Enabling CHIP-SPV in Chapel GPUAPI module

CHIUW2023, June 2, 2023

Jisheng Zhao (Georgia Tech)
Akihiro Hayashi (Georgia Tech)
Brice Videau (ANL)
Vivek Sarkar (Georgia Tech)
Enabling CHIP-SPV in Chapel GPUAPI module

MOTIVATION
GPU Programming in Chapel: 
no “intermediate” programming model

Highest-level Chapel-GPU Programming

```chapel
forall i in 1..n {
   A(i) = B(i) + alpha * C(i);
}
```

A huge gap!

Lowest-level Chapel-GPU Programming
(C Interoperability only or GPUIterator)

```chapel
// separate C file
__global__ void stream(float *dA, float *dB, float *dC, float alpha, int N) {
   int id = blockIdx.x * blockDim.x + threadIdx.x;
   if (id < N) {
      dA[id] = dB[id] + alpha * dC[id];
   }
}
void GPUST(float *A, float *B, float *C, float alpha, int start, int end, int GPUN) {
   float *dA, *dB, *dC;
   CudaSafeCall(cudaMalloc(&dA, sizeof(float) * GPUN));
   CudaSafeCall(cudaMalloc(&dB, sizeof(float) * GPUN));
   CudaSafeCall(cudaMalloc(&dC, sizeof(float) * GPUN));
   CudaSafeCall(cudaMemcpy(dB, B + start, sizeof(float) * GPUN, cudaMemcpyHostToDevice));
   CudaSafeCall(cudaMemcpy(dC, C + start, sizeof(float) * GPUN, cudaMemcpyHostToDevice));
   stream<<<ceil(((float)GPUN)/1024), 1024>>> (dA, dB, dC, alpha, GPUN);
   CudaSafeCall(cudaDeviceSynchronize());
   CudaSafeCall(cudaMemcpy(A + start, dA, sizeof(float) * GPUN, cudaMemcpyDeviceToHost));
   CudaSafeCall(cudaFree(dA));
   CudaSafeCall(cudaFree(dB));
   CudaSafeCall(cudaFree(dC));
}
```

Research Question:
What is an appropriate and portable programming interface that bridges the "forall" and GPU versions?
Big Picture: A Multi-level Chapel GPU Programming Model

**HIGH-level:**
The compiler compiles `forall` to CUDA, HIP, and OpenCL

**LOW-level:**
The user prepares full GPU programs and invokes them from Chapel (w/ or w/o the `GPUIterator`)

**forall**

**The missing link**

**Chapel programmer friendly GPU APIs:**
MID-level

```plaintext
var dA = new GPUArray(A);
dA.toDevice();
```

**Thin wrappers for low-level GPU APIs:**
MID-LOW-level

```plaintext
Malloc(); Memcpy();
```

**Goal: increase productivity with no performance loss**

**Our proposal**

"GPUIterator: Bridging the gap between Chapel and GPU Platforms" (CHIUW’19)
Contributions

- Why higher-level abstraction of GPU API?
  - For improving productivity
    - Our observation: The complexity in GPU programming comes not only from writing GPU kernels in the device part, but also from writing the host part
    - ✓ Our GPUAPI is designed to simplify the host part
  - For improving portability
    - Our observation: There are different GPU programming models from different vendors
    - ✓ Our GPUAPI is implemented to work on different platforms (NVIDIA, AMD, Intel, ...)

- Contribution (specific to this talk):
  - Enhance support for Intel GPUs by implementing a CHIP-SPV backend in the GPUAPI module
Enabling CHIP-SPV in Chapel GPUAPI module
Summary of the Chapel GPUAPI module

- **Use case:**
  - The user would like to 1) write GPU kernels, or 2) utilize highly-tuned GPU libraries, and would like to stick with Chapel for the other parts (allocation, data transfers)

- **Provides two levels of GPU API**
  - **MID-LOW:** Provides wrapper functions for raw GPU APIs
    - Example: `var ga: c_void_ptr = GPUAPI.Malloc(sizeInBytes);`
  - **MID:** Provides more user-friendly APIs
    - Example: `var ga = new GPUArray(A);`

- **Note**
  - The user is still supposed to write kernels in CUDA/HIP/SYCL (DPC++)
  - The APIs significantly facilitates the orchestration of:
    - Device memory (de)allocation, and host-to-device/device-to-host data transfers,
  - The use of the APIs does not involve any modifications to the Chapel compiler
  - The module can be utilized in real-world Chapel applications such as Champs and ChOp
Example: Distributed execution of STREAM (MID-level w/ GPUIterator)

```plaintext
var D: domain(1) dmapped Block(boundingBox={1..n}) = {1..n};
var A: [D] real(32);  // Chapel's Distributed Array Allocation
var B: [D] real(32);  // (n divided by # of nodes float elements)
var C: [D] real(32);
var GPUCallBack = lambda (lo: int, hi: int, nElems: int) {
    var dA = new GPUArray(A.localSlice(lo..hi));
    var dB = new GPUArray(B.localSlice(lo..hi));
    var dC = new GPUArray(C.localSlice(lo..hi));
    toDevice(dB, dC);
    LaunchST(dA.dPtr(), dB.dPtr(),
             dC.dPtr(), alpha,
             dN: size_t);
    DeviceSynchronize();
    FromDevice(dA);
    Free(dA, dB, dC);
};
forall i in GPU(D, GPUCallBack, CPUPercent) {
    A(i) = B(i) + alpha * C(i);
}
```

The user has the option of writing device functions, device lambdas, or library calls

// separate C file (CUDA w/ device lambda)
void LaunchST(float *dA, float *dB,
              float *dC, float alpha, int N) {
    GPU_FUNCTOR(N, 1024, NULL,
                [=] __device__ (int i) {
                  dA[i] = dB[i] + alpha * dC[i];
                });
}
Multi-platform Support: Module implementation

- Our module supports a wide variety of GPUs
- Our cmake-based build system detects types of GPUs and generate corresponding static and shared libraries
  - libGPUAPIX_static.a and libGPUAPIX.so
  - \( X = \text{CUDA, HIP, or DPCPP} \)
Multi-platform Support: Module implementation

- GPUAPI module
  - GPUAPI.chpl
  - GPUAPI.cu

  nvcc
  - Binary for NVIDIA GPUs

  hipify
  - Binary for AMD GPUs

  hipcc
  - Binary for Intel GPUs

  dpct
  - CHIP-SPV
Multi-platform Support: User-written kernels

- Our multi-platform GPUAPI support allows the user to choose any GPU programming models to write their kernel code

<table>
<thead>
<tr>
<th></th>
<th>CUDA</th>
<th>HIP</th>
<th>SYCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIDIA</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>AMD</td>
<td>Yes (hipify)</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Intel</td>
<td>Yes (dpct or CHIP-SPV)</td>
<td>Yes (CHIP-SPV)</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Enabling CHIP-SPV in Chapel GPUAPI module

CHIP-SPV
CHIP-SPV

- Connect heterogeneous programming models (CUDA/HIP) to low-level Intel Level Zero runtime
- Intel Level Zero: A system level programming interface that bridges high level libraries to Intel devices
CHIP-SPV in the GPUAPI module

CUDA source .cu

Clang/LLVM Compiler

Host LLVM IR

Device LLVM IR

X86 Target

SPIR-V Target

LLVM Linker

X86 ELF

SPIR-V IL

CUDA APIs implemented via Intel Level Zero Runtime APIs

Chapel Compiler

Chapel Host Binary

Chapel Runtime Library

GPUAPI module

GPUAPI chpl

GPUAPI cu

Clang/LLVM

Library

SPIR-V IL

CUDA APIs implemented via Intel Level Zero Runtime APIs

Georgia Tech

Argonne National Laboratory
Enabling CHIP-SPV in Chapel GPUAPI module

PRELIMINARY PERFORMANCE EVALUATIONS
Performance Evaluations

- Platforms (Single-node, integrated-GPUs)
  - Intel Xeon Processor E5-1585 v5 (4-core) + Iris Pro P580 (Gen9)
  - Intel i7-12700 + UHD770

- Applications
  - Micro-benchmark: STREAM, BlackScholes, Matrix Multiplication, Logistic Regression

- Chapel Compilers & Options
  - Chapel Compiler 1.29.0 with the --fast option
  - Used with the GPUIterator (CPU+GPU execution)
Preliminary Performance Numbers (vs. forall execution on CPUs)

Key take aways:
- Our end-to-end compilation flow with CHIP-SPV is verified
- Performance improvements are not significant due to integrated GPUs, but this is where the GPUiterator module comes into play
Enabling CHIP-SPV in Chapel GPUAPI module
Conclusions

- Intel GPUs support in the GPUAPI module
  - Verified with microbenchmarks on two Intel GPU platforms
- Future work:
  - Use discrete Intel GPUs
  - Discuss the possibility of using CHIP-SPV in the current Chapel compiler

**HIGH-level:** The compiler compiles `forall` to CUDA, HIP, and OpenCL

**LOW-level:** The user prepares full GPU programs and invokes them from Chapel (w/ or w/o the GPUiterator)

**Our proposal**

Chapel programmer friendly GPU APIs:
- MID-level
  - `var dA = new GPUArray(A);` `dA.toDevice();`

Thin wrappers for low-level GPU APIs:
- MID-LOW-level
  - `Malloc(); Memcpy();`

**Goal:** Increase productivity with no performance loss
Join our community

- GPUAPI+GPUIterator:
  - The repository
    - [https://github.com/ahayashi/chapel-gpu](https://github.com/ahayashi/chapel-gpu)
  - Detailed Documents
    - [https://ahayashi.github.io/chapel-gpu/index.html](https://ahayashi.github.io/chapel-gpu/index.html)

- Our community is growing!

Contact: ahayashi “at” gatech.edu
Acknowledgements

- This work was supported by the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357, and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration). We also gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory.
Enabling CHIP-SPV in Chapel GPUAPI module
Chapel GPU API Design: MID-LOW GPU API

- **Summary**
  - Provides the same functionality as CUDA/HIP/OpenCL
  - The user is still supposed to write CUDA/HIP/OpenCL kernels
  - The user is supposed to handle both C types and Chapel types

- **Key APIs**
  - Device Memory Allocation
    - `Malloc(…);`
    - `MallocPitch(…);`
  - Host-to-device, and device-to-host data transfers
    - `Memcpy(…);`
    - `Memcpy2D(…);`
  - Ensuring the completion of GPU computations
    - `DeviceSynchronize();`
  - Device Memory deallocation
    - `Free(…);`
Chapel GPU API Design: MID GPU API

- Summary
  - More natural to Chapel programmers
  - The user is still supposed to write CUDA/HIP/OpenCL kernels

- Key APIs
  - Device Memory Allocation
    - `var dA = new GPUArray(A);`
    - `var dA = new GPUJaggedArray(A);`
  - Host-to-device, and device-to-host data transfers
    - `ToDevice(dA:GPUArray, ...); FromDevice(dA: GPUArray, ...);`
    - `dAToDevice(); dA.fromDevice();`
  - Implicit Device Memory deallocation
    - Automatically “freed” when a GPUArray/GPUJaggedArray object is deleted
  - Explicit Device Memory deallocation
    - `delete`
Chapel GPU API Design: MID-LOW/MID GPU API Example

**MID-LOW Level**

```chapel
1 use GPUAPI;
2 var A: [1..n] real(32);  }
3 var B: [1..n] real(32);  }
4 var C: [1..n] real(32);  }
5 var dA, dB, dC: c_void_ptr;
6 var size: size_t =
7   (A.size:size_t * c_sizeof(A.eltType));
8 Malloc(dA, size);
9 Malloc(dB, size);
10 Malloc(dC, size);
11 Memcpy(dB, c_ptrTo(B), size, TODEVICE);
12 Memcpy(dC, c_ptrTo(C), size, TODEVICE);
13 LaunchST(dA, dB, dC, alpha, N: size_t);
14 DeviceSynchronize();
15 Memcpy(c_ptrTo(A), dA, size, FROMDEVICE);
16 Free(dA); Free(dB); Free(dC);
```

**MID-level**

```chapel
1 use GPUAPI;
2 var A: [1..n] real(32);  }
3 var B: [1..n] real(32);  }
4 var C: [1..n] real(32);  }
5 var dA = new GPUArray(A);
6 var dB = new GPUArray(B);
7 var dC = new GPUArray(C);
8 toDevice(dB, dC);
9 LaunchST(dA.dPtr(), dB.dPtr(),
10    dC.dPtr(), alpha,
11    dN: size_t);
12 DeviceSynchronize();
13 FromDevice(dA);
14 Free(dA, dB, dC);
```

For more details, please see [https://ahayashi.github.io/chapel-gpu/index.html](https://ahayashi.github.io/chapel-gpu/index.html)
Our GPUIterator module facilitates running GPUAPI programs across CPUs+GPUs

1 forall i in 1..n {
2   A(i) = B(i) + alpha * C(i);
3 }

1 forall i in GPU(1..n, GPUCallBack, CPUPercent) {
2   A(i) = B(i) + alpha * C(i);
3}

GPUCallBack is a function that includes a sequence of GPUAPI (next slide).

CPUPercent

GPUPercent = 100 - CPUPercent