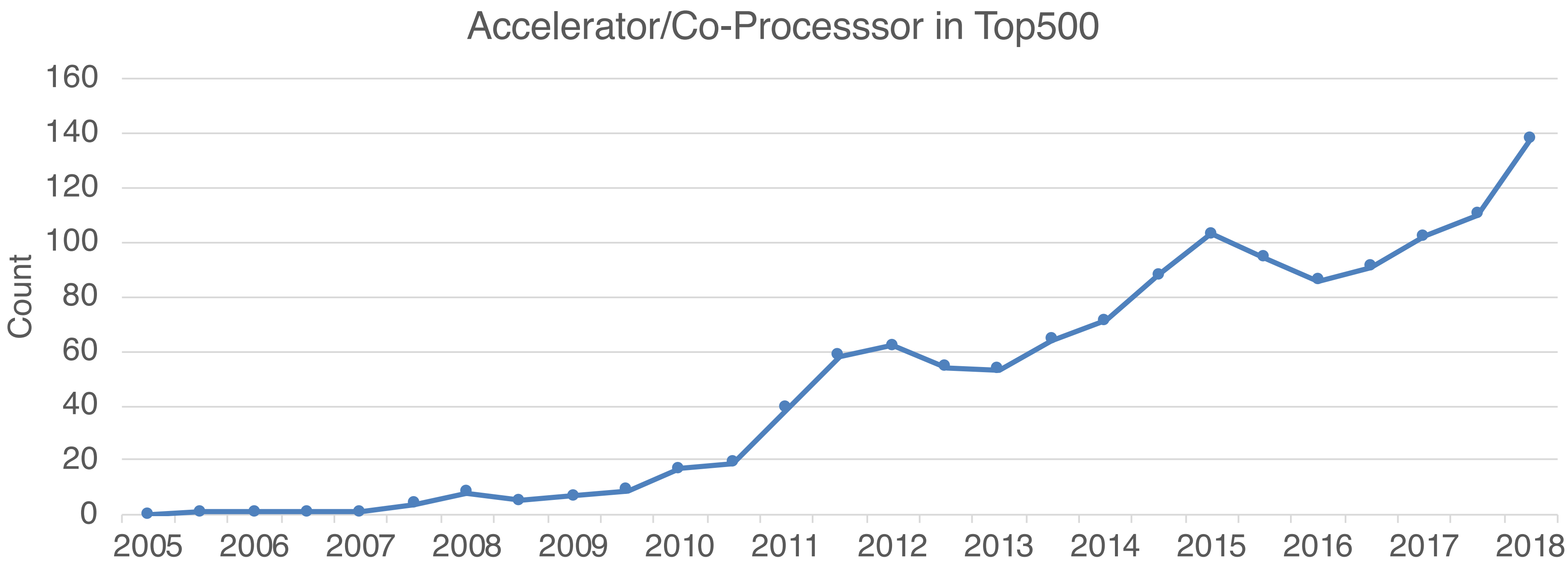
GPUIterator: Bridging the Gap between Chapel and GPU platforms

Akihiro Hayashi (Rice),
Sri Raj Paul (Georgia Tech),
Vivek Sarkar (Georgia Tech)

GPUs are a common source of performance improvement in HPC



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

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 writing GPU kernels using CUDA/OpenCL or other accelerator language, and invoke them from Chapel (Focus of this paper)**

Low-level

RICE 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).

Motivation: Vector Copy (Original)

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

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

RICE Georgia Tech

Motivation: Vector Copy (GPU)

- Invoking CUDA/OpenCL code using the C interoperability feature

```
extern proc GPUVC(A: [] real(32),
                  B: [] real(32),
                  lo: int, hi: int);

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

// Invoking CUDA/OpenCL program
GPUVC(A, B, 1, n);
```

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

Motivation:
The code is not very portable

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

```
// GPU Version
GPUVC(A, B, 1, n);
```

- Potential “portability” problems
  - How to switch back and forth between the the original version and the GPU version?
  - How to support hybrid execution?
  - How to support distributed arrays?

Research Question:
What is an appropriate and portable programming interface that bridges the ”forall” and GPU versions?

Our Solution: GPUIterator

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

```
// GPU Version
GPUVC(A, B, 1, n);
```

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

- Contributions:
  - Design and implementation of the GPUIterator
  - Performance evaluation of different CPU+GPU execution strategies

RICE Georgia Tech

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

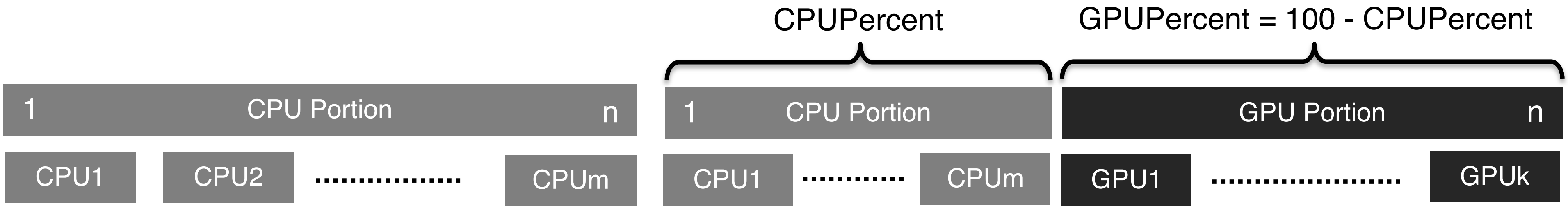
RICE Georgia Tech

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

RICE Georgia Tech

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

The GPUIterator supports Distributed Arrays

```
var D: domain(1) dmapped Block(boundingBox={1..n}) = {1..n};
var A: [D] real(32);
var B: [D] real(32);
var GPUCallBack = lambda (lo: int, hi: int, nElems: int) {
  GPUVC(A.localSlice(lo..hi),
        B.localSlice(lo..hi),
        0, hi-lo, nElems);
};
forall i in GPU(D, GPUCallBack,
                CPUPercent) {
  A(i) = B(i);
}
```

The GPUIterator supports Zippered-forall

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

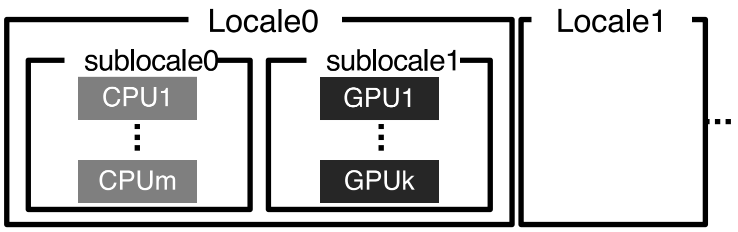
- Restriction
  - The GPUIterator must be the leader iterator

RICE Georgia Tech

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

Implementation of the GPUIterator

- Internal modules
  - https://github.com/ahayashi/chapel
  - Created the GPU Locale model
    - ✓ CHPL_LOCALE_MODEL=gpu



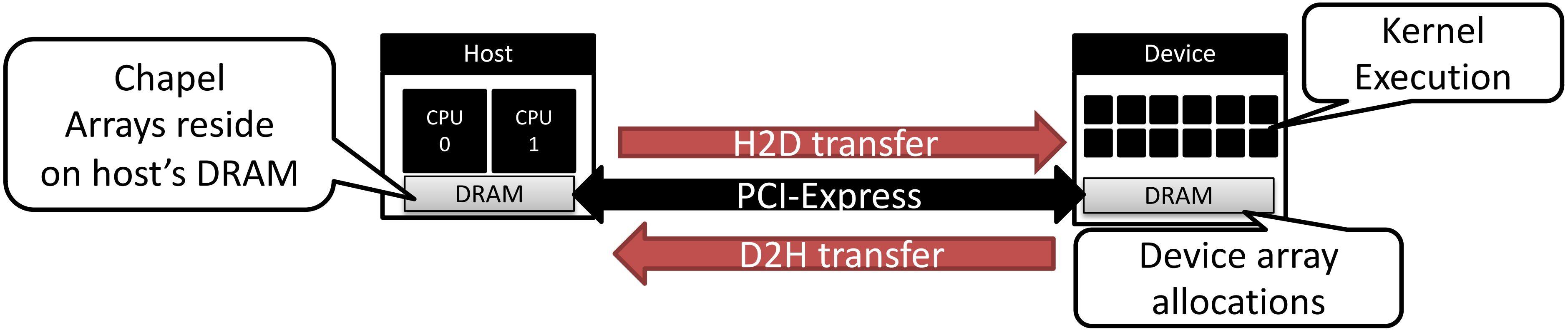
- External modules
  - https://github.com/ahayashi/chapel-gpu
  - Fully implemented in Chapel

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(…);
  }
}
```

Writing CUDA/OpenCL Code for the GPUIterator

- GPU programs for the GPUIterator should include typical host and device operations



Performance Evaluations

- Platforms
  - Intel Xeon CPU (12 cores) + NVIDIA Tesla M2050 GPU
  - IBM POWER8 CPU (24 cores) + NVIDIA Tesla K80 GPU
  - Intel Core i7 CPU (6 cores) + Intel UHD Graphics 630/AMD Radeon Pro 560X
  - Intel Core i5 CPU (4 cores) + NVIDIA TITAN Xp
- Chapel Compilers & Options
  - Chapel Compiler 1.20.0-pre (as of March 27) with the --fast option
- GPU Compilers
  - CUDA: NVCC 7.0.27(M2050), 8.0.61 (K80) with the -O3 option
  - OpenCL: Apple LLVM 10.0.0 with the -O3 option

Performance Evaluations (Cont'd)

- Tasking
  - CUDA: CHPL_TASK=qthreads
  - OpenCL: CHPL_TASK=fifo
- Applications (https://github.com/ahayashi/chapel-gpu)
  - Vector Copy
  - Stream
  - BlackScholes
  - Logistic Regression
  - Matrix Multiplicaiton

How many lines are added/modified?

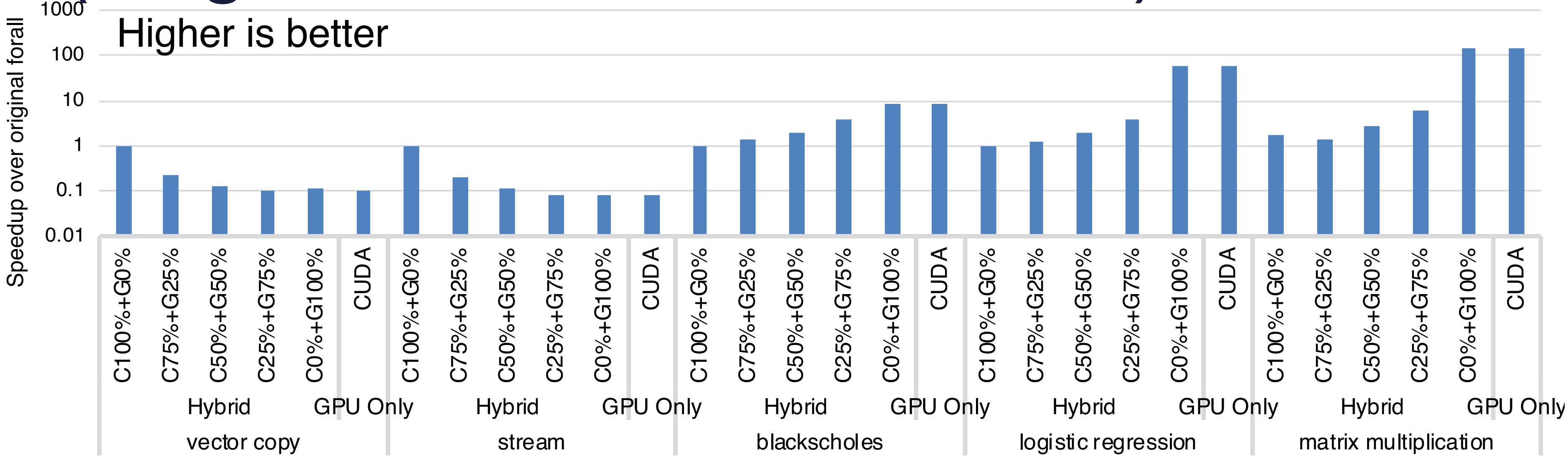
	LOC added/modified (Chapel)	CUDA LOC (for NVIDIA GPUs)	OpenCL LOC (for Intel/AMD GPUs)
Vector Copy	**6**	53	256
Stream	**6**	56	280
BlackScholes	**6**	131	352
Logistic Regression	**11**	97	472
Matrix Multiplication	**6**	213	290

- Source code changes are minimal

RICE Georgia Tech

LOC for CUDA/OpenCL are out of focus

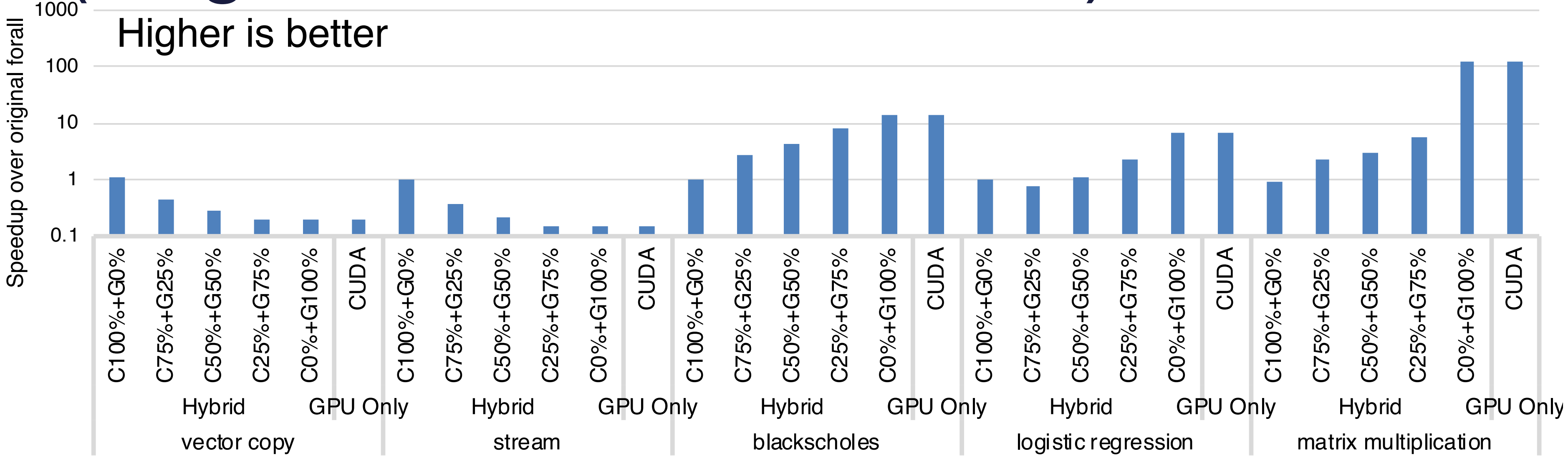
How fast are GPUs? (Single-node, POWER8 + K80)



- 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

RICE Georgia Tech

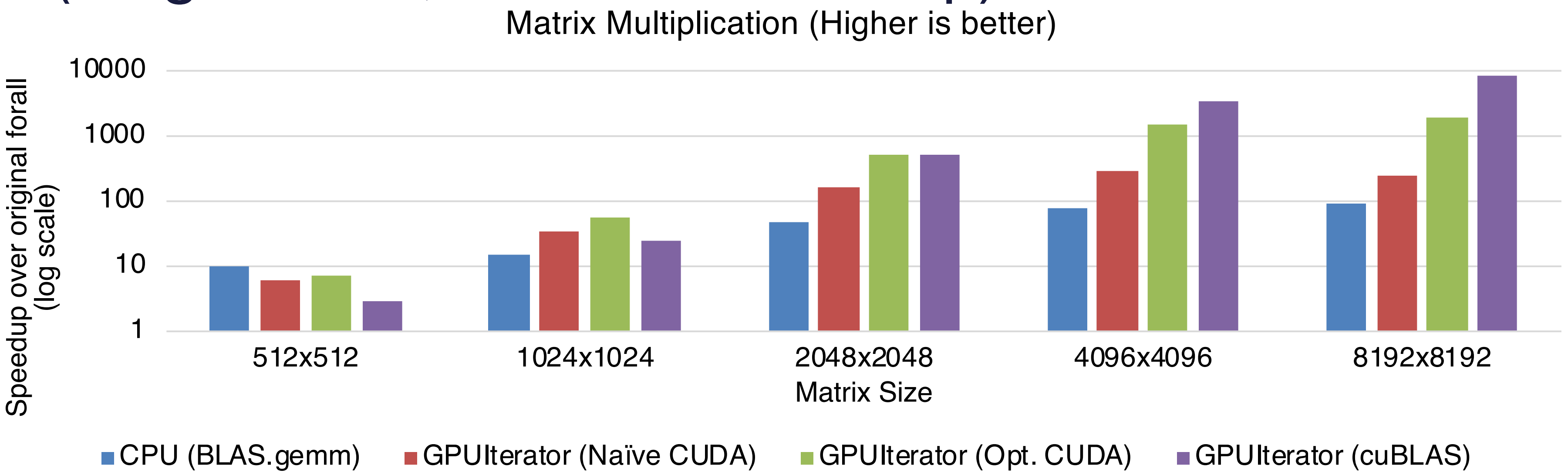
How fast are GPUs? (Single-node, Xeon + M2050)



- 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

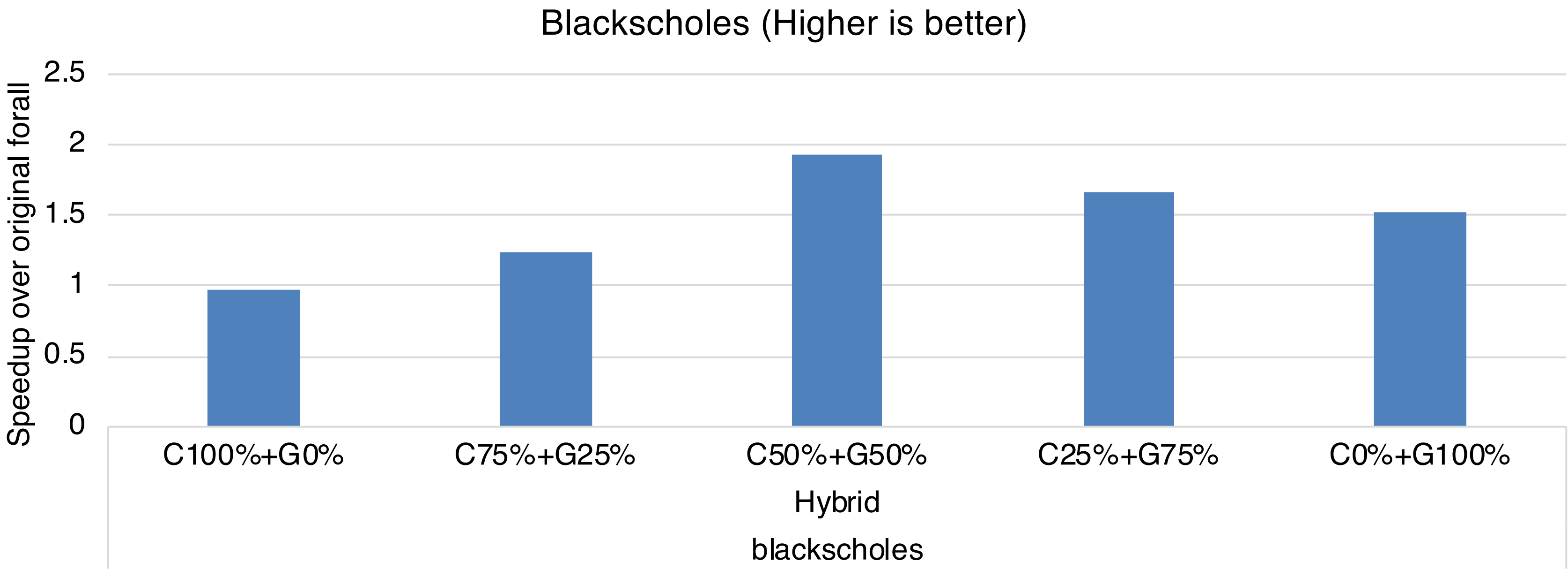
RICE Georgia Tech

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



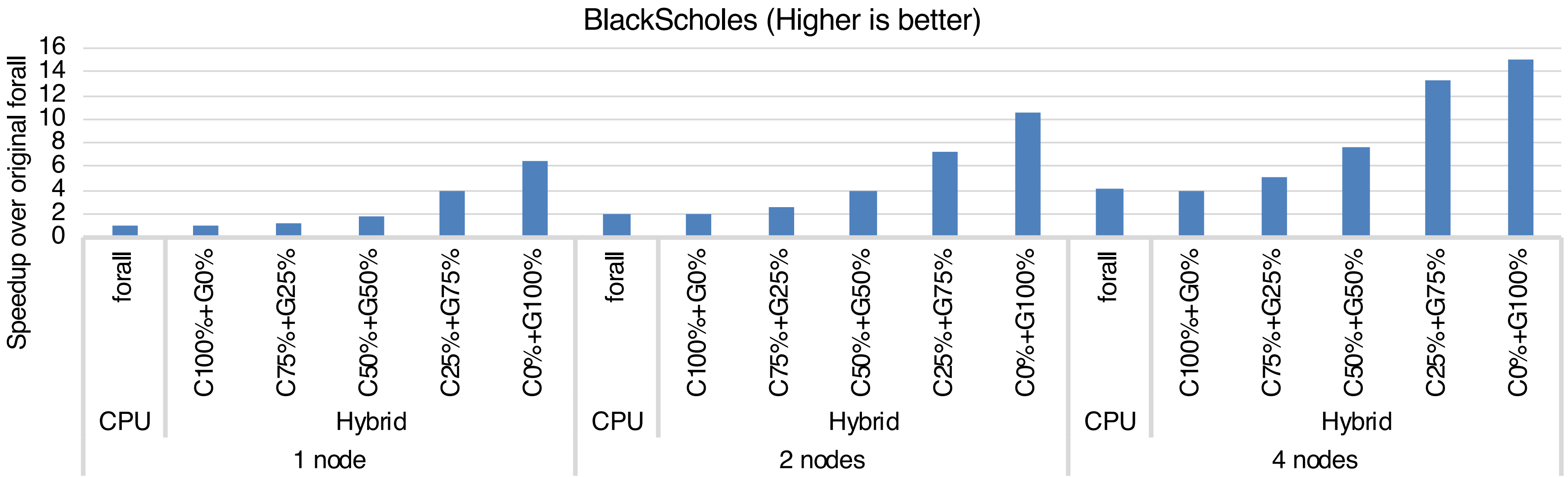
- Motivation: to verify how fast the GPU variants are compared to a highly-tuned Chapel-CPU variant
- Result: the GPU variants are mostly faster than OpenBLAS's gemm (4 core CPUs)

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



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

Multi-node performance numbers (Xeon + M2050)



- ❑ The original forall show good scalability
- ❑ The GPU variants give further performance improvements

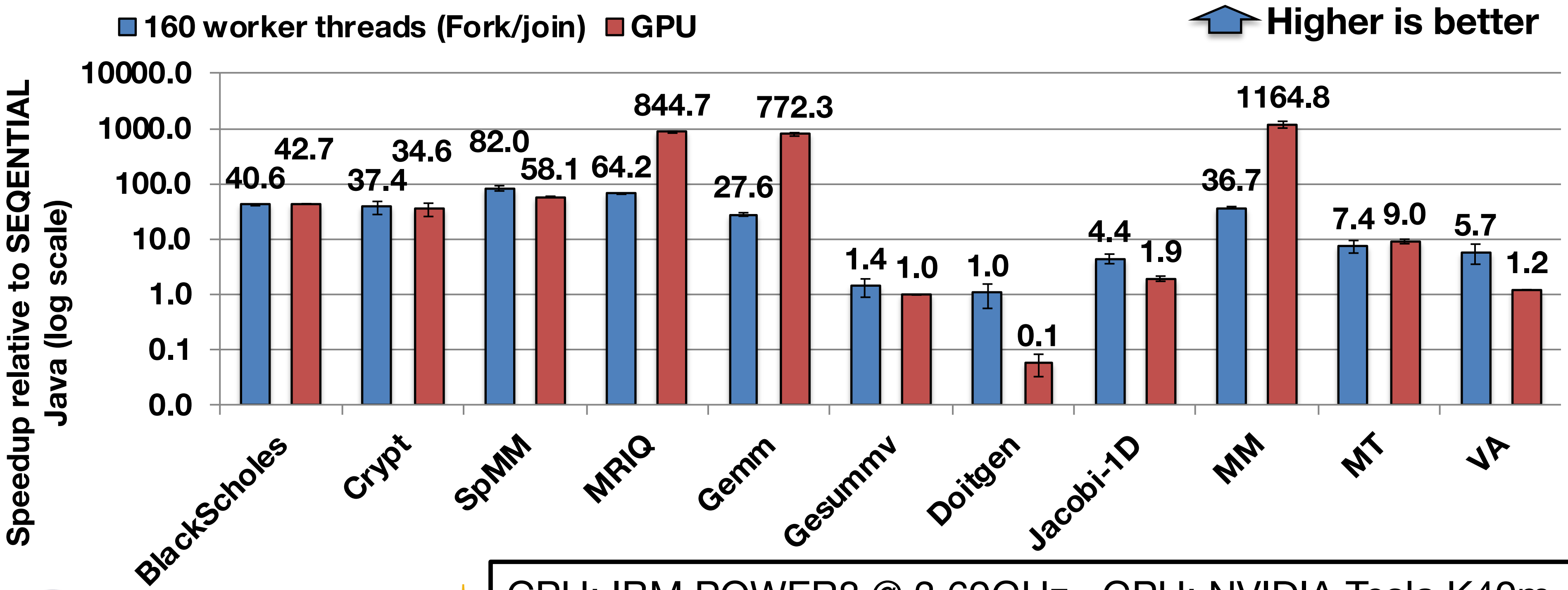
RICE Georgia Tech

Conclusions & Future Work

- Summary
  - The GPUIterator provides an appropriate interface between Chapel and accelerator programs
    - ✓ Source code is available:
      - https://github.com/ahayashi/chapel-gpu
  - The use of GPUs can significantly improves the performance of Chapel programs
- Future Work
  - Support reduction
  - Further performance evaluations on multi-node CPU+GPU systems
  - Automatic selection of the best "CPUPercent"

Backup Slides

GPU is not always faster

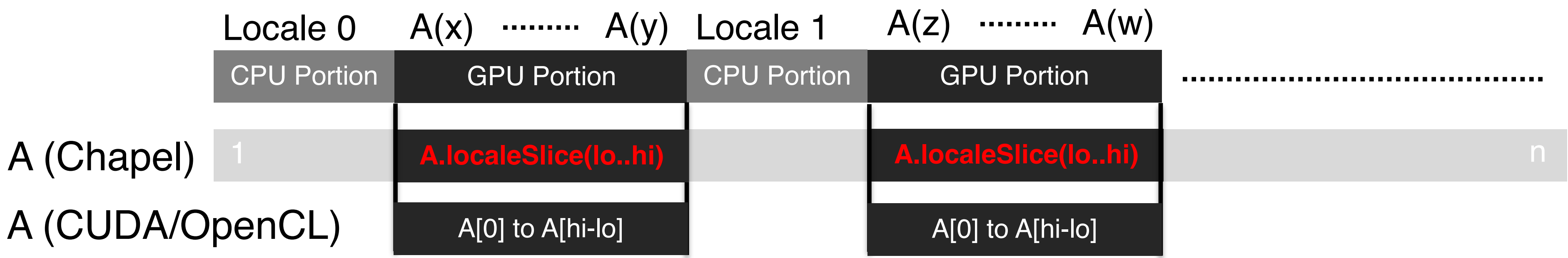


CPU: IBM POWER8 @ 3.69GHz , GPU: NVIDIA Tesla K40m

RICE Georgia Tech

The GPUIterator supports Distributed Arrays (Cont'd)

- No additional modifications for supporting multi-locales executions



Note: localeSlice is Chapel's array API