

A close-up, angled view of an AMD integrated circuit chip. The chip is green with a central square area containing the AMD logo and the word 'AMD' in white. The chip is surrounded by numerous gold-colored pins. The background is a dark blue, textured surface.

# GPGPU Support in Chapel with the Radeon Open Compute Platform

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

- ▲ 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-interopability 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



- ▲ 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);
  }
}
```

- ▲ 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;
    }
  }
}
```

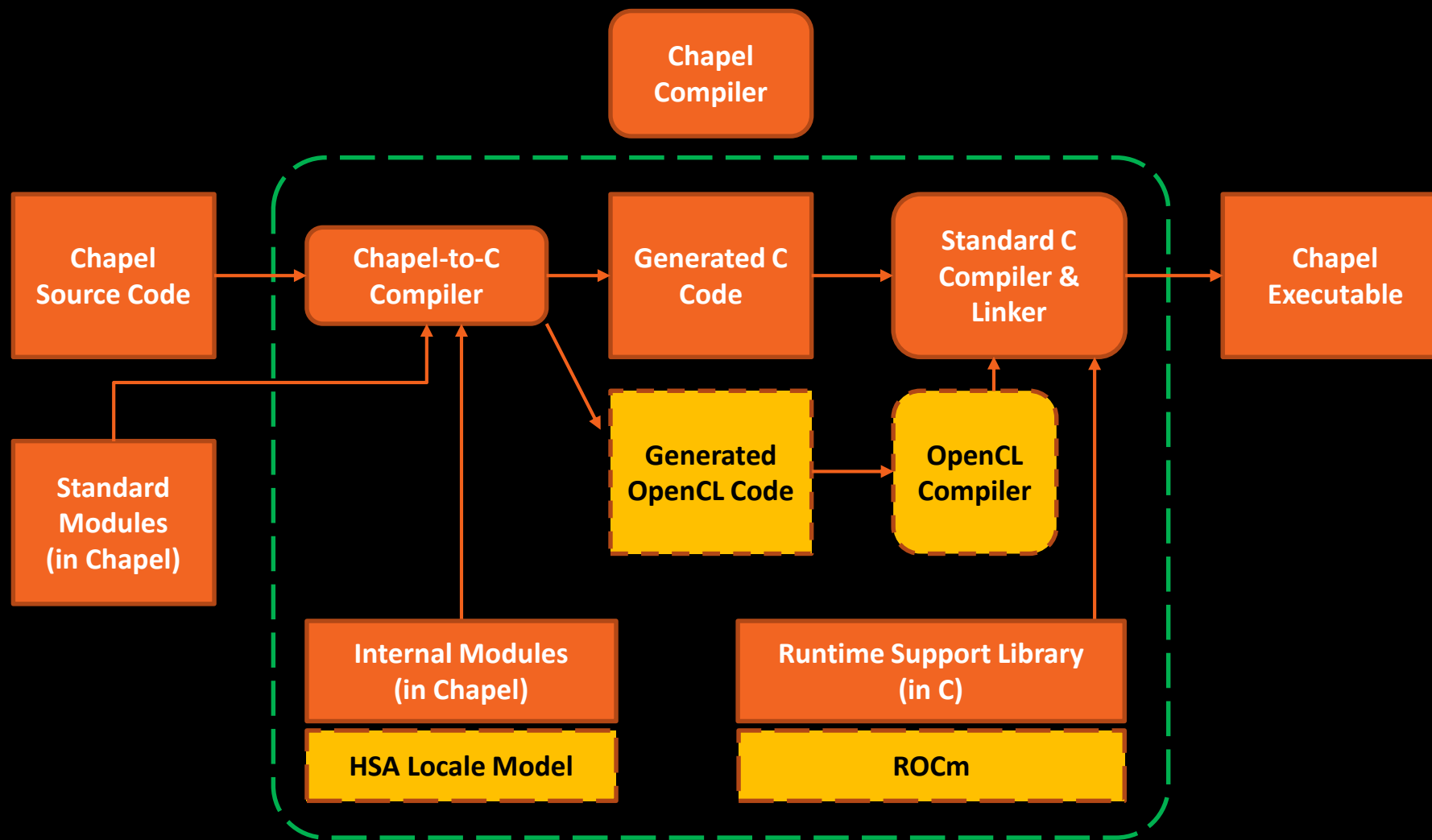
# AUTOMATIC GENERATION OF GPU LOOPS



- ▲ 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;
  }
}
```

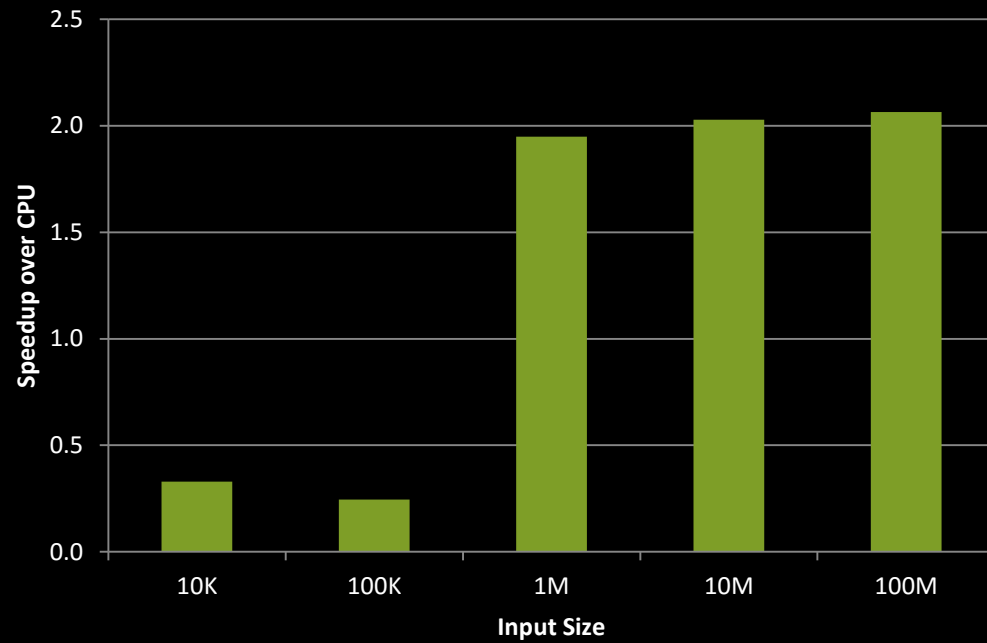
# CHAPEL CODEBASE MODIFICATIONS



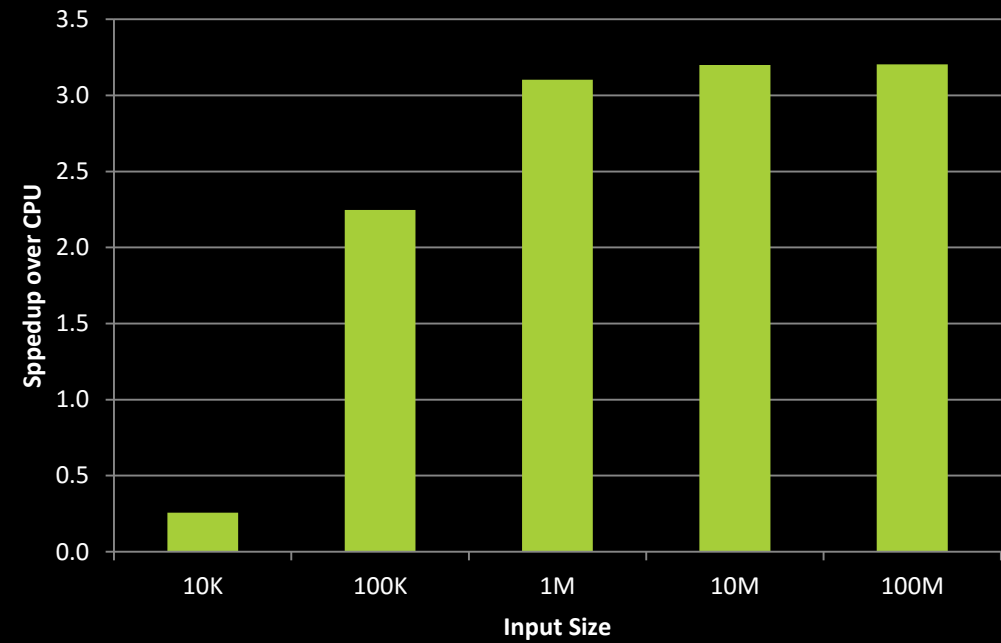


- Both algorithms show benefit over CPU for large input sizes
  - Running on AMD A10-8700B APU

### DAXPY



### VecAdd

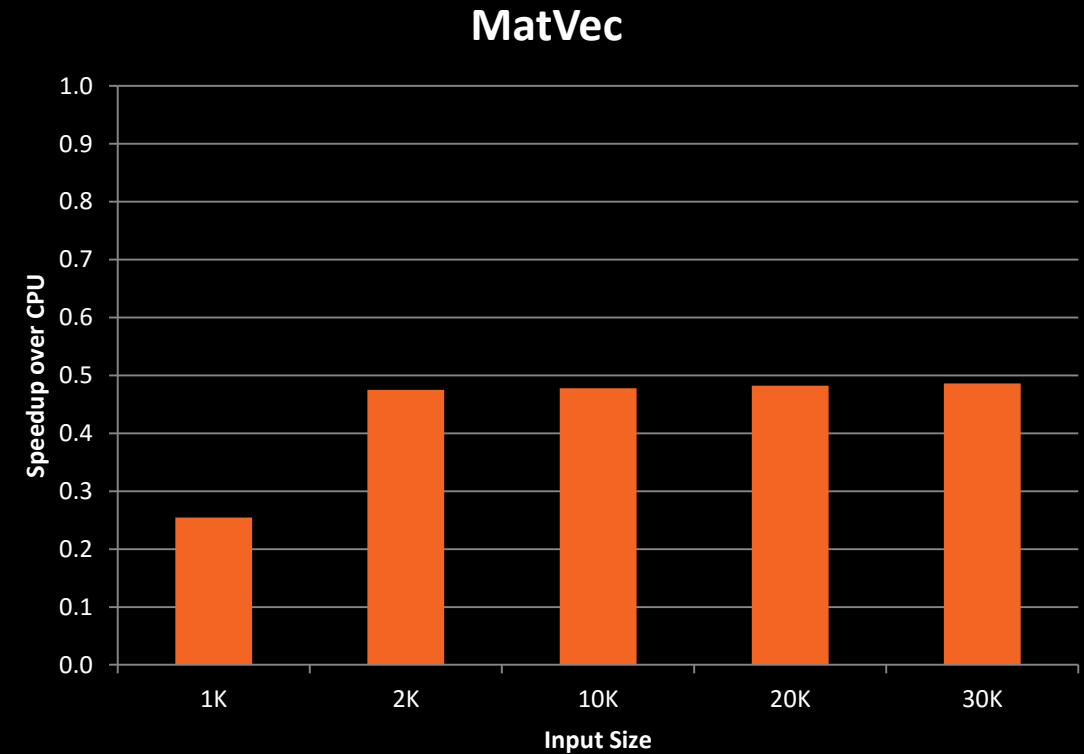


▲ However, speedups do not translate over to matrix-vector multiplication

▲ 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?



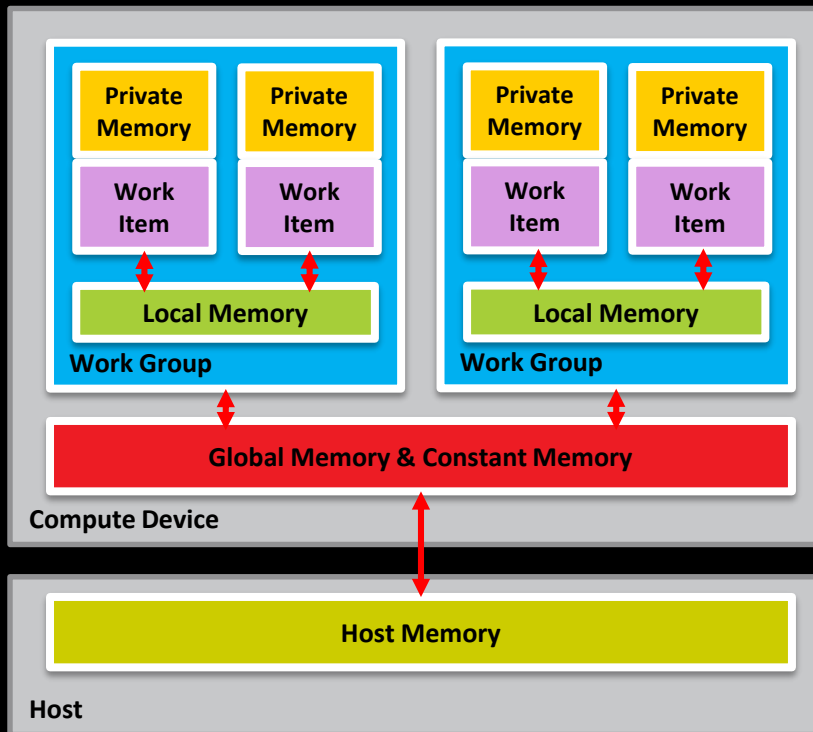
- ▲ Optimized GPU code requires specification of execution parameters for:
  - Workgroup size
  - Grid size
- ▲ Discussions have started on how to best add this to the Chapel language
- ▲ Current proposal extends “with” intent clause

```
// 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



- GPU 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?



- ▲ 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)

- ▲ 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

The image features the AMD logo in a bold, white, sans-serif font, centered horizontally. The logo consists of the letters 'A', 'M', 'D', and a stylized square symbol with a diagonal cut. The background is a dark, grayscale photograph of a computer keyboard, with the keys and the 'STEP' label on a function key visible. The lighting is dramatic, with the keyboard keys appearing as bright highlights against a dark background.

**AMD**

# DISCLAIMER & ATTRIBUTION



The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.

The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

## ATTRIBUTION

© 2017 Advanced Micro Devices, Inc. All rights reserved. AMD, the AMD Arrow logo and combinations thereof are trademarks of Advanced Micro Devices, Inc. in the United States and/or other jurisdictions. Other names are for informational purposes only and may be trademarks of their respective owners.