A C/C++ Toolchain for your GPU

Joseph Huber *(joseph.huber@amd.com)*
LLVM Developer’s Conference 2024

AMD
together we advance_

Introduction — GPGPU

- Modern GPUs have evolved into general purpose accelerators
- Provide a C/C++ toolchain that runs on the GPU
  - Trivially port applications to run on the GPU
  - Generic implementations of math / utility functions
  - Run unit tests on the GPU
  - It’s also fun
- How to port the existing C/C++ libraries to the GPU?
  - Lots of existing options

AMD
together we advance_

Targeting GPUs — CUDA / HIP

- Ubiquitous for targeting GPUs
- GPU code is manually declared
  - **`__global__`** and **`__device__`**
- Difficult to integrate into existing build systems
  - One compile job yields many files
  - Host and device compilations must both compile
- Less portable
- Compiled by the clang frontend
- Runtime implemented by builtins

NVIDIA CUDA

```
__global__ void saxpy(int n, float a, float *x, float *y) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n)
        y[i] = a * x[i] + y[i];
}
```

```
$ clang++ -x hip hip.cpp --offload-arch=gfx940 -###
-cc1 -triple amdgcn-amd-amdhsa … -fcuda-is-device -x hip
```

```
#include "device_amd_hsa.h" // HIP Runtime

ATTR size_t __ockl_get_local_id(uint dim) {
  switch (dim) {
  case 0: return __builtin_amdgcn_workitem_id_x();
  case 1: return __builtin_amdgcn_workitem_id_y();
  case 2: return __builtin_amdgcn_workitem_id_z();
  }
}
```

Targeting GPUs — OpenMP

- Uses C++ with compiler **pragmas**
  - **`#pragma omp declare target`**
- More "standard" C++
- Same issues with build systems
- Very portable
- Compiled by the clang frontend
- Uses the **same** builtins for the runtime

```
void saxpy(int n, float a, float *x, float *y) {
    #pragma omp target teams distribute parallel for
    for (int i = 0; i < n; ++i)
        y[i] = a * x[i] + y[i];
}
```

```
$ clang++ -x cpp openmp.cpp -fopenmp --offload-arch=gfx940 -###
-cc1 -triple amdgcn-amd-amdhsa … -fopenmp-is-target-device
```

```
#include <Mapping.h> // OpenMP Runtime

uint32_t getThreadIdInBlock(int32_t Dim) {
  switch (Dim) {
  case 0: return __builtin_amdgcn_workitem_id_x();
  case 1: return __builtin_amdgcn_workitem_id_y();
  case 2: return __builtin_amdgcn_workitem_id_z();
  }
}
```

Targeting GPUs — OpenCL

- Uses a more conventional approach
  - Easier to fit into existing builds
- OpenCL is fundamentally limited
  - No function pointers
  - No recursion
  - Conflicting definitions of C functions
  - OpenCLC++ has templates at least
- Also compiled through the clang frontend
- The runtime uses the **same** builtin functions
  - You get the idea

```
__kernel void
saxpy(int n, float a, __global float* x, __global float* y) {
    int i = get_global_id(0);
    if (i < n) y[i] = a * x[i] + y[i];
}
```

```
$ clang++ -x cl opencl.cl --target=amdgcn-- -mcpu=gfx940
-cc1 -triple amdgcn-amd-amdhsa … -x cl
```

```
#include <clc/clc.h> // OpenCL Runtime

_CLC_DEF _CLC_OVERLOAD size_t get_local_id(uint dim) {
  switch (dim) {
  case 0: return __builtin_amdgcn_workitem_id_x();
  case 1: return __builtin_amdgcn_workitem_id_y();
  case 2: return __builtin_amdgcn_workitem_id_z();
  }
}
```

Targeting GPUs — C/C++

- Why bother porting anything in the first place?
- All of these languages are just different versions of **clang**
  - We can just target C/C++ directly without a GPU language
- Use **`--target=amdgcn-amd-amdhsa`** to invoke the clang target
- Our linker is **ld.lld** so we can use LTO and static libraries
  - Don’t specify **-mcpu=** and we can get generic LLVM-IR

Targeting GPUs — ISO C/C++

```
void matmul(float *A, float *B, float *C, int N) {
  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) {
      float sum = 0.0;
      for (int k = 0; k < N; ++k)
        sum += A[i * N + k] * B[k * N + j];
      C[i * N + j] = sum;
    }
  }
}
```

Targeting GPUs — C/C++ Extensions

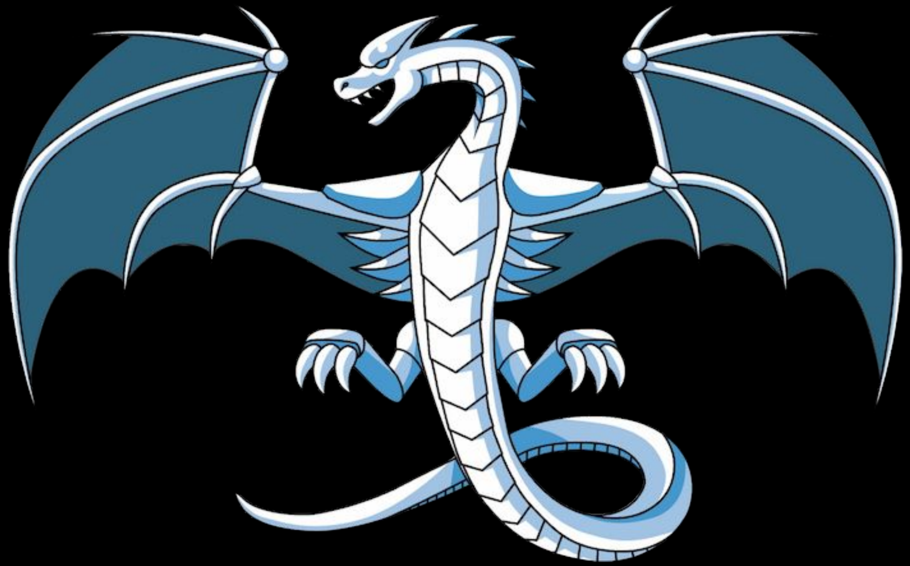
```
[[clang::amdgpu_kernel]] void matmul(float *A, float *B, float *C, int N) { // Target calling convention
  static [[clang::address_space(3)]] float A_s[TILE][TILE]; // Target address space for __shared__
  static [[clang::address_space(3)]] float B_s[TILE][TILE]; // Target address space for __shared__

  int bx = __builtin_amdgcn_workgroup_id_x(); int tx = __builtin_amdgcn_workitem_id_x(); // Builtin functions
  int by = __builtin_amdgcn_workgroup_id_x(); int ty = __builtin_amdgcn_workitem_id_y(); // Builtin functions

  for (int ph = 0; ph < N / TILE; ++ph) {
    A_s[ty][tx] = A[(by * TILE + ty) * N + ph * TILE + tx];
    B_s[ty][tx] = B[(ph * TILE + ty) * N + bx * TILE + tx];
    __builtin_amdgcn_s_barrier(); // Builtin functions
    float sum = 0.0f;
    for (int k = 0; k < TILE; ++k)
      sum += A_s[ty][k] * B_s[k][tx];
    __builtin_amdgcn_s_barrier(); // Builtin functions
  }
  C[(by * TILE + ty) * N + bx * TILE + tx] = sum;
}
```

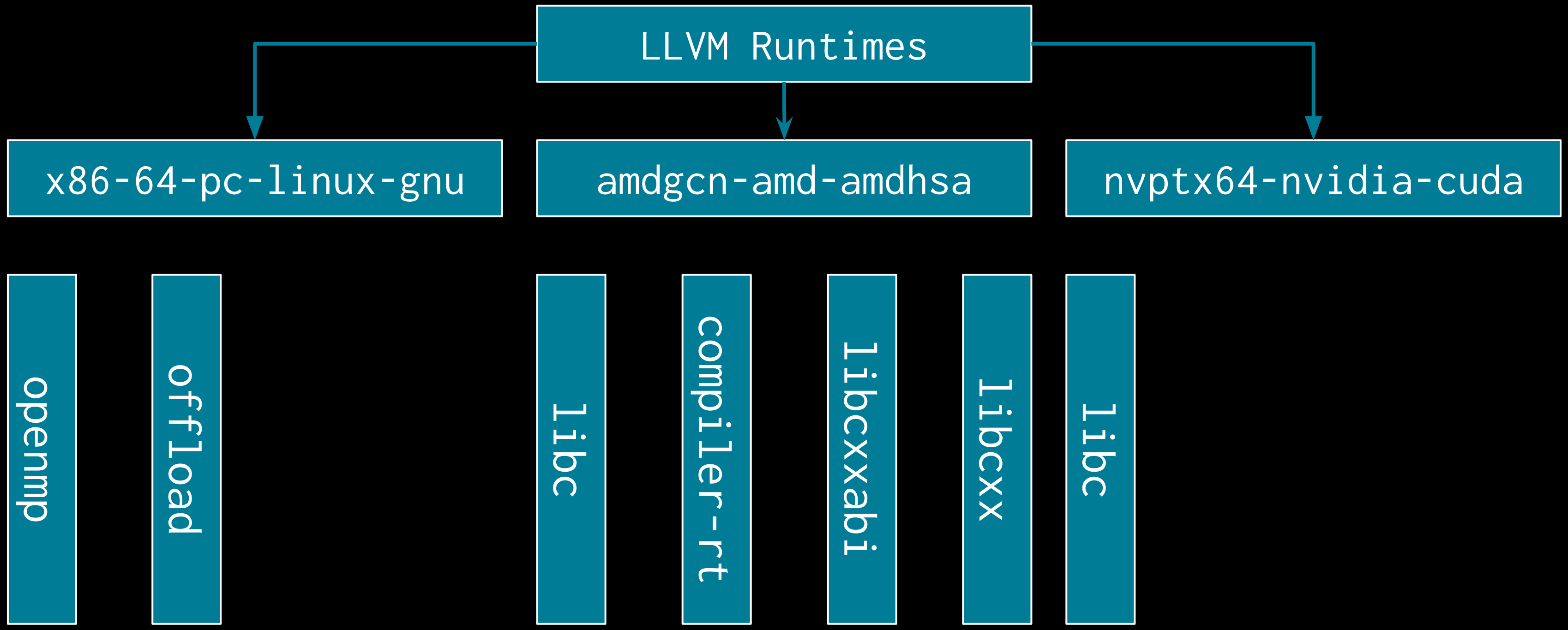
Cross Compiling — C/C++

- This is just cross compiling!
  - Plenty of build systems support this natively
- A complete compiler toolchain has…
  - A compiler frontend ✓
  - An assembler ✓
  - A linker ✓
  - Runtime libraries ✗
- Lets port the LLVM C and C++ runtimes to the GPU!



LLVM Runtimes — Introduction

- Bootstraps multiple libraries using the just-built clang
- Create the libraries for multiple targets
  - -DLLVM_RUNTIME_TARGETS=default;amdgcn-amd-amdhsa;nvptx64-nvidia-cuda
- CMake arguments can be passed individually to each runtime target
  - -DRUNTIMES_<triple>_<var>=<value>



Clang/LLVM — Multilibs

- Each runtime gets its own directory
  - `-DLLVM_ENABLE_PER_TARGET_RUNTIME_DIR=ON`
- Use the GPU target to create the toolchain
- Clang will point to the appropriate folder
  - Only need to pass **-lm -lc …**
- Now let’s actually build them

```
□ install
├── □ bin
│   ├── amdhsa-loader
│   └── clang
├── □ include
│   ├── □ amdgcn-amd-amdhsa
│   │   ├── □ c++
│   │   │   └── □ v1
│   │   │       └── __config_site
│   │   └── <libc headers>
│   └── □ c++
│       └── □ v1
│           └── <libc++ headers>
└── □ lib
    ├── □ amdgcn-amd-amdhsa
    │   ├── crt1.o
    │   ├── libc++.a
    │   ├── libc.a
    │   ├── libc++abi.a
    │   └── libm.a
    └── □ clang
        └── □ 20
            └── □ lib
                └── □ amdgcn-amd-amdhsa
                    └── libclang_rt.builtins.a
```

LLVM Runtimes — LLVM libc

- The C library is the basis of other applications
- The LLVM C library is highly configurable
    - Just enable the functions we want and compile them
- System calls are handled through **Remote Procedure Calls**
    - Client / server protocol communicating through mutual exclusion on unified memory
- See my talk last year for more detailed information
    - **The LLVM C Library for GPUs**

```
set(TARGET_LIBC_ENTRYPOINTS
  …
  # stdio.h entrypoints
  libc.src.stdio.printf
  libc.src.stdio.vprintf
  libc.src.stdio.fprintf
  libc.src.stdio.vfprintf
  libc.src.stdio.snprintf
  libc.src.stdio.sprintf
  libc.src.stdio.vsnprintf
  libc.src.stdio.vsprintf
  libc.src.stdio.asprintf
  libc.src.stdio.vasprintf
  libc.src.stdio.sscanf
  libc.src.stdio.vsscanf
  libc.src.stdio.fscanf
  …
)
```

LLVM Runtimes — LLVM libc

- Make the GPU look like a normal hosted target
- Standard **libc** implementations use a startup object (i.e. **crt1.o**) to call the main function
  - Just write one for the GPU
- Cross compiling emulators run tests
  - Write one for the GPU using the GPU runtime

```
void call_init_callbacks(int argc, char **argv, char **env) {
  /* Call global constructors. */
}

void call_fini_callbacks() { /* Call global destructors. */ }

extern "C" {
[[gnu::visibility("protected"), clang::amdgpu_kernel]] void
_begin(int argc, char **argv, char **env, void *in, void *out)
{
  atexit(&call_fini_callbacks);
  call_init_callbacks(argc, argv, env);
}

[[gnu::visibility("protected"), clang::amdgpu_kernel]] void
_start(int argc, char **argv, char **envp, int *ret) {
  __atomic_fetch_or(ret, main(argc, argv, envp),
                    __ATOMIC_RELAXED);
}

[[gnu::visibility("protected"), clang::amdgpu_kernel]] void
_end(int retval) {
  exit(retval);
}
}
```

LLVM Runtimes — Compiler-RT

- Provides a runtime library that Clang will implicitly call
  - `libclang_rt.builtins.a`
- Porting is very straightforward
  - Just enable the runtimes build and set a few flags
- This gives us a functional C toolchain for the GPU
- Now we can compile some more interesting things for the GPU

```c
#include <stdio.h>

int main(int argc, char **argv) {
  fprintf(stdout, "%s %d\n", argv[0] , __builtin_amdgcn_workgroup_id_x());
}
```

```
$> clang app.c --target=amdgcn-amd-amdhsa -flto -mcpu=native -lc -lm -lclang_rt.builtins crt1.o -nogpulib
$> amdhsa-loader --blocks 3 ./a.out
./a.out 1
./a.out 0
./a.out 2
```

LLVM Runtimes — libc++

- Build the C++ libraries on top of the C toolchain
- Disable some things we can't handle right now
  - Threads and Filesystem support mostly
  - Lots of config options so we provide a cache file
    - `-DRUNTIMES_amdgcn-amd-amdhsa_CACHE_FILES=libcxx/caches/AMDGPU.cmake`
- Other approaches exist but typically require a separate include path

LLVM Runtimes — libc++ example

```cpp
#include <...>

int main(int argc, char **argv) {
  std::mt19937 generator(__builtin_amdgcn_workitem_id_x());
  std::uniform_int_distribution<int> dist(1, 100);

  std::vector<int> vec(8);
  std::ranges::generate(vec, [&]() { return dist(generator); });

  for (int x : vec)
    std::cout << x << " ";
}
```

```
$> clang++ app.cpp --target=amdgcn-amd-amdhsa -flto -mcpu=native -lc -lm -lc++ -lc++abi \
     crt1.o -lclang_rt.builtins -stdlib=libc++ -nogpulib -fno-exceptions
$> amdhsa-loader ./a.out
45 48 65 68 68 10 84 22
```

AMD
together we advance_

LLVM Runtimes — Testing libc++

- libc++ already has support for custom test executors
- We can run the test suite on the GPU for free
  - Found a lot of obscure compiler bugs
- Still a lot of failures but it's a start

```
$> ninja -C
runtimes/runtimes-amdgcn-amd-amdhsa-bins check-cxx

Testing Time: 1092.29s
Total Discovered Tests: 9749
Unsupported   : 2743 (28.14%)
Passed        : 6804 (69.79%)
Expectedly Failed:   48 (0.49%)
Failed        :  154 (1.58%)
```


AMD
together we advance_

LLVM Runtimes — Bringing to Offloading Languages

- Must specify which definitions are available to the device
- We export a static library in the target-specific directory
- Just link it with the device-side compilation, the compiler knows where to find it
  - All these languages use the same linker and toolchain remember?

```
#include <iostream>

#pragma omp declare target to(std::cout)

int main() {
#pragma omp target
  std::cout << "Hello World\n";
}
```

```
$> clang++ hello.cpp -fopenmp --offload-arch=native -Xoffload-linker -lc++ \
     -Xoffload-linker -lc++abi -Xoffload-linker -lc -fno-exceptions -stdlib=libc++
$> ./a.out
Hello World
```

LLVM Runtimes — Bringing it all Together

- We can build a functional C/C++ toolchain that targets the GPU
- The magic spell to summon it

```
$> cd llvm-project  # The llvm-project checkout
$> mkdir build
$> cd build
$> cmake ../llvm -G Ninja                                           \
   -DLLVM_ENABLE_PROJECTS="clang;lld"                               \
   -DCMAKE_BUILD_TYPE=<Debug|Release>   \ # Select build type
   -DCMAKE_INSTALL_PREFIX=<PATH>        \ # Where the libraries will live
   -DRUNTIMES_amdgcn-amd-amdhsa_CACHE_FILES="./libcxx/cmake/caches/AMDGPU.cmake"    \
   -DRUNTIMES_nvptx64-nvidia-cuda_CACHE_FILES="./libcxx/cmake/caches/NVPTX.cmake"   \
   -DRUNTIMES_amdgcn-amd-amdhsa_LLVM_ENABLE_RUNTIMES="libc;compiler-rt;libcxx;libcxxabi"   \
   -DRUNTIMES_nvptx64-nvidia-cuda_LLVM_ENABLE_RUNTIMES="libc;compiler-rt;libcxx;libcxxabi" \
   -DLLVM_RUNTIME_TARGETS="default;amdgcn-amd-amdhsa;nvptx64-nvidia-cuda"
$> ninja install
```

Challenges

- How to handle **<mutex>** and **<thread>**?
  - GPUs have limited forward progress guarantees
  - Cooperative launches can probably help
- Compile times!
  - Really bad because we need LTO for both performance and portability
- HIP/CUDA/OpenMP must compile the same headers on the CPU/GPU
  - Things must be explicitly declared on the GPU
- What if we copy something from the CPU to the GPU?
  - std::vector calls realloc on the GPU, instant segfault
  - std::mutex calls futex, GPU cannot notify the thread
- Not as many useful functions in **libc++**
  - Huge resource requirements when compared to **libc**
- Should we provide exceptions?
- NVIDIA is limited by PTX and nvlink
- …

¯_(ツ)_/¯

Running DOOM on the GPU

https://github.com/jhuber6/doomgeneric

DOOM — Demo

AMD
together we advance_

Disclaimer

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. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. 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.

THIS INFORMATION IS PROVIDED 'AS IS." 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 NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, 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.

AMD