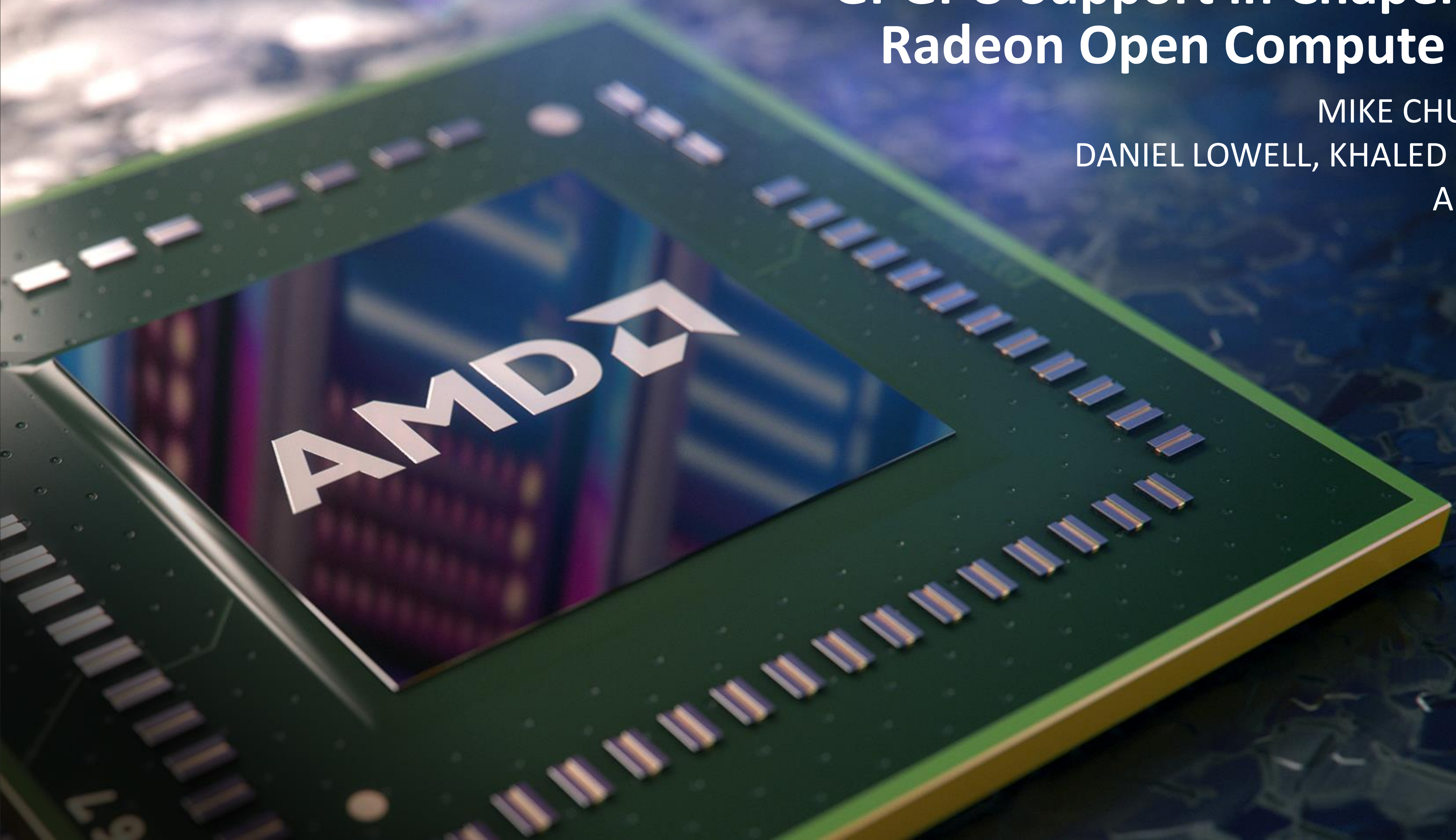
GPGPU Support in Chapel with the Radeon Open Compute Platform

MIKE CHU, ASHWIN AJI,
DANIEL LOWELL, KHALED HAMIDOUCHE
AMD RESEARCH



WHY GPGPUS?

- General Purpose GPUs (GPGPUs) are becoming a standard in high-performance computing
  - Most common accelerator in top supercomputers
  - Important for both improving performance and energy efficiency
- GPUs can better exploit parallelism
  - Many ops per instruction (e.g., 64)
  - Many threads per Compute Engine (hundreds)
  - Many Compute Engines per GPU (tens)
- GPUs can reduce overheads
  - SIMD operations: 1 instruction -> many ops
  - In-order pipeline
  - Overlap memory latency by thread oversubscription (SMT-like)
- Bonus! You already have one in your desktop/laptop/tablet/smartphone

RADEON OPEN COMPUTE PLATFORM (ROCM)

- AMD’s software stack for heterogeneous computing
  - Implementation of the Heterogeneous System Architecture
  - Interface between programming models/languages and AMD hardware

- Provides a software stack for heterogeneous computing
  - HCC C/C++ compiler (LLVM/CLANG based)
  - Support for APUs and Radeon discrete GPUs with HBM
  - Runtime APIs for task queuing, signaling, etc.

- Our goal: study how to interface Chapel with ROCm

GPU PROGRAMMING IN CHAPEL

WHAT WORKS TODAY?

- GPU could simply rely on Chapel's built-in C-interoperability functionality
  - Chapel developers would need to write:
    - Kernel code in GPU-specific language (e.g., OpenCL™)
    - C-host code which handles initialization and queues the kernel for execution
    - Chapel code to call the extern C-function

- Can we make this better?

```
// C host code file
void cpu_func()
{
  // initiate Device, Allocate and copy A on device
  // start the kernel
  err=clEnqueueNDRangeKernel(hello);
  ….
}
```

```
// OpenCL kernel file
_global hello(void * A, …)
{
  int idx= get_global_id(0);
  A[idx] =0;
}
```

```
// Chapel file
proc run ()
{
  extern proc cpu_func(int* A) : void;
  // will call the hello kernel which will be executed on GPU
  cpu_func(A);

}
```

PROPOSED EXTENSION FOR LAUNCHING KERNELS

AMD

- Simple extension for easier launching of current GPU code/libraries
- Removes the need for user to write C-host code
  - Chapel developers would just write in Chapel and their GPU language of choice
- Can the “GPU language of choice” be Chapel itself?

```
// Chapel file
proc run ()
{
  chpl_launch (gpu_func, gpu_file, A);

  coforall loc in Locales
  {
    on loc do chpl_launch (gpu_func, gpu_file, A);
  }
}
```

HSA LOCALE

- What if we relied on Chapel's hierarchical locales to handle what is on/not on GPU?
  - Let the compiler generate the required host and kernel code
  - Chapel developer could then just write Chapel code

```
// Chapel file

proc run (){
  var A : [0..100] int;

  on here.GPU do
  {
    // transform this loop to a kernel when
    // compiling with GPU support
    forall (x, i) in zip(A, A.domain) {
      x = i + 147;
    }
  }
}
```

AUTOMATIC GENERATION OF GPU LOOPS

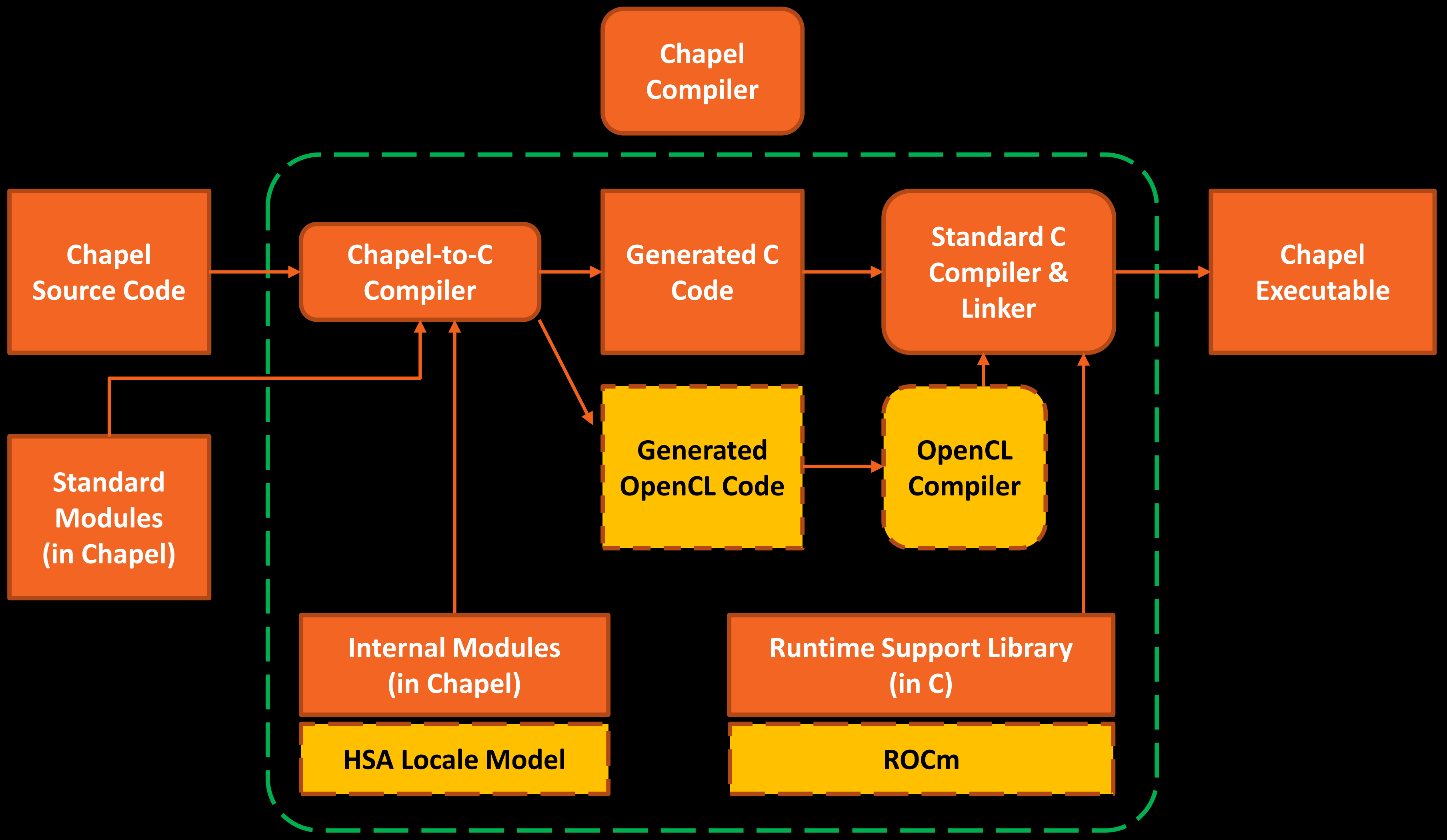
AMD

- Can currently handle a variety of parallel statements within a GPU sublocale
- OpenCL™ is generated for kernel code
- Required C-host code is generated
  - if/else statement which checks if GPU is requested
  - Otherwise, executes original CPU implementation
- Runtime interfaces with ROCm to launch kernels

```
// Chapel file
proc run() {
  var A: [D] int;
  var B: [D] int;
  var C: [D] int;
  on (Locales[0]:LocaleModel).GPU do {
    // 1. Reduction
    var sum = + reduce A;
    // 2. for-all expression
    [i in D] A(i) = i;
    // 3. for-all statement
    forall i in 1..n {
      A[i] = A[i] + 1164;
    }
    // 4. Array assignment
    B = A;
    // 5. Function promotion
    C = square(B);
    // 6. for-all with filtering
    [i in D] if i % 2 == 1 then C[i] = 0;
  }
}
```

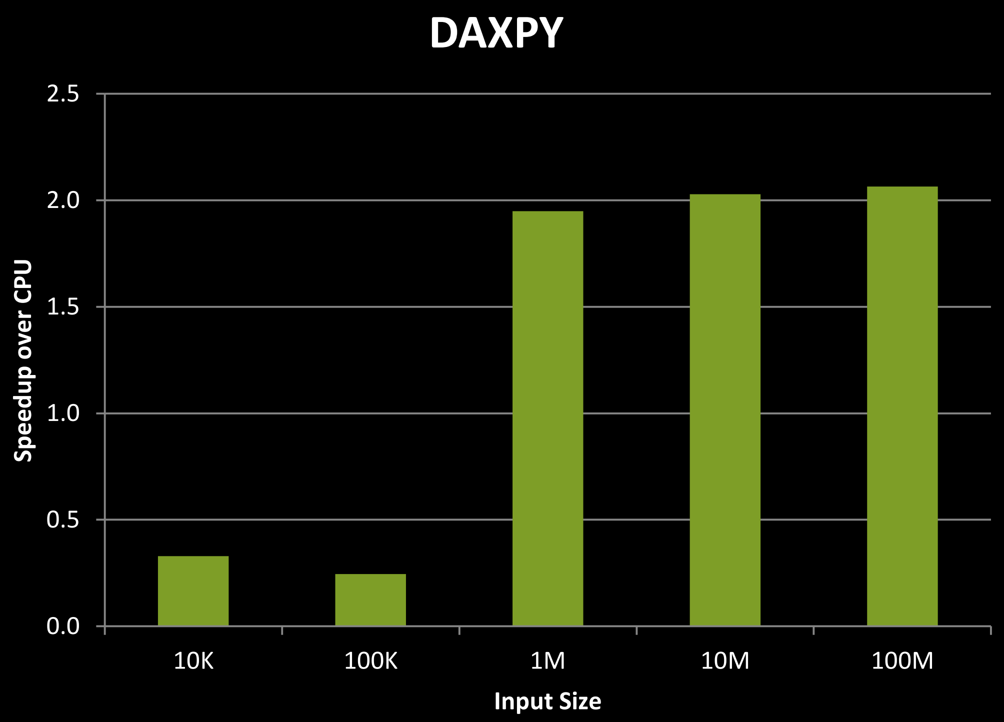
CHAPEL CODEBASE MODIFICATIONS

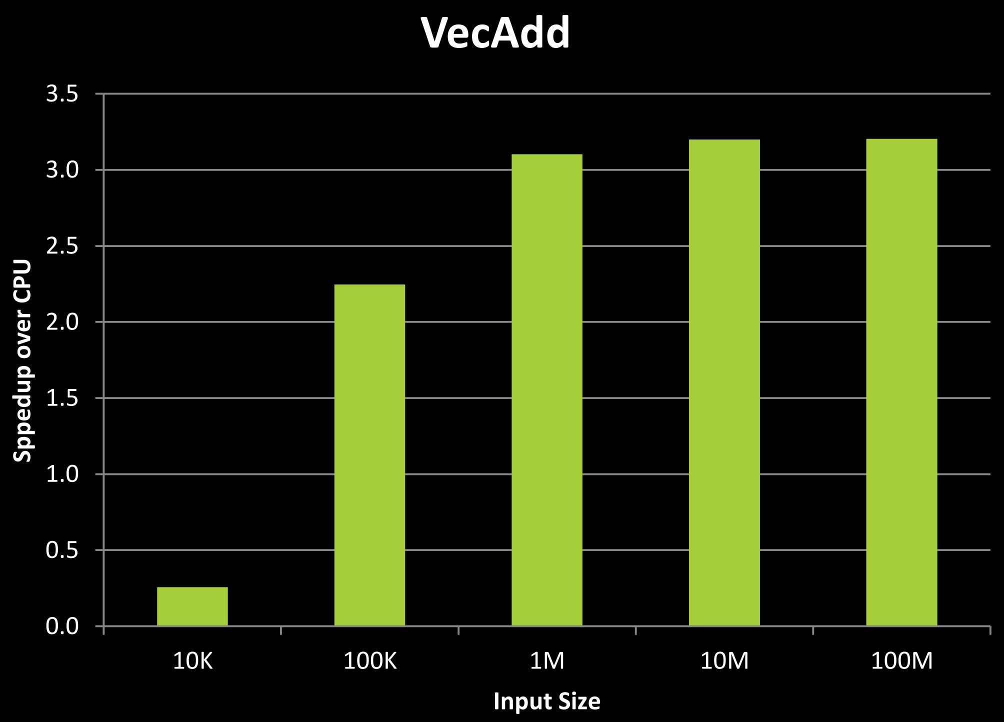
AMD



DAXPY & VECADD

- Both algorithms show benefit over CPU for large input sizes
  - Running on AMD A10-8700B APU



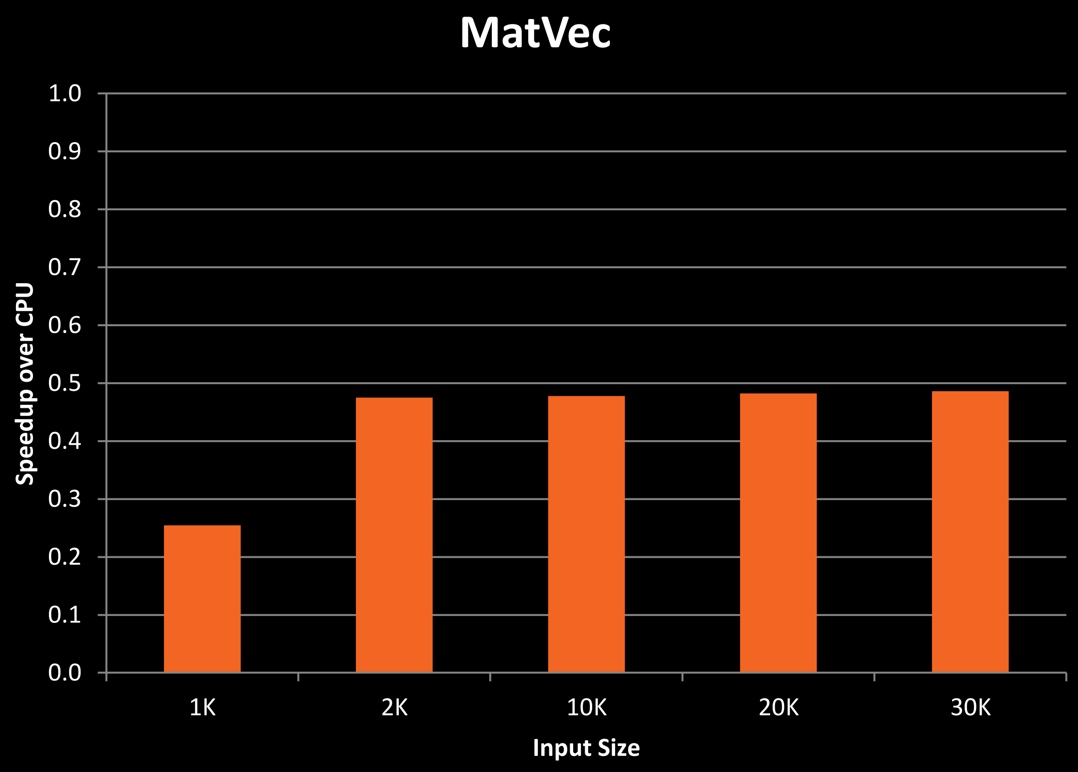


MATRIX-VECTOR MULTIPLICATION

- However, speedups do not translate over to matrix-vector multiplication

- CPU performs better because:
  - Generated GPU code only parallelizing outer loop
  - Each kernel thread is executing another loop
  - GPU programmers typically merge loops, and use grid size and workgroup size parameters to split the loop
  - But… our current implementation cannot merge loops and has no way to specify GPU execution parameters

- How can we make this better?



PROPOSED EXTENSIONS FOR EXECUTION PARAMETERS

- Optimized GPU code requires specification of execution parameters for:
  - Workgroup size
  - Grid size

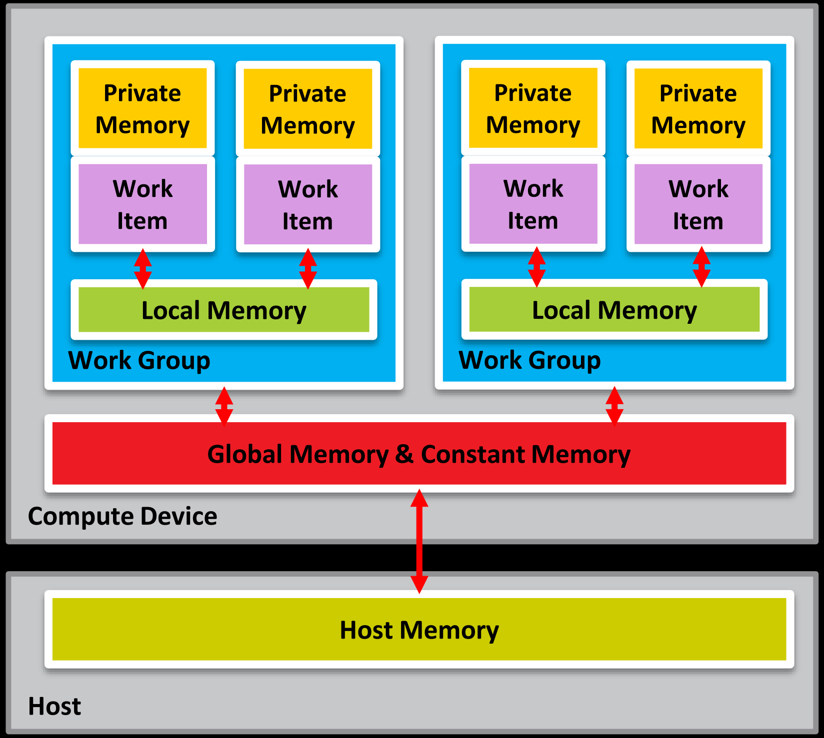
- Discussions have started on how to best add this to the Chapel language

- Current proposal extends "with" intent clause

```
// Chapel file
proc run ()
{
  var A : [0..100] int;
  on here.GPU do
  {
    // transform this loop to a kernel when
    // compiling with GPU support
    forall a in A with (WG(64) GRID(4096)) do {
      // loop body
    }
  }
}
```

PROPOSED EXTENSIONS – MEMORY MANAGEMENT

- GPUs have concepts of local memory (local data store, or LDS)
  - Fast and shared across workgroups
  - High-performance code needs to make use of different memories




```
// Chapel file

proc run ()
{
  var lid = get_locale_id();
  scratch buf : int[64];
  // if called inside Kernel, compiler will
  // automatically allocate a space of 64*sizeof(int)
  // in the LDS.

}
```

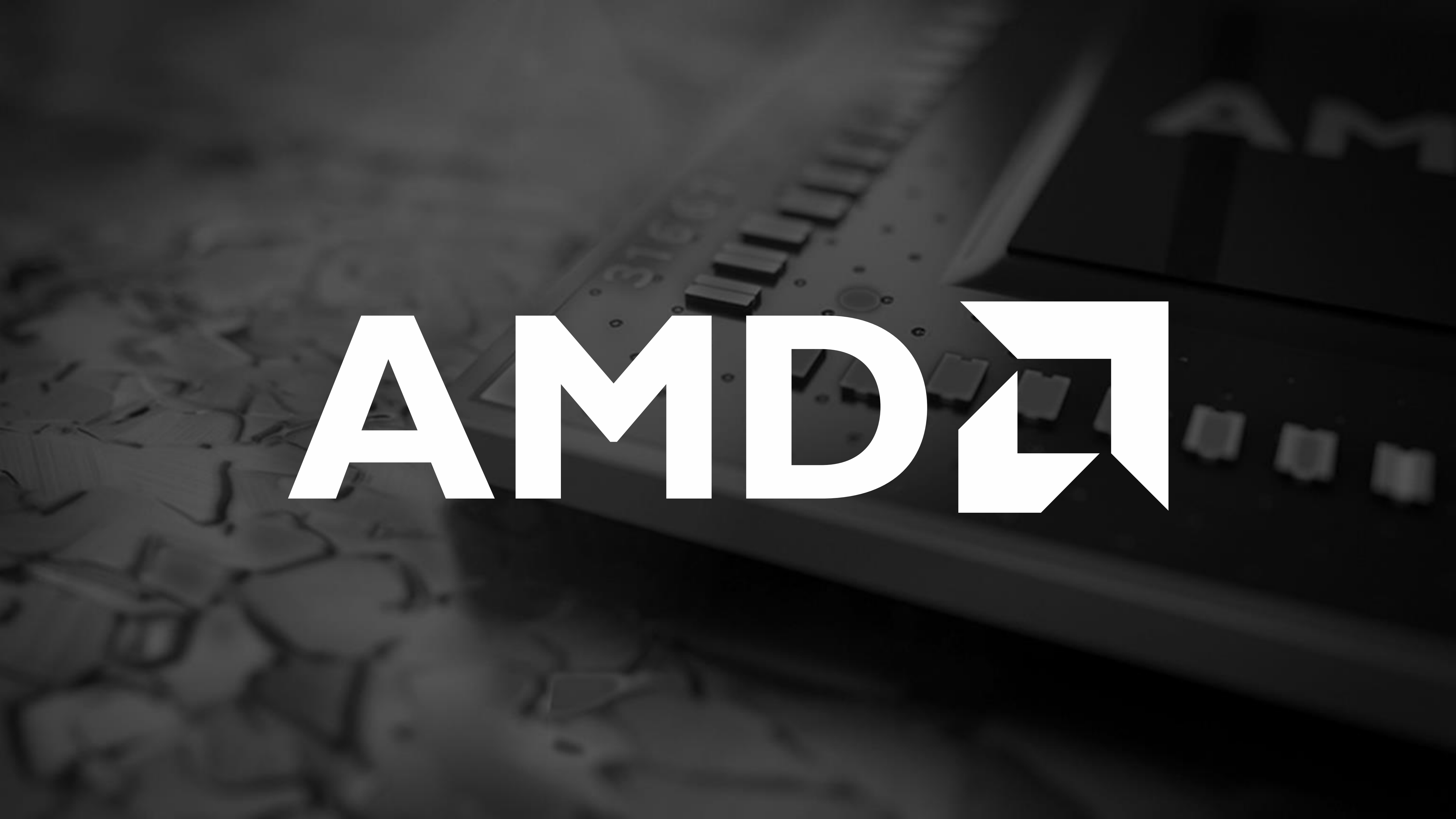
CURRENT STATUS & WHAT'S NEXT?

AMD

- Presented our work at a Chapel deep dive in May 2017
- Open sourced our implementation:
  - git clone –b chpl-hsa-master https://github.com/RadeonOpenCompute/chapel
- Begun working with Cray to look at:
  - What parts of current implementation can move back into master branch
  - How proposed language changes should evolve going forward (created CHIP)

CONCLUSION

- GPUs can be supported in the Chapel language today
  - The interface may be unwieldy and require three programming languages
- By interfacing with ROCm and developing an OpenCL™ codegen compiler pass we can make this better
  - Parallel loops can be translated to kernel language
  - Required host code to launch kernels can be generated by the compiler
  - Runtime can handle queuing of kernel tasks
- To fully exploit capabilities of a GPU, language extensions may be necessary


AMD

DISCLAIMER & ATTRIBUTION

ATTRIBUTION