ROCM:
An open platform for GPU computing exploration

Ben Sander:
Senior Fellow, Radeon Open Compute

Gregory Stoner:
Senior Director, Radeon Open Compute
REVOLUTION IN GPU COMPUTING

Radeon Open Compute Platform (ROCm)
Modern Heterogeneous HPC and Hyper Scale Accelerator Platform for Large Scale Systems

Performance
Rich foundation built for latency reduction and throughput optimization

Open
First fully "Open Source" professional GPU accelerator computing solution

Hyper Scale
Built from the ground up to service multi-accelerators in node and across the rack
Introducing ROCm Software Platform
A new, fully “Open Source” foundation for Hyper Scale and HPC-class GPU computing

Graphics Core Next Headless Linux® 64-bit Driver
- Large memory single allocation
- Peer-to-Peer Multi-GPU
- Peer-to-Peer with RDMA
- Systems management API and tools

HSA Drives Rich Capabilities Into the ROCm Hardware and Software
- User mode queues
- Architected queuing language
- Flat memory addressing
- Atomic memory transactions
- Process concurrency & preemption

Rich Compiler Foundation For HPC Developer
- LLVM native GCN ISA code generation
- Offline compilation support
- Standardized loader and code object format
- GCN ISA assembler and disassembler
- Full documentation to GCN ISA

“Open Source” Tools and Libraries
- Rich Set of “Open Source” math libraries
- Tuned “Deep Learning” frameworks
- Optimized parallel programming frameworks
- CodeXL profiler and GDB debugging
## ROCm Programming Model Options

### HIP

**Convert CUDA to portable C++**

- Single-source Host+Kernel
- C++ Kernel Language
- C Runtime
- Platforms: AMD GPU, NVIDIA (same perf as native CUDA)

**When to use it?**

- Port existing CUDA code
- Developers familiar with CUDA
- New project that needs portability to AMD and NVIDIA

### HCC

**True single-source C++ accelerator language**

- Single-source Host+Kernel
- C++ Kernel Language
- C++ Runtime
- Platforms: AMD GPU

**When to use it?**

- New projects where true C++ language preferred
- Use features from latest ISO C++ standards

### OpenCL

**Khronos Industry Standard accelerator language**

- Split Host/Kernel
- C99-based Kernel Language
- C Runtime
- Platforms: CPU, GPU, FPGA

**When to use it?**

- Port existing OpenCL code
- New project that needs portability to CPU, GPU, FPGA
HIP : Key Features

- Strong support for most commonly used parts of CUDA API
  - Streams, events, memory allocation/deallocation, profiling
  - HIP includes driver API support (modules and contexts)

- Full C++ support including templates, namespace, classes, lambdas
  - AMD’s open-source GPU compiler based on near-tip clang+llvm
  - Support C++11, C++14, some C++17 features

- Hipified code is portable to AMD/ROCM and NVIDIA/CUDA
  - On CUDA, developers can use native CUDA tools (nvcc, nvprof, etc)
  - On ROCM, developers can use native ROCM tools (hcc, rocm-prof, codexl)
  - HIP ecosystem includes hipBlas, hipFFT, hipRNG, MIOpen

- Hipify tools automate the translation from CUDA to HIP
  - Developers should expect some final cleanup and performance tuning
Hipification of CUDA Kernel (CAFFE)

CUDA

namespace caffe {

template <typename Dtype> 
__global__ void 
BNLLForward(const int n,
const Dtype* in, Dtype* out)
{
for (int i = blockIdx.x * blockDim.x + threadIdx.x;
    i < (n); i += blockDim.x * gridDim.x) {
    out[index] = in[index] > 0 ? 
in[index] + log(1. + exp(-in[index])) : 
log(1. + exp(in[index]));
}
}

HIP

namespace caffe {

template <typename Dtype> 
__global__ void 
BNLLForward(hipLaunchParm lp, const int n,
const Dtype* in, Dtype* out)
{
for (int i = hipBlockIdx_x * hipBlockSize + hipThreadIdx_x;
    i < (n); i += hipBlockSize * hipGridSize) {
    out[index] = in[index] > 0 ? 
in[index] + log(1. + exp(-in[index])) : 
log(1. + exp(in[index]));
}
}
Hipification of CUDA Runtime APIs (CAFFE)

CUDA

```c
void SyncedMemory::async_gpu_push(const cudaStream_t& stream) {
    CHECK(head_ == HEAD_AT_CPU);
    if (gpu_ptr_ == NULL) {
        cudaGetDevice(&gpu_device_);
        cudaMalloc(&gpu_ptr_, size_);
        own_gpu_data_ = true;
    }
    const cudaMemcpyKind put = cudaMemcpyHostToDevice;
    cudaMemcpyAsync(gpu_ptr_, cpu_ptr_, size_, put, stream);
    // Assume caller will synchronize on the stream
    head_ = SYNCED;
}
```

HIP

```c
void SyncedMemory::async_gpu_push(const hipStream_t& stream) {
    CHECK(head_ == HEAD_AT_CPU);
    if (gpu_ptr_ == NULL) {
        hipGetDevice(&gpu_device_);
        hipMalloc(&gpu_ptr_, size_);
        own_gpu_data_ = true;
    }
    const hipMemcpyKind put = hipMemcpyHostToDevice;
    hipMemcpyAsync(gpu_ptr_, cpu_ptr_, size_, put, stream);
    // Assume caller will synchronize on the stream
    head_ = SYNCED;
}
```
Porting with hipify tool

- ~99%+ Automatic Conversion
- Developer maintains HIP port
- Resulting C++ code runs on NVIDIA or AMD GPUs
HIP Compilation Process

**Portable HIP C++ (Kernels + HIP API)**

**NVIDIA**
- HIP API implemented as inlined calls to CUDA Runtime
- Compute kernels mostly unchanged
- Code compiled with NVCC (same as CUDA)
- Can use `nvprof`, CUDA debugger, other tools
- Source Portable
- Not Binary Portable

**AMD**
- HIP API implemented with lightweight HIP runtime
- Uses HCC's `hc::accelerator`, `hc::accelerator_view`, `hc::completion_future`
- Some calls directly into ROCm RT
- Compute kernels mostly unchanged
- Code compiled with HCC
- Can use CodeXL Profiler/Debugger

**HIP->CUDA Header**
- CUDA (Kernels + CUDA API)
  - NVCC
  - CUDA Executable

**HIP->HC Header**
- HCC C++ (Kernels + HC/ROCr)
  - HCC
  - HCC Executable
**ROCm : Deep Learning Gets HIP**

Bringing a faster path to bring deep learning application to AMD GPUs

- **The Challenge: CAFFE**
  - Popular machine-learning framework
  - Tip version on GitHub has 55000+ lines-of-code
  - GPU-accelerated with CUDA

- **Results:**
  - 99.6% of code unmodified or automatically converted
  - Port required less than 1 week developer time
  - Supports all CAFFE features (multi-gpu, P2P, FFT filters)
  - HIPCAFFE is the fastest CAFFE on AMD hardware – 1.8X faster than CAFFE/OpenCL

---

**Complexity of Application Porting: CAFFE**

<table>
<thead>
<tr>
<th>Complexity</th>
<th>Lines of Code Changed</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OpenCL Port</strong></td>
<td>32227</td>
</tr>
<tr>
<td><strong>HIP Port</strong></td>
<td>219</td>
</tr>
</tbody>
</table>

*Manual, 32227*  
*Automatic, 688*  

AMD Internal Data
HCC: Heterogeneous Compute Compiler

Architecture:
- Built on open-source CLANG/LLVM
- Single source compiler for both CPU & GPU
- Standard object code can be linked with g++, clang, icc
- Performance optimized for accelerators:
  - Explicit and implicit data movement
  - Scratchpad memories
  - Asynchronous commands

Dialects:
- HC:
  - C++ runtime – hc::accelerator, hc::accelerator_view, hc::completion_future
  - Kernels launched with parallel_for_each around lambda expression
- ISO C++
  - C++17 Parallel Standard Template Library
  - Next steps include executors and concurrency controls
- OpenMP
  - OpenMP 3.1 support for CPU
  - OpenMP 4.5 GPU offload at SC2016
#include <hc.hpp>
int main(int argc, char *argv[]) {
    int sizeElements = 1000000;

    // Alloc auto-managed array_view
    hc::array_view<double> A(sizeElements);
    hc::array_view<double> B(sizeElements);
    hc::array_view<double> C(sizeElements);

    // Initialize host memory
    for (int i = 0; i < sizeElements; i++) {
        A[i] = 1.618 * i;
        B[i] = 3.142 * i;
    }

    // Tell runtime not to copy CPU host data.
    C.discard_data();

    // Launch kernel onto default accelerator
    // HCC runtime ensures that A and B are available on
    // the accelerator before kernel launch:
    hc::parallel_for_each(hc::extent<1>(sizeElements),
        [=] (hc::index<1> idx) [[hc]] {
            // Kernel is lambda inside for-loop
            int i = idx[0];
            C[i] = A[i] + B[i];
        });

    // Check result
    for (int i = 0; i < sizeElements; i++) {
        double ref = 1.618 * i + 3.142 * i;
        // Because C is an array_view, the HCC runtime
        // will copy C back to host at first access:
        if (C[i] != ref) {
            printf("error:%d computed=%6.2f, reference=%6.2f\n", i, C[i], ref);
        }
    }
}
HCC Example – “HC” Syntax
EXPLICIT MEMORY MANAGEMENT VIA ARRAY

```c++
#include <hc.hpp>

int main(int argc, char *argv[]) {
    int sizeElements = 1000000;

    // Alloc GPU arrays
    hc::array<double> Ad(sizeElements);
    hc::array<double> Bd(sizeElements);
    hc::array<double> Cd(sizeElements);
    double * Ah = malloc(sizeElements*8);
    double * Bh = malloc(sizeElements*8);
    double * Ch = malloc(sizeElements*8);

    // Initialize host memory
    for (int i=0; i<sizeElements; i++) {
        Ah[i] = 1.618 * i;
        Bh[i] = 3.142 * i;
    }

    // Copy host data to GPU
    copy(Ah, Ad);
    copy(Bh, Bd);

    // Launch kernel onto default accelerator
    // HCC runtime ensures that A and B are available on
    // the accelerator before kernel launch:
    hc::parallel_for_each(hc::extent<1> (sizeElements),
        [=] (hc::index<1> idx) [[hc]] {
            // Kernel is lambda inside for-loop
            int i = idx[0];
            Cd[i] = Ad[i] + Bd[i];
        });
    copy(Cd, Ch); // Copy results GPU to host

    // Check result
    for (int i=0; i<sizeElements; i++) {
        double ref= 1.618 * i + 3.142 * i;
        if (Ch[i] != ref) {
            printf("error:%d computed=%6.2f, reference=%6.2f\n", i , Ch[i], ref);
        }
    }
}
```
HCC “HC” Mode

KEY FEATURES

- Many core structures similar to C++AMP
  - Implementation uses "hc" namespace
  - `hc::accelerator_view`, `hc::array_view`, `hc::completion_future`
  - With expanded capabilities...

- Controls over asynchronous kernel and data commands
  - `hc::parallel_for_each` returns `hc::completion_future`
  - Asynchronous copy commands
  - C++17 `then`, `when_any`, `when_all` for managing device-side dependencies [under development]

- Memory Management
  - Approachable `hc::array_view` for managed memory and implicit synchronization
  - Explicit pointer-based memory allocation (`am_alloc` / `am_free`)

- Language Restrictions
  - Remove C++AMP “restrict”
  - Support rich set of C++ language features and data types
  - Advanced C++ language features (virtual functions, recursion, etc) [under development]
ISO C++17 Parallel STL

- **Standard Template Library**
  - `sort(data.begin(), data.end());` // STL
  - `sort(par, data.begin(), data.end());` // PSTL

- **Execution policy**
  - New first parameter to PSTL function
  - `par` indicates algorithm can be run in parallel
  - Can accelerate and run on GPU or multicore CPU
  - Abstraction allows use of architecture-specific optimizations (workgroups, LDS)
  - Formalization of ideas in TBB, NV Thrust, Bolt libs

- **Proposal for C++ 17 Parallelism Tech Spec**
  - Approved in Jacksonville ISO meeting!

- **Next steps:**
  - Executors to control where (which device)
  - Provide `std::future` to track status
ISO C++: Template Library for Parallel For Loops

- [Proposed for C++20 and currently under discussion](http://open-std.org/JTC1/SC22/WG21/docs/papers/2016/p0075r1.pdf)
- Provides straightforward porting of OpenMP #pragma loops into C++
- Key advantage over Parallel STL is that “position” (i) inside loop can be easily determined
- For_loop, for_loop_strided, reductions, inductions
- Similar to PSTL, par policy can be extended with Executors to control where/how kernel is executed

```cpp
// Propose ISO C++ parallel for_loop:
void saxpy_ref(int n, float a, float x[], float y[]) {
  for_loop(par, 0, n, [&](int i) {
    y[i] += a *x[i];
  });
}
```
GPU Architecture Basics
- Memory-based queues used to schedule and execute commands
- Commands include data-copies, parallel execution “kernels”, dependencies, configuration
- Hardware-based dependency resolution
  - Efficiently wait for dependencies, signal completion – all without host intervention

hc::completion_future
- Based on C++ std::future
- Returned by asynchronous commands
- Extend “then” to schedule device-side commands (no host intervention)
  - HCC implementation identifies accelerator commands via specialization and leverages GPU HW
  - `copy(...) . then(for_each(...) . then(copy(...)`;
  - `when_all, when_any` (N4501)
    - Combine futures, return another future, in a single function
    - Can leverage dependency resolution hardware
Delivering An Open Platform For GPU Computing

Language neutral solution to match developer needs as heterogeneous programing models evolve

- Compiler Front End (CLANG)
  - GCN Compiler
    - Direct-to-ISA
    - GCN Docs
    - CLANG/LLVM
    - GCN Assembler
    - Open-source
  - CPU Compiler
    - LLVM Opt Passes
    - CPU ISA Target
  - GPU Code
  - CPU Code

- Language Runtime API
- UCX
- ROCr System Runtime API
- ROck/AMDGPU Driver

- Linux OS
Benefits from Open Source Community

typedef float MyFloat_t;

__global__ void
scale(MyFloat_t *c, MyFloat_t *a)
{
    const MyFloat_t scalar = 3.0;
    const int i = blockDim.x * blockIdx.x + threadIdx.x;
    c[i] = scalar * a[i];
}

nVidia NVCC closed-source compiler

typo_type.cpp(8): error: identifier "Myfloat_t" is undefined
    const Myfloat_t scalar = 3.0;
    ^~~~~~~~~
MyFloat_t
typo_type.cpp:3:15: note: 'MyFloat_t' declared here
typedef float MyFloat_t;

AMD HCC open-source compiler

typo_type.cpp:8:11: error: unknown type name 'MyFloat_t'; did you mean 'MyFloat_t'?
    const Myfloat_t scalar = 3.0;
    ^~~~~~~~~
MyFloat_t
typo_type.cpp:3:15: note: 'MyFloat_t' declared here
typedef float MyFloat_t;
ROCm Supports OpenCL™

OpenCL 1.2+
- New Core Foundation to best leverage ROCr runtime
- OpenCL 2.0 Kernel Language
- OpenCL 1.2 compatible runtime

Key Features
- Coarse Grain SVM
- C11 Atomics
- OpenCL 2.0 Images Support
- Latency to compute optimization
- User Mode DMA – Dual engines with ASYNC transfer, User Mode Queue support

New GCN ISA LLVM Code Generator
- Support GCN ISA assembly optimization, Assembler, Disassembler, inline ASM
- Support Offline, ahead of time compilation
- Register allocation and occupancy controls
# Innovation by Terminology?

<table>
<thead>
<tr>
<th>Term</th>
<th>HIP</th>
<th>HC</th>
<th>OpenCL 1.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device</td>
<td>int deviceld (0..n-1)</td>
<td>hc::accelerator</td>
<td>cl_device</td>
</tr>
<tr>
<td>Queue</td>
<td>hipStream_t</td>
<td>hc::accelerator_view</td>
<td>cl_command_queue</td>
</tr>
<tr>
<td>Event</td>
<td>hipEvent_t</td>
<td>hc::completion_future</td>
<td>cl_event</td>
</tr>
<tr>
<td>Memory</td>
<td>void *</td>
<td>void *; hc::array;</td>
<td>cl_mem</td>
</tr>
<tr>
<td></td>
<td></td>
<td>hc::array_view</td>
<td></td>
</tr>
<tr>
<td></td>
<td>grid</td>
<td>extent</td>
<td>NDRange</td>
</tr>
<tr>
<td></td>
<td>block</td>
<td>tile</td>
<td>work-group</td>
</tr>
<tr>
<td></td>
<td>thread</td>
<td>thread</td>
<td>work-item</td>
</tr>
<tr>
<td></td>
<td>warp</td>
<td>wavefront</td>
<td>sub-group</td>
</tr>
<tr>
<td>Device Kernel</td>
<td><strong>global</strong></td>
<td>lambda inside</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>hc::parallel_for_each or [[hc]] __kernel</td>
<td></td>
</tr>
<tr>
<td>Kernel Launch</td>
<td>hipLaunchKernel</td>
<td>hc::parallel_for_each</td>
<td>clEnqueueNDRangeKernel</td>
</tr>
<tr>
<td>Atomic Builtins</td>
<td>atomicAdd</td>
<td>hc::atomic_fetch_add</td>
<td>atomic_add</td>
</tr>
<tr>
<td>Precise Math</td>
<td>cos(f)</td>
<td>hc::precise_math::cos(f)</td>
<td>cos(f)</td>
</tr>
</tbody>
</table>
Extending Support To A Broader Hardware Ecosystem

ROCM “Open Source” platform brings a rich foundation to these new ecosystems

AMD64 Support
- AMD “Zen”
- Intel Xeon E5 v3 v4

ARM® AArch64 Support
- Cavium ThunderX

IBM OpenPower Support
- IBM Power 8

ROCm is being built to support next generation I/O Interfaces

GenZ Founding Member

CCIX Founding Member

OpenCAPI Founding Member
miOpen

- Open-source optimized Deep Learning GPU kernels for OpenCL and HIP
  - Convolutions
  - Pooling
  - Softmax
  - Normalization
  - Activation Functions
  - Data as 4-D tensor

- Describes operations as a function on tensors
  - Example: a convolution

- Support major MI frameworks including CAFFE, TensorFlow, Torch [under development]
Open-source computing – who wins?

Developers
Community delivers superior tools
First access to new language features

Applications
Source access enables control and optimization

Research
Innovate above the infrastructure
ROCM: First open GPU compiler

Customers
Value and request open solutions
Some ROCm Research Opportunities
OPEN SOURCE COMPILER AND RUNTIME

- GPU Register Allocation and Optimization
  - Large register files (i.e. 256/thread)
  - Complex relationship between IPC and occupancy
  - Unique Scalar and Vector Registers, Uniform Access is important optimization
  - ROCm LLVM compiler exposes full compiler stack including register allocator, scheduler

- Feedback-directed Optimization
  - Best way to identify optimal code generation is to run the code
  - Can we capture appropriate state from one or more runs and use this to influence future compilation?

- Dynamic Parallelism Done Right
  - “Architected Queuing Language”:
  - Standard architecture command packet, enabled GPUs to send work to themselves or other GPUs
Some ROCm Research Opportunities
OPEN-SOURCE KERNEL DRIVER AND LIBRARIES

- **Peer-to-Peer communication**
  - Large-BAR access from other PCIe devices to all of GPU’s memory
  - Enables interesting experimentation with other open-source device drivers (FPGAs, NVME, NIC, etc)

- **Memory Management**
  - Recent GPUs include automated migration of data to GPU
  - Enables single unified pool of memory from developer perspective
  - Many heuristics and optimization opportunities for when to migrate

- **MIOpen**
  - Innovate with new algorithms, layer fusion, tuning, understanding
Where To Go Deeper On ROCm

Welcome to the ROCm Platform

We are excited to present ROCm, the first open-source HPC/Ulrascale-class platform for GPU computing that’s also programming-language independent. We are bringing the UNIX philosophy of choice, minimalism and modular software development to GPU computing. The new ROCm foundation lets you choose or even develop tools and a language run time for your application.

ROCm is built for scale; it supports multi-GPU computing in and out of server-node communication through RDMA. It also simplifies the stack when the driver directly incorporates RDMA peer-sync support.

ROCm has a rich system run time with the critical features that large-scale application, compiler and language-run-time development requires.

https://radeonopencompute.github.io/index.html
Open Source Professional Computing Solution
Foundation For Direct Access To The Hardware
Delivering Choice in Programming Models
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

© 2016 Advanced Micro Devices, Inc. All rights reserved. AMD, the AMD Arrow logo, FirePro and combinations thereof are trademarks of Advanced Micro Devices, Inc. in the United States and/or other jurisdictions. ARM is a registered trademark of ARM Limited in the UK and other countries. PCIe is a registered trademarks of PCI-SIG Corporation. OpenCL and the OpenCL logo are trademarks of Apple, Inc. and used by permission of Khronos. OpenVX is a trademark of Khronos Group, Inc. Other names are for informational purposes only and may be trademarks of their respective owners. Use of third party marks / names is for informational purposes only and no endorsement of or by AMD is intended or implied.