Exploring a multi-resolution GPU programming model for Chapel

Akihiro Hayashi (Georgia Tech)

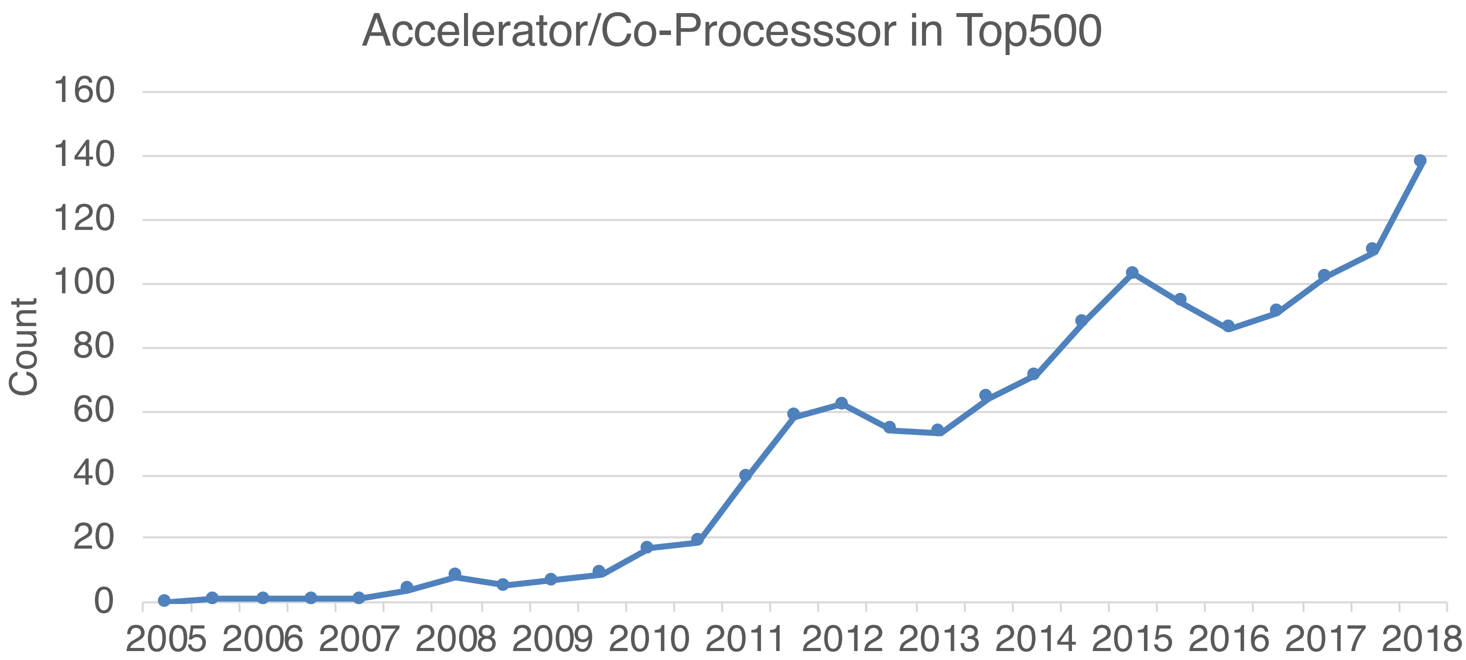
Sri Raj Paul (Georgia Tech)

Vivek Sarkar (Georgia Tech)

A multi-resolution GPU programming model for Chapel

MOTIVATION

GPUs are a common source of performance improvement in HPC



Aurora (ANL)
El Capitan(LLNL)
Frontier (ORNL)
all plan
to include GPUs!

2021

Source: https://www.top500.org/statistics/list/

Georgia Tech

GPU Programming in Chapel

- Chapel's multi-resolution concept

High-level

  - Start with writing "forall" loops
    (on CPU, proof-of-concept)

```
forall i in 1..n {
  …
}
```

  - Apply automatic GPU code generators [1][2] when/where possible
  - Consider using the GPUIterator module [3] (Mason-registry)

Low-level

  - Consider writing GPU kernels using CUDA/HIP/OpenCL or other accelerator language, and invoke them from Chapel

Georgia Tech

[1] Albert Sidelnik et al. Performance Portability with the Chapel Language (IPDPS '12).
[2] Michael L. Chu et al. GPGPU support in Chapel with the Radeon Open Compute Platform (CHIUW'17).
[3] Akihiro Hayashi et al. GPUIterator: Bridging the gap between Chapel and GPU Platforms (CHIUW'19).

Example:
STREAM (Original)

```
var A: [1..n] real(32);
var B: [1..n] real(32);
var C: [1..n] real(32);
// STREAM
forall i in 1..n {
  A(i) = B(i) + alpha * C(i);
}
```

Georgia Tech

Example: STREAM (C interoperability)

- Invoking CUDA/HIP/OpenCL code using the C interoperability feature

```
extern proc GPUST(A: [] real(32),
                  B: [] real(32),
                  C: [] real(32),
                  al: real(32),
                  lo: int, hi: int);
var A: [1..n] real(32);
var B: [1..n] real(32);
var C: [1..n] real(32);
// Invoking CUDA/OpenCL program
GPUST(A, B, C, alpha, 1, n);
```

```
// separate C file
void GPUST(float *A,
           float *B,
           float *C,
           float al,
           int start,
           int end) {
  // CUDA/HIP/
     OpenCL Code
}
```

Georgia Tech

Example: STREAM (GPUIterator)

- Connecting the GPU version with the forall loop using the GPUIterator module

```
// GPU Iterator (in-between)
var G = lambda (lo: int, hi: int,
                nElems: int) {
  GPUST(A, B, C, alpha, 1, n);
};
var CPUPercent = 50;
forall i in GPU(1..n, G, CPUPercent) {
  A(i) = B(i) + alpha * C(i);
}
```

```
// separate C file
void GPUST(float *A,
           float *B,
           float *C,
           float al,
           int start,
           int end) {
  // CUDA/HIP/
     OpenCL Code
}
```

Note: the GPUIterator is designed to facilitate 1) hybrid execution (CPUs+GPUs), and 2) distributed execution

Georgia Tech

Example:
No appropriate GPU abstraction

Highest-level Chapel-GPU Programming

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

A huge gap!

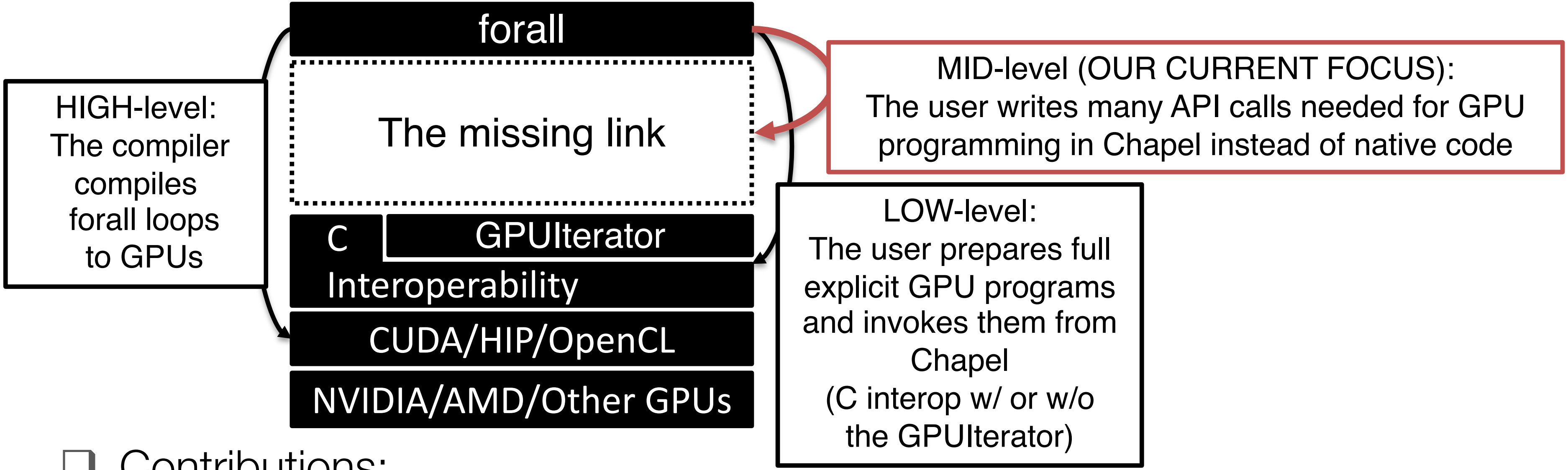
Lowest-level Chapel-GPU Programming (C Interoperability only or GPUIterator)

```
// 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

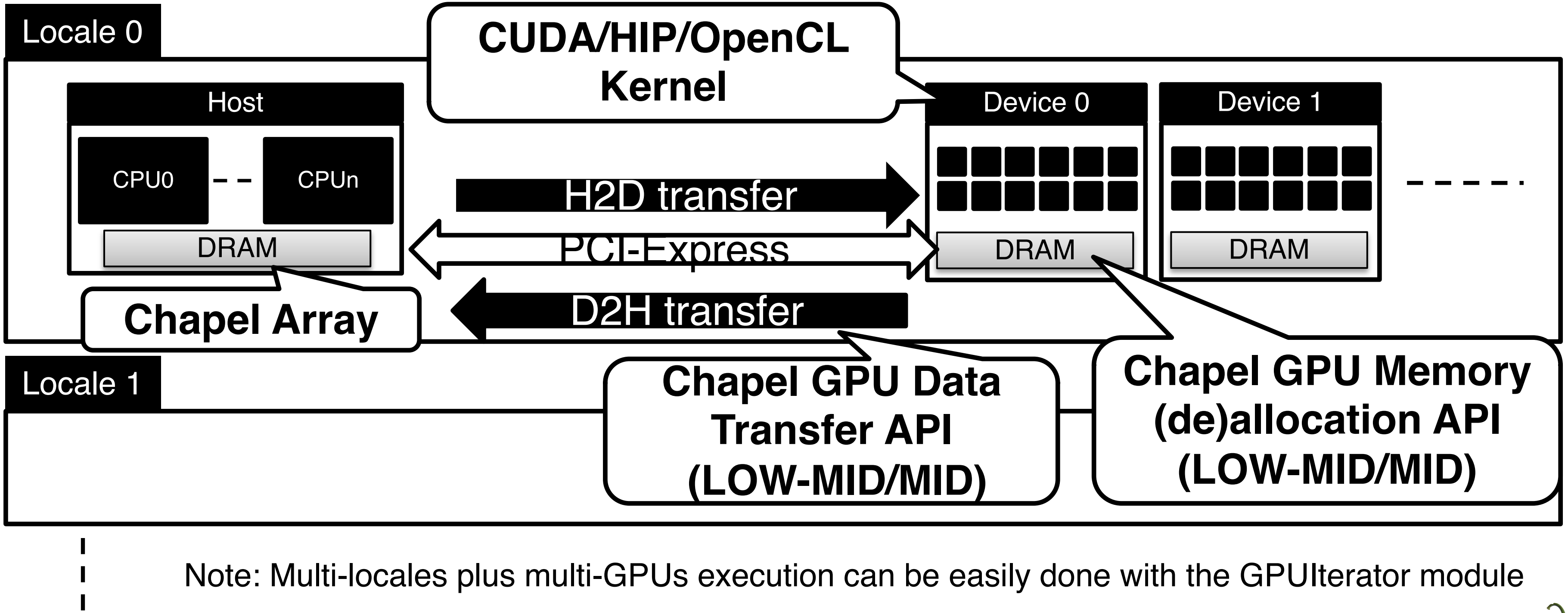


- Contributions:
  - Design and implementation of “MID”/” LOW-MID” levels Chapel GPU API
  - Performance evaluations on different distributed and single-node platforms (Summit, Cori, and a single-node AMD machine)

A multi-resolution GPU programming model for Chapel

DESIGN

An overview of the LOW-MID/MID GPU API for Chapel



Note: Multi-locales plus multi-GPUs execution can be easily done with the GPUIterator module

Chapel GPU API Design: Summary

- 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
  - LOW-MID: 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);`
- Important design decisions
  - The user is still supposed to write kernels in CUDA/HIP/OpenCL
  - 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

Georgia Tech

Chapel GPU API Design: LOW-MID 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(ref devPtr: c_void_ptr, size: size_t);`
  - Host-to-device, and device-to-host data transfers
    - ✓ `Memcpy(dst: c_void_ptr, src: c_void_ptr, count: size_t, direction: int);`
  - Ensuring the completion of GPU computations
    - ✓ `DeviceSynchronize(void);`
  - Device Memory deallocation
    - ✓ `Free(c_void_ptr);`

Georgia Tech

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);`
  - Host-to-device, and device-to-host data transfers
    - ✓ `ToDevice(dA:GPUArray, …); FromDevice(dA: GPUArray, …);`
    - ✓ `dA.ToDevice(); dA.fromDevice();`
  - Device Memory deallocation
    - ✓ `Free(dA:GPUArray, …);`
    - ✓ `dA.Free();`

Georgia Tech

Chapel GPU API Design: LOW-MID/MID GPU API Example

LOW-MID Level

```
use GPUAPI;
var A: [1..n] real(32);
var B: [1..n] real(32);
var C: [1..n] real(32);
var dA, dB, dC: c_void_ptr;
var size: size_t =
   (A.size:size_t * c_sizeof(A.eltType));
Malloc(dA, size);
Malloc(dB, size);
Malloc(dC, size);
Memcpy(dB, c_ptrTo(B), size, TODEVICE);
Memcpy(dC, c_ptrTo(C), size, TODEVICE);
LaunchST(dA, dB, dC, alpha, N: size_t);
DeviceSynchronize();
Memcpy(c_ptrTo(A), dA, size, FROMDEVICE);
Free(dA); Free(dB); Free(dC);
```

MID-level

```
use GPUAPI;
var A: [1..n] real(32);
var B: [1..n] real(32);
var C: [1..n] real(32);
var dA = new GPUArray(A);
var dB = new GPUArray(B);
var dC = new GPUArray(C);
toDevice(dB, dC);
LaunchST(dA.dPtr(), dB.dPtr(),
         dC.dPtr(), alpha,
         dN: size_t);
DeviceSynchronize();
FromDevice(dA);
Free(dA, dB, dC);
```

Georgia Tech

Example: Single-node execution of STREAM (MID-level w/ GPUIterator)

```
var A: [1..n] real(32);
var B: [1..n] real(32);
var C: [1..n] real(32);
var GPUCallBack = lambda (lo: int, hi: int, nElems: int) {
  var dA = new GPUArray(A);
  var dB = new GPUArray(B);
  var dC = new GPUArray(C);
  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(1..n, GPUCallBack,
                      CPUPercent) {
  A(i) = B(i) + alpha * C(i);
}
```

The user has the option of writing a device function(s), a device lambda(s), or a library call(s)

```
// 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];
    });
}
```

Example: Distributed execution of STREAM (MID-level w/ GPUIterator)

```
var D: domain(1) dmapped Block(boundingBox={1..n}) = {1..n};
var A: [D] real(32);
var B: [D] real(32);
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 a device function(s), a device lambda(s), or a library call(s)

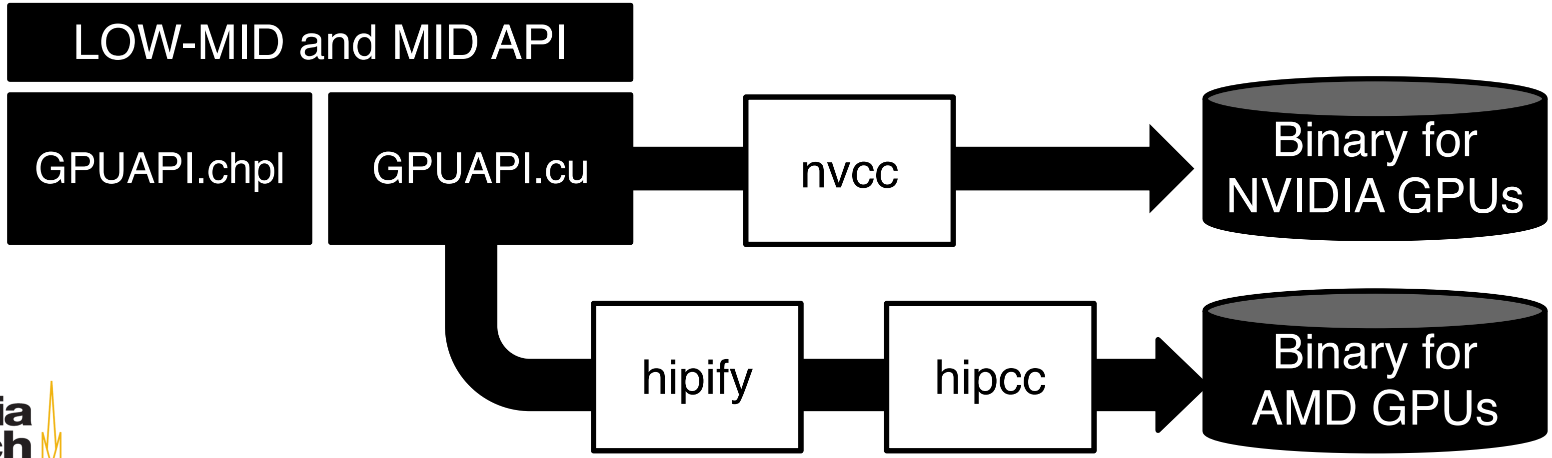
```
// separate C file (CUDA w/ device lambda)
void LaunchST(float *dA, float *dB,
              float *dC, float alpha, int N) {
  GPU_FUNCTOR(N, 1024, NULL,
    [=] __device__ (int i) {
      dA[i] = dB[i] + alpha * dC[i];
    });
}
```

A multi-resolution GPU programming model for Chapel

IMPLEMENTATION

Implementation of GPU API

- Provides an external module (GPUAPI)
  - Can be used either stand-alone or with the GPUIterator module
  - https://github.com/ahayashi/chapel-gpu
    - ✓ The “feature/explicit” branch
- Currently supports NVIDIA and ROCM-ready AMD GPUs



A multi-resolution GPU programming model for Chapel

PERFORMANCE EVALUATIONS

Performance Evaluations

- ❑ Platforms
  - Cori GPU@NERSC: Intel Xeon (Skylake) + NVIDIA V100 GPU
  - Summit@ORNL: IBM POWER9 + NVIDIA Tesla V100 GPU
  - A single-node AMD machine: Ryzen9 3900 + Radeon RX570
- ❑ Chapel Compilers & Options
  - Chapel Compiler 1.20.0 with the --fast option
- ❑ GPU Compilers
  - CUDA: NVCC 10.2 (Cori), 10.1 (Summit) with the -O3 option
  - AMD: ROCM 2.9.6, HIPCC 2.8 with the -O3 option

Performance Evaluations (Cont'd)

- Tasking & Multi-locale execution
  - CHPL_TASK=qthreads
  - CHPL_COMM=gasnet
  - CHPL_COMM_SUBSTRATE=ibv
- GPUIterator (For distributed GPU execution)
  - https://github.com/ahayashi/chapel-gpu/tree/feature/explicit
- Applications
  - Stream
  - BlackScholes
  - Matrix Multiplicaiton
  - Logistic Regression
  - Source code can be found at: https://github.com/ahayashi/chapel-gpu/tree/feature/explicit/apps

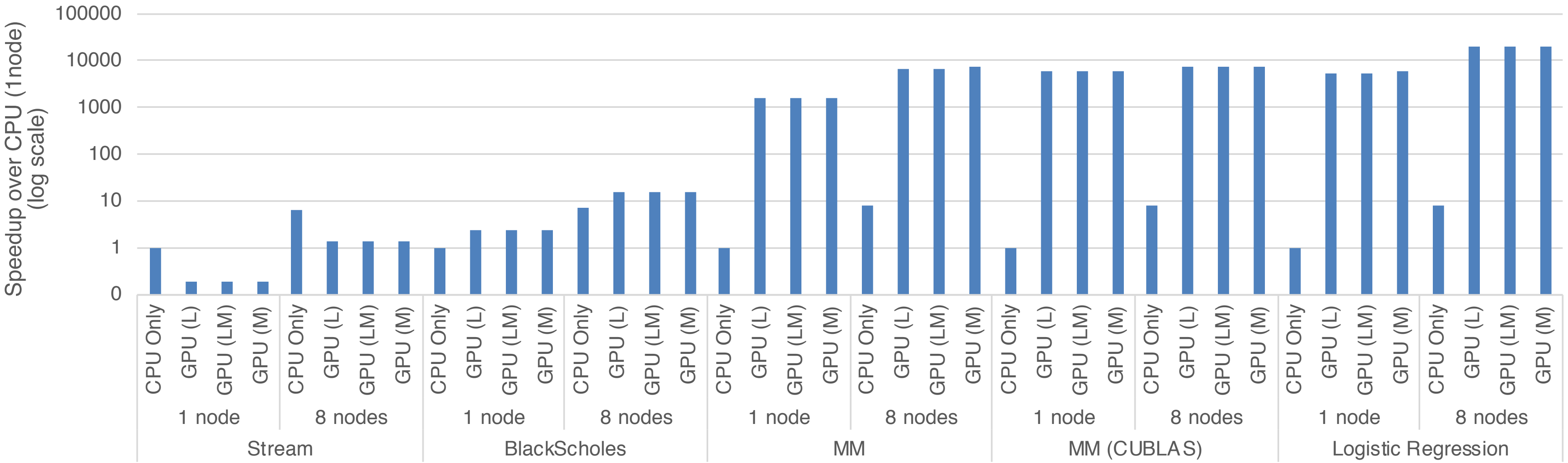
How many lines are added/modified?

	LOC Baseline (Chapel LOW + CUDA)	LOC LOW-MID (Chapel LOW-MID + CUDA)	LOC MID (Chapel MID + CUDA)
Stream	4 + 24 = 28	16 + 9 = 25	11 + 9 = 20
BlackScholes	4 + 99 = 103	16 + 83 = 99	11 + 83 = 94
Logistic Regression	2 + 36 = 38	16 + 18 = 34	10 + 18 = 28
Matrix Multiplication	3 + 30 = 33	14 + 15 = 29	10 + 16 = 26

- ☐ The use of GPU API decreases LOC

Georgia Tech

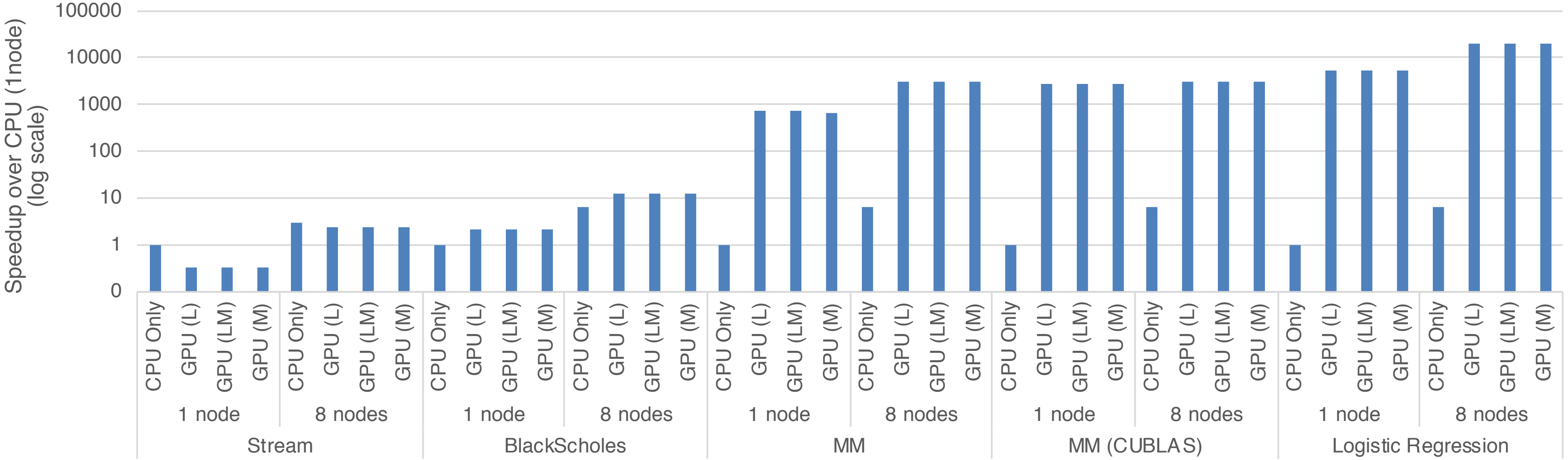
How fast are GPUs? (Multi-nodes, 1GPU/node, Cori)



- Each app shows good strong scalability
- No significant performance difference between L (LOW), LM (LOW-MID), and M (MID)

Data Size: n = 2^30 (Stream, BlackScholes)
4096x4096 (MM) nFeatures = 2^18, nSamples = 2^4

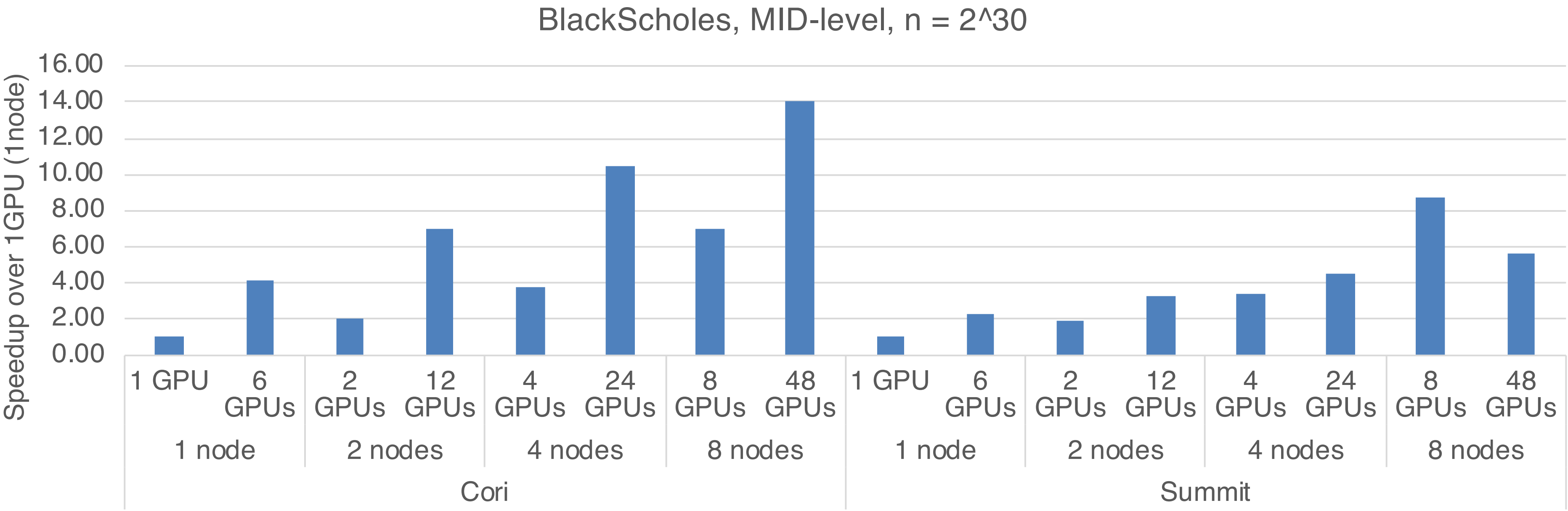
How fast are GPUs? (Multi-nodes, 1GPU/node, Summit)



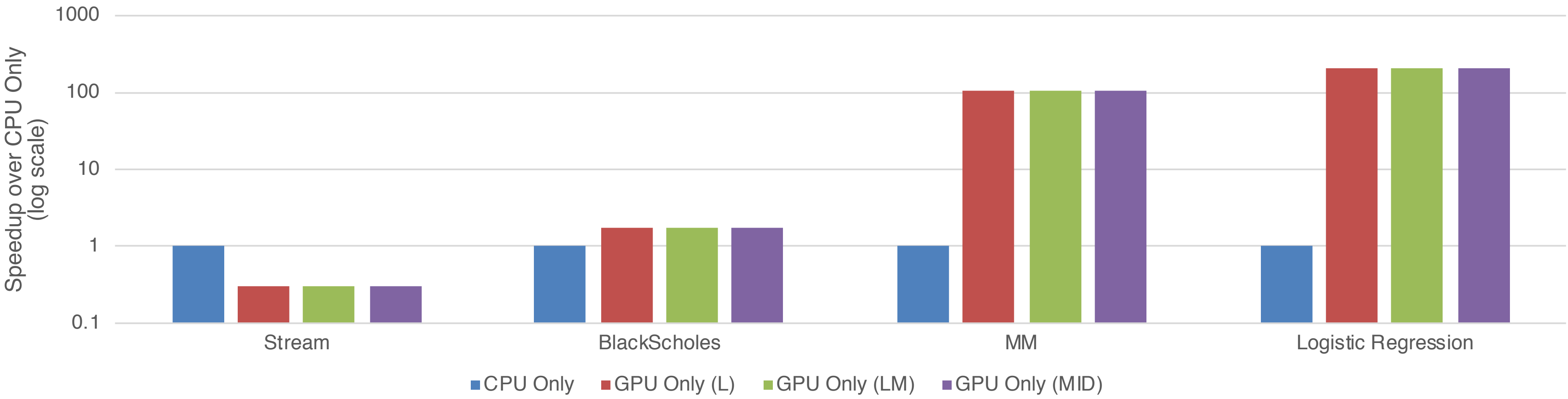
- ❑ Each app shows good strong scalability
- ❑ No significant performance difference between L (LOW), LM (LOW-MID), and M (MID)

Data Size: n = 2^30 (Stream, BlackScholes)
4096x4096 (MM) nFeatures = 2^18, nSamples = 2^4

How fast are GPUs? (Multi-nodes, 6GPUs/node, Cori, Summit)



How fast are GPUs? (A single-node, Ryzen9 + Radeon RX570)



- ❑ The GPUAPI works on an AMD GPU machine

A multi-resolution GPU programming model for Chapel

CONCLUSIONS & FUTURE WORK

Conclusions

- Summary
  - The GPUAPI provides an appropriate interface between Chapel and accelerator programs
    - ✓ Souce code is available (the feature/explicit branch):
      - https://github.com/ahayashi/chapel-gpu
  - The LOW-MID and MID level GPU API enable a higher-level GPU programming interface w/ minimal performance overhead
    - ✓ Verified on NVIDIA and AMD GPUs
  - The GPUAPI + the GPUIterator module facilitate distributed GPU programming using Chapel
    - ✓ Verified on Summit and Cori
    - ✓ The use of GPUs can significantly improve the performance of Chapel programs

Future Work

- Building a higher-level programming model
  - Built on top of the MID-level API
  - forall + “intents”

```
forall i in D with (in: B, out: A) {…}
=>
var dA = GPUArray(A); var dB = GPUArray(B); toDevice(B);
kernel(); fromDevice(A);
```

  - For the kernel code generation, will explore the possibility of avoiding/minimizing compiler modifications
    - ✓ e.g., generate a C/C++ loop + OpenMP/OpenACC/Other pragma
      - Chapel’s Vectorizing Iterator does a similar thing (vectorizeOnly())
- Asynchronous GPU API + Futures

Future work (Cont'd)

- Wish List: lambda + capture by reference

Current MID-level

```
use GPUAPI;
var A: [1..n] real(32);
var B: [1..n] real(32);
var dA = new GPUArray(A);
var dB = new GPUArray(B);
toDevice(dB);
LaunchVC(dA.dPtr(),
         dB.dPtr(), N: size_t);
DeviceSynchronize();
FromDevice(dA);
Free(dA, dB);
```

If dA, dB, N can be captured…

```
use GPUAPI;
var A: [1..n] real(32);
var B: [1..n] real(32);
var dA = new GPUArray(A);
var dB = new GPUArray(B);
var in = (dB); // tuple
var out = (dA); // tuple
Launch (in, out, lambda () {
 LaunchVC(dA.dPtr(),
          dB.dPtr(), N: size_t);
});
```

Thank you for your attention!

Any questions?

Backup Slides

Chapel’s iterator

- Chapel’s iterator allows us to control over the scheduling of the loops in a productive manner

```
// Iterator over fibonacci numbers
forall i in fib(10) {
  A(i) = B(i);
}
```

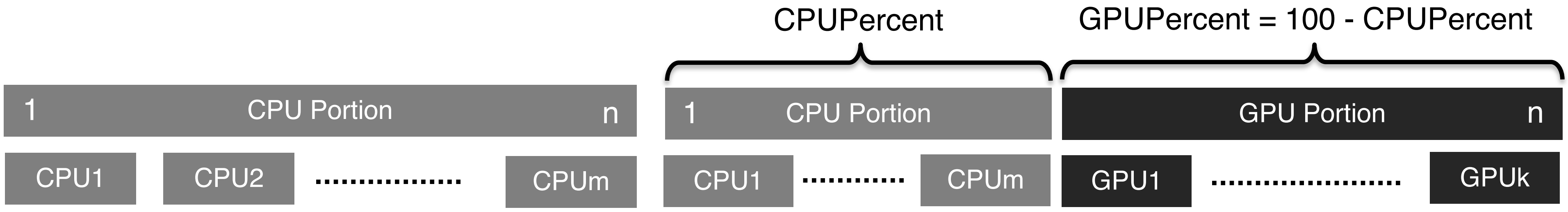
CPU1					CPU2				
0	1	1	2	3	5	8	13	21	34

https://chapel-lang.org/docs/master/primers/parIters.html

The GPUIterator automates work distribution across CPUs+GPUs

```
forall i in 1..n {
  A(i) = B(i);
}
```

```
forall i in GPU(1..n, GPUWrapper,
                CPUPercent) {
  A(i) = B(i);
}
```



How to use the GPUIterator?

```
var GPUCallBack = lambda (lo: int,
                          hi: int,
                      nElems: int){
  assert(hi-lo+1 == nElems);
  GPUVC(A, B, lo, hi);
};
forall i in GPU(1..n, GPUCallBack,
                      CPUPercent) {
  A(i) = B(i);
}
```

This callback function is called after the GPUIterator has computed the subspace (lo/hi: lower/upper bound, n: # of elements)

GPU() internally divides the original iteration space for CPUs and GPUs

Georgia Tech

Tip: declaring CPUPercent as a command-line override ("config const") helps us to explore different CPU+GPU executions

The GPUIterator supports Zippered-forall

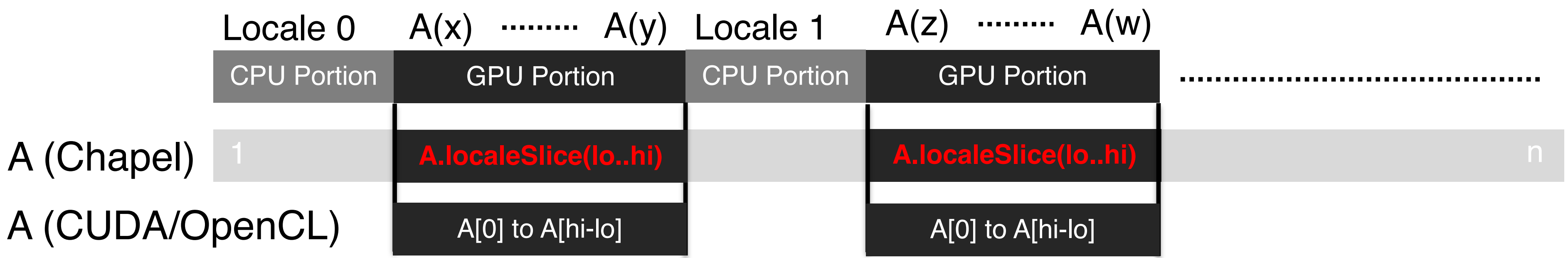
```
forall (_, a, b) in zip(GPU(1..n, ...), A, B) {
  a = b;
}
```

- Restriction
  - The GPUIterator must be the leader iterator

Bradford L. Chamberlain et al. “User-Defined Parallel Zippered Iterators in Chapel.” (PGAS2011)

The GPUIterator supports Distributed Arrays (Cont'd)

- No additional modifications for supporting multi-locales executions

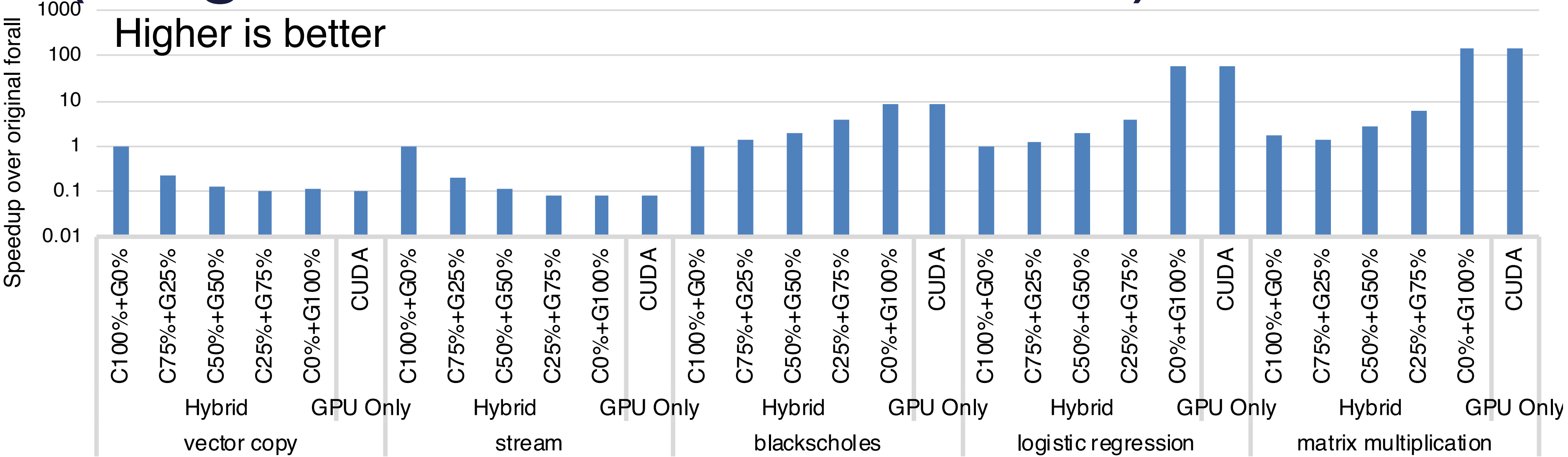


Note: localeSlice is Chapel's array API

Implementation of the GPUIterator

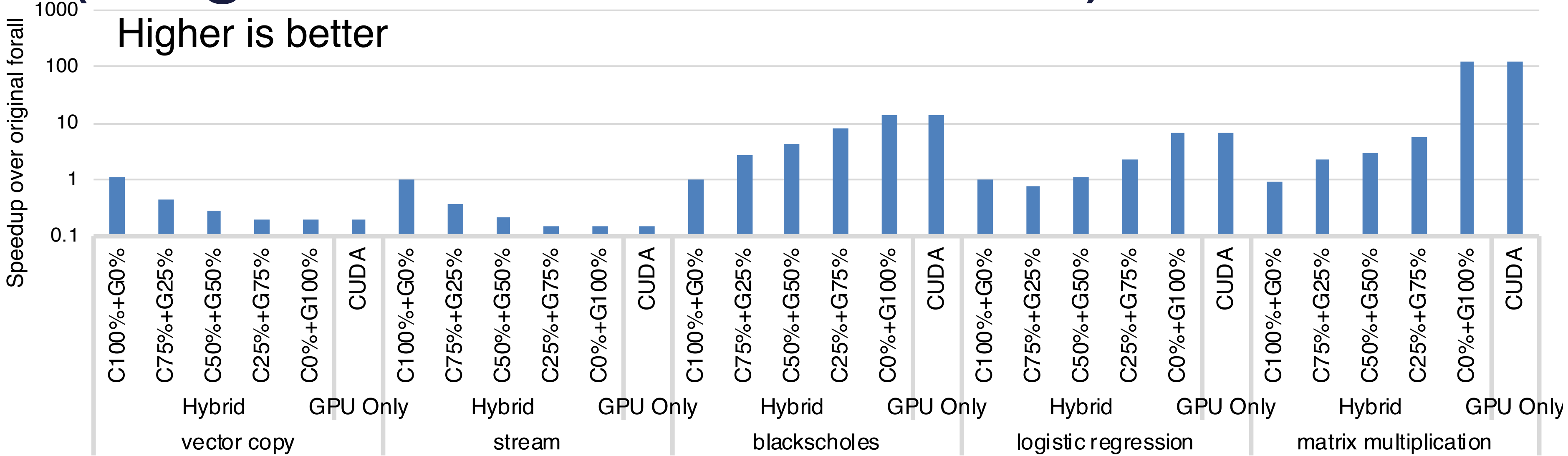
```
coforall subloc in 0..1 {
  if (subloc == 0) {
     const numTasks = here.getChild(0).maxTaskPar;
     coforall tid in 0..#numTasks {
       const myIters = computeChunk(…);
       for i in myIters do
         yield i;
     }
  } else if (subloc == 1) {
     GPUCallBack(…);
  }
}
```

How fast are GPUs? (Single-node, POWER8 + K80)



- The iterator enables exploring different CPU+GPU strategies with very low overheads
- The GPU is up to 145x faster than the CPU, but is slower than the GPU due to data transfer costs in some cases

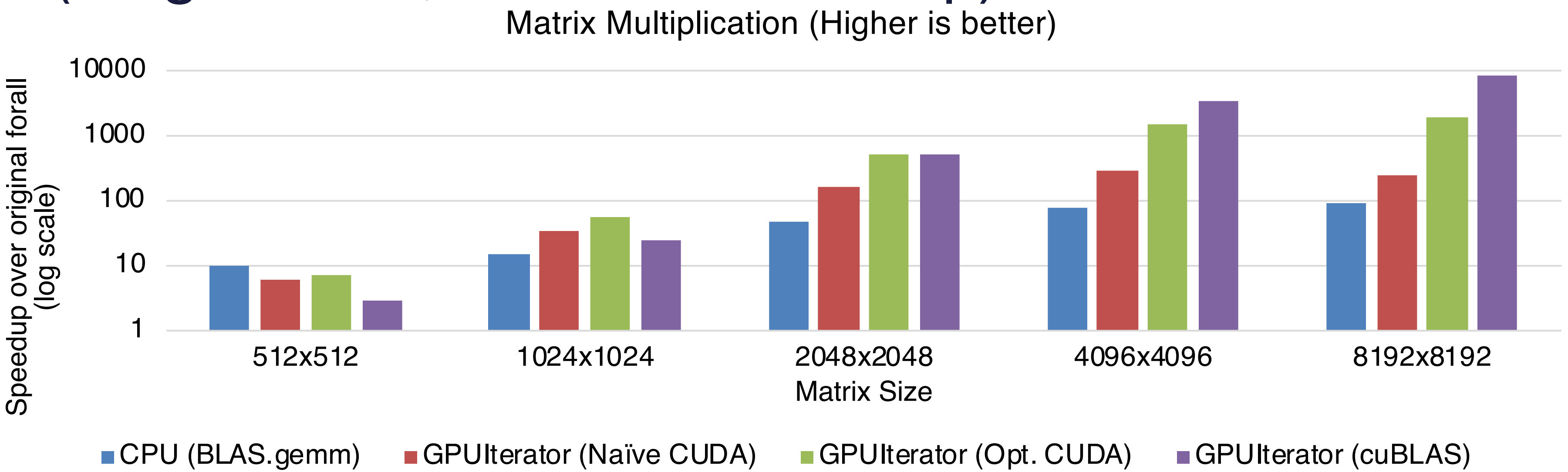
How fast are GPUs? (Single-node, Xeon + M2050)



- The iterator enables exploring different CPU+GPU strategies with very low overheads
- The GPU is up to 126x faster than the CPU, but is slower than the GPU due to data transfer costs in some cases

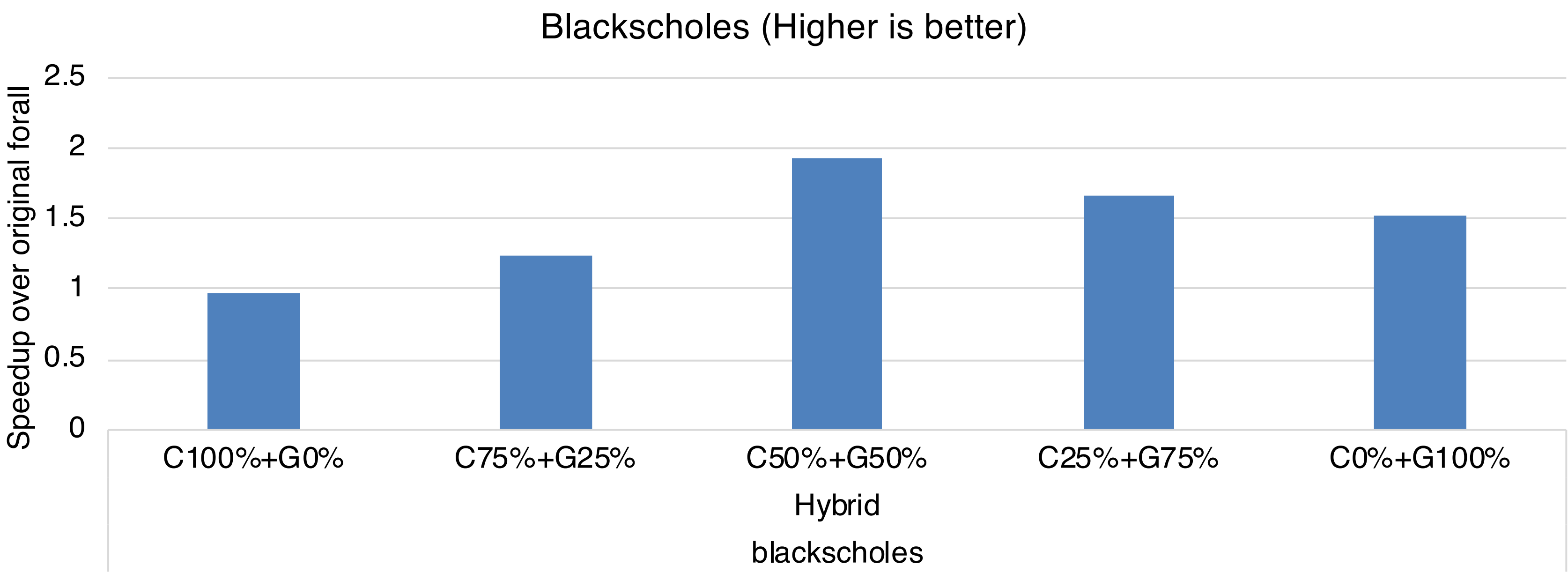
Georgia Tech

How fast are GPUs compared to Chapel's BLAS module on CPUs? (Single-node, Core i5 + Titan Xp)



- Motivation: to verify how fast the GPU variants are compared to a highly-tuned Chapel-CPU variant
- Result: the GPU variants are mostly faster than OpenBLAS's gemm (4 core CPUs)

When is hybrid execution beneficial? (Single node, Core i7+UHD)



- ❑ With tightly-coupled GPUs, hybrid execution is more beneficial

Georgia Tech