, and `&C[0]` in the host part point to `A[lo]`, `B[lo]`, and `C[lo]` in the Chapel part respectively.

For completeness, for the CPU space, it is possible to optimize the CPU part for multiple sub-nodes such as NUMA domains thanks to Chapel. Specifically, the user may let Chapel's tasking runtime map sub-nodes to NUMA domains by doing `export CHPL_LOCALE_MODEL=numa`.

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 1.3, most of the host part includes 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.

As discussed in Sect. 1, our main focus is to develop MID-level/MID-LOW-level explicit GPU API. 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.

Also, related to the point on data transfer optimizations, it is worth noting that, while the calls to our GPU API routines are inside the callback function in the code examples below, this does not necessarily mean that these calls should be placed there. The user has the option of placing these calls outside of the callback function to optimize data transfers.

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/SYCL-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 1.4. An example distributed implementation of STREAM (The MID-LOW version).

```
1  /* steram-mid-low.chpl */
2  use BlockDist; use GPUIterator; use GPUAPI; use CTypes;
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: c_size_t = (lA.size:c_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 or equivalent (HIP, DPC++, ...) */
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, alpha, start, end, nElems);
28 }
```

Listing 1.4 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 pointers, `use CTypes;`[2] 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()`).

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). Note that the user has the option of writing the kernel part in another language (e.g., HIP, DPC++, and so on). For more details, please see Sect. 4. While this MID-LOW-level abstraction simplifies the host code compared to the original host part in Listing 1.3, 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

[2] In Chapel 1.27, 1) `SysCTypes` is replaced with `CTypes`, and 2) `size_t` is replaced with `c_size_t`.

Listing 1.5. 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: c_size_t = D.dim(1).size:c_size_t * c_sizeof(A.eltType);
4 var spitch = widthInBytes;
5 var dA: c_void_ptr;
6 var dpitch: c_size_t;
7 MallocPitch(dA, dpitch, widthInBytes, D.dim(0).size:c_size_t);
8 Memcpy2D(dA, dpitch, c_ptrTo(A), spitch, widthInBytes,
9         D.dim(0).size:c_size_t, 0);
```

Listing 1.6. An example distributed implementation of STREAM. (The MID version)

```
1 use BlockDist; use GPUIterator; use GPUAPI; /* use CTypes; is no longer required */
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, lo, hi, nElems);
10  DeviceSynchronize();
11  dA.fromDevice();
12  // allocate GPU memory automatically deallocated
13 }
```

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 1.5 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 (`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 utilizes Chapel features so the programming style can be more Chapel programmer-friendly. For example, the user can allocate GPU memory using the `new` keyword and no longer need to manipulate C types explicitly.

Listing 1.6 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 11) 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 1.6 with Listing 1.4 and Listing 1.3, 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 Chapel program [10] heavily uses this pattern. Let us discuss our motivation using a simple Chapel program. Consider the Chapel code shown in Listing 1.7. 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.

3.4 Supporting Asynchrony

While the current implementation of the `GPUAPI` module does not directly support asynchronous calls, one can asynchronously invoke GPU-related routines using Chapel's `async` API. Listing 1.8 shows an example of an asynchronous GPU invocation. Line 1 creates a lambda function that performs the boilerplate GPU invocation code with the MID-level API routines. First, the `async` API returns a *future* variable (`F`) immediately after the lambda function is asynchronously spawned. Then, the completion of `F` can be detected by calling `F.get()` (on Line 9). Note that `F.get()` blocks until the returning value is available.

We also plan to directly support asynchronous GPUAPI routines in the future.

Listing 1.7. A jagged array example.

```
class C {
  var n: int;
  proc init(_n: int) { n = _n; }
  var x: [0..#n] int;
}

var Cs = [new C(256), new C(512)];
const N = Cs.size;

// MIDLOW
{
    var dA: [0..#N] c_void_ptr;
    var dAs: c_ptr(c_void_ptr);
    for i in 0..#N {
        const size = Cs[i].x.size:c_size_t*c_sizeof(int);
        Malloc(dA[i], size);
        Memcpy(dA[i], c_ptrTo(Cs[i].x), size, 0);
    }
    const size = N: c_size_t * c_sizeof(c_ptr(c_void_ptr));
    Malloc(dAs, size);
    Memcpy(dAs, c_ptrTo(dA), size, 0);
    // kernel invocation
}
// MID
{
    var dAs = new GPUJaggedArray(Cs.x);
    dAs.toDevice();
    // kernel invocation
}
```

Listing 1.8. An asynchronous GPU invocation example.

```
var F = async(lambda () {
        writeln("GPU Ctrl Thread");
        var dA = new GPUArray(A);
        dA.toDevice();
        kernel(dA.dPtr());
        dA.fromDevice();
        return 1;
    });
if (F.get() == 1) { // F is done }
```

4 Implementation

4.1 Library 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 [11].

In the current implementation, the module mainly supports NVIDIA CUDA-supported GPUs, AMD ROCm-supported GPUs, Intel DPC++ (SYCL) supported GPUs (and FPGAs) through different vendor-provided libraries/frameworks as shown in Fig. 1. One of the interesting aspects of our implementation is that there is only a CUDA implementation of the `GPUAPI` module. We utilize the `hipify` from AMD and `dpct` from Intel to convert the CUDA implementation to a HIP and DPC++ version respectively. Also, for Intel platforms, it is possible to run the `hipify`ed code with `hipLZ` [14]. More specifically, at the time

Table 1. How user-written kernels work on different GPU platforms.

	CUDA	HIP	SYCL
NVIDIA	✓	✓	✓
AMD	✓ (via `hipify`)	✓	✓
Intel	✓ (via `dpct`)	✓ (via `hipLZ`)	✓

of installation, our cmake-based build system identifies installed GPUs and generates an appropriate static (.a) and/or shared (.so) library with the conversion. (Fig. 2).

Because the cmake-generated library (.a and/or .so) includes all of the MID-LOW-level API routines and we provide a cmake file that helps an external cmake project to find this module, it is technically possible to link the MID-LOW-level library from other languages than Chapel. Also, while the MID-level API is tightly-coupled with Chapel, we believe it is feasible to port our module to other PGAS languages.

4.2 The GPU Kernel Part by the User

As we discussed, the user is supposed to write the kernel part using vendor-provided GPU libraries/frameworks such as CUDA, HIP, SYCL, and so on. The user can simply write their kernels using their favorite framework and link it with the corresponding version of GPUAPI library (`libGPUAPICUDA.so`, and so on). If there is any conversion required, the user can also utilize our cmake-based build system. Table 1 summarizes how user-written kernels work on different GPU platforms.

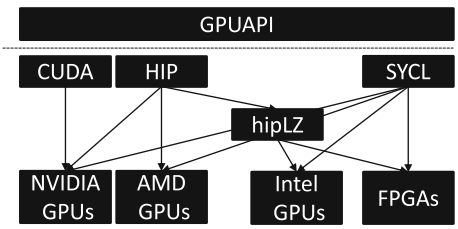


Fig. 1. Multi-platform support in the `GPUAPI` module.

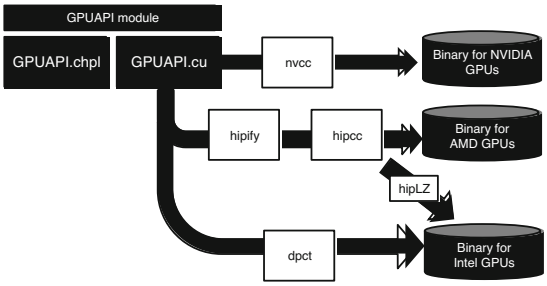


Fig. 2. The implementation of the `GPUAPI` module.

Also, it is also worth noting that this auto-conversion approach works very well even with real-world applications. For example, while the kernel part of the distributed tree search application in Sect. 5 was originally implemented in CUDA, the `hipify` tool was able to produce the HIP version flawlessly. Similarly, in [10], we were able to produce the HIP version of a computational fluid dynamics (CFD) application.

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 and a supercomputer. 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 384 GB and 8 NVIDIA Tesla V100 GPUs, each with 16 GB HBM2 memory, connected via PCIe 3.0[3]. 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 3.45 GHz with a total main memory size of 512 GB and 6 NVIDIA Tesla V100 GPUs, each with 16 GB HBM2 memory, connected via NVLink. The third platform is a single-node AMD server, which consists of 12-core Ryzen9 3900X running at 3.8 GHz and a Radeon RX570 GPU with 8 GB 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), $nFeatures =$

[3] Interconnection network between the GPUs is NVLink.

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 with `CPUPercent=0`.
  - **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 3, 4, and 5 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_SUBSTRAT E=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 these figures, 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.

Table 2 shows source code additions and modifications required for using the GPUAPI. We measure the productivity in term of source lines of code[4]. 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.

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

Table 2. Source code additions and modifications required for using the `GPUAPI` module in terms of source lines of code (SLOC).

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

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

Figure 6a, 6b, and 6c show speedup values relative to the LOW version on a single node of each platform with the 95% confidence intervals. Note that, on the Summit supercomputer, 6 GPUs/node are used without any modifications to the source code thanks to the `GPUIterator` module, while the use of multiple

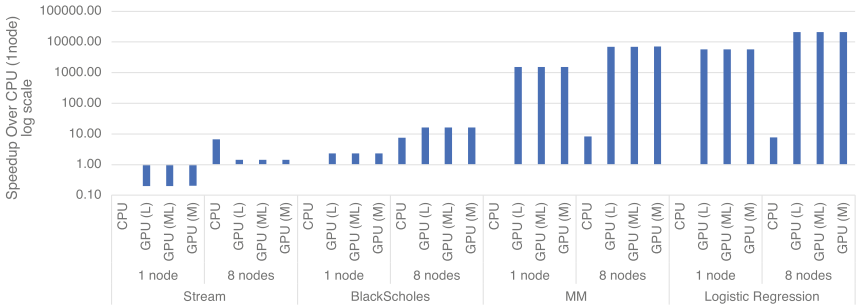


Fig. 3. Performance improvements of mini applications on the Cori GPUs (log scale, multi-nodes: 1GPU/node)

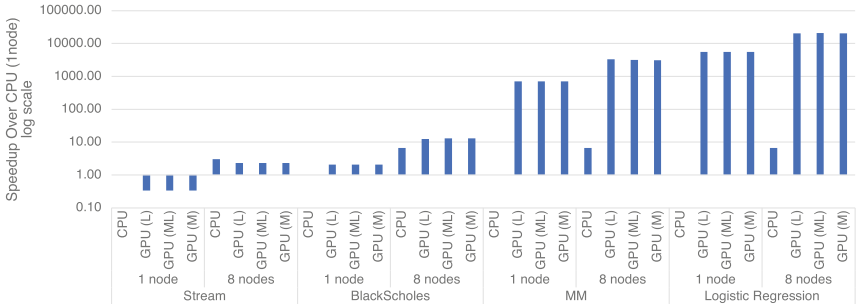


Fig. 4. Performance improvements of mini applications on the Summit supercomputer (log scale, multi-nodes: 1GPU/node)

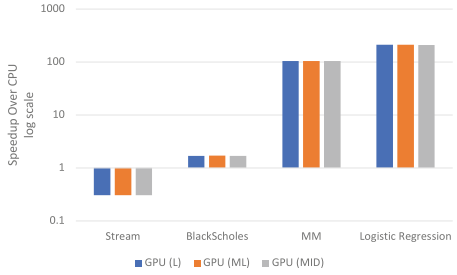
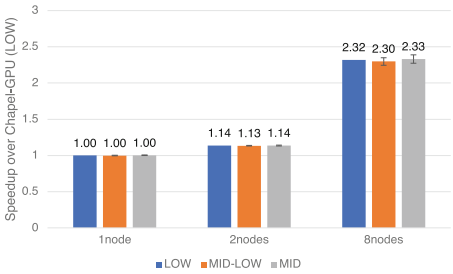


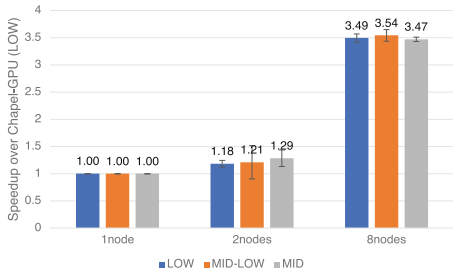
Fig. 5. Performance improvements of mini applications on the AMD server (log scale, single-node:1GPU/node)

GPUs gives an error that is unrelated to our modules on the Cori GPUs. Also, in Fig. 6c, the intervals are not very visible because the numbers are very stable. As with the mini applications discussed in Sect. 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

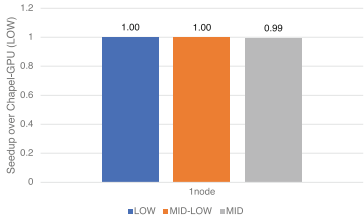
scalability is not as good as that of the mini applications. However, improving the scalability is orthogonal to this work.



(a) Cori (Multi-nodes: 1GPU/node)



(b) Summit (Multi-nodes: 6GPUs/node)



(c) AMD server (Single-node:1GPU/node)

Fig. 6. Performance improvements of the distributed tree search application.

Also, the last row of Table 2 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.

6 Related Work

In the context of compiling PGAS langauges to GPUs, X10CUDA [7] uses the concept of *places* to map a nested parallel loop to blocks and threads on a GPU. It also provides thin wrappers for low-level GPU API rountines, which is analogous to our MID-LOW API.

For Chapel, while Sidelnik et al. [13], Chu et al. [6], and recent versions of Chapel compiler compile Chapel's `forall` constructs to GPUs, it is unfortunate that these approaches are still early and do not support multi-node GPUs or multiple GPUs on a single node. Also, Ghangas [8] compiles a Chapel statement containing multiple arrays GPUs with a single kernel. However, performance results have not been demonstrated yet.

In contrast, our approach is designed to facilitate manual CPU-GPU programming for multi-node platforms with Chapel, while keeping Chapel constructs as much as possible.

7 Conclusions

In this paper, we introduced the `GPUAPI` module, which allows PGAS programmers to have the option of explicitly manipulating device memory (de)allocation API, and data transfer API in their primary language. While it can be used standalone, when it is used with the `GPUIterator` module [9], it significantly facilitates distributed and hybrid execution on multiple CPU+GPU nodes.

We use Chapel as the primary example. Our preliminary performance evaluation using mini-applications and a real-world application is conducted on a wide range of CPU+GPU platforms. The results show that the use of the GPUAPI module significantly simplifies GPU programming in a high-level programming model like Chapel, while targeting different multi-node CPUs+GPUs platforms with no performance loss.

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. Also, the authors would like to thank Josh Milthorpe for his contribution to our code base.

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. Carneiro, T., Melab, N., Hayashi, A., Sarkar, V.: 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 (2021). https://hal.archives-ouvertes.fr/hal-03149394
2. Chamberlain, B.L.: Chapel (cray inc. HPCS language). In: Encyclopedia of Parallel Computing, pp. 249–256 (2011). https://doi.org/10.1007/978-0-387-09766-4_54
3. Chapel: Chapel documentation (2022). https://chapel-lang.org/docs/index.html
4. the Chapel team: GPU programming in chapel documentation (2022). https://chapel-lang.org/docs/latest/technotes/gpu.html
5. Charles, P., et al.: X10: An object-oriented approach to non-uniform cluster computing. Acm Sigplan Not. **40**(10), 519–538. OOPSLA 2005. ACM, New York, NY (2005). https://doi.org/10.1145/1094811.1094852
6. Chu, M.L., Aji, A.M., Lowell, D., Hamidouche, K.: GPGPU support in Chapel with the Radeon Open Compute Platform (Extended Abstract). CHIUW 2017 (2017)
7. Cunningham, D., Bordawekar, R., Saraswat, V.: GPU programming in a high level language: compiling X10 to CUDA. In: Proceedings of the 2011 ACM SIGPLAN X10 Workshop. X10 2011. Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/2212736.2212744

8. Ghangas, R., Milthorpe, J.: Chapel on accelerators. In: 2020 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2020, New Orleans, LA, USA, 18–22 May 2020, pp. 679–679. IEEE (2020). https://doi.org/10.1109/IPDPSW50202.2020.00121
9. Hayashi, A., Paul, S.R., Sarkar, V.: GPUIterator: bridging the gap between chapel and GPU platforms. In: Proceedings of the ACM SIGPLAN 6th on Chapel Implementers and Users Workshop, pp. 2–11. CHIUW 2019. Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3329722.3330142
10. Hayashi, A., Paul, S.R., Sarkar, V.: Accelerating CHAMPS on GPUs. CHIUW 2022 (2022). https://chapel-lang.org/CHIUW/2022/Hayashi.pdf
11. Hayashi, A., et al.: GPUIterator and GPUAPI repository. https://github.com/ahayashi/chapel-gpu (2019)
12. Kayraklioglu, E., Stone, A., Iten, D., Nguyen, S., Ferguson, M., Strout, M.: Targeting GPUs Using Chapel's Locality and Parallelism Features. CHIUW 2022 (2022). https://chapel-lang.org/CHIUW/2022/Kayraklioglu.pdf
13. Sidelnik, A., Maleki, S., Chamberlain, B.L., Garzarán, M.J., Padua, D.: Performance portability with the chapel language. In: Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium, pp. 582–594. IPDPS 2012, IEEE Computer Society, Washington, DC, USA (2012). https://doi.org/10.1109/IPDPS.2012.60
14. Zhao, J., et al.: hipLZ repository (2021). https://github.com/jz10/anl-gt-gpu
15. Zheng, Y., et al.: UPC++: a PGAS extension for C++. In: 2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, 19–23 May 2014, pp. 1105–1114. IPDPS 2014 (2014). https://doi.org/10.1109/IPDPS.2014.115