LLVM C Library for GPUs

Joseph Huber (joseph.huber@amd.com)
LLVM Developer’s Conference 2023
What is the LLVM C library for GPUs

Using libc for GPUs

Table of Contents
- Building the GPU library
- Usage

Building the GPU library

LLVM’s libc GPU support must be built with an up-to-date clang compiler due to heavy reliance on clang’s GPU support. This can be done automatically using the LLVM_ENABLE_RUNTIMES=libc option. To enable libc for the GPU, enable the LIBC_GPU_BUILD option. By default, libcgpu.a will be built using every supported GPU architecture. To restrict the number of architectures build, either set LIBC_GPU_ARCHITECTURES to the list of desired architectures manually or use native to detect the GPUs on your system. A typical cmake configuration will look like this:

```
$> cd llvm-project  # The llvm-project checkout
$> mkdir build
$> cd build
$> cmake ../llvm -G Ninja
  -DLLVM_ENABLE_PROJECTS="clang;lld;compiler-rt"
  -DLLVM_ENABLE_RUNTIMES="libc,operand"
  -DCMAKE_BUILD_TYPE=Debug|Release
  -DLIBC_GPU_BUILD=ON
  -DLIBC_GPU_ARCHITECTURES=all
  -DCMAKE_INSTALL_PREFIX=<PATH>
  # Where ‘libcgpu.a’ will live
$> ninja install
```

- Build of the LLVM C library targeting GPUs
  - [https://libc.llvm.org/gpu](https://libc.llvm.org/gpu)
  - `-DLIBC_GPU_BUILD=ON`
- Supports AMD and NVIDIA GPUs
Why write a C library for the GPU

• Initially wanted a portable implementation of `printf`
  • Might as well do everything else while we’re at it
• Trivially port CPU applications and tests to the GPU
• Portable GPU math functions in `libmgpu.a`
• Basis for more libraries, i.a. `libc++`
LLVM C Library — Language Support

- Exported as `libcgpu.a` and `libmgpu.a`
- Compatible with OpenMP, C++*, CUDA*, HIP*
- Support for most common libc functions
  - [https://libc.llvm.org/gpu/support.html](https://libc.llvm.org/gpu/support.html)
What’s Working — Compiling and Running

```c
#include <omp.h>
#include <stdio.h>

int main() {
    #pragma omp target
    #pragma omp parallel num_threads(4)
    fprintf(stdout, "Thread id: ", __builtin_amdgcn_workitem_id_x());
}
```

```c
#include <omp.h>
#include <stdio.h>

int main() {
    #pragma omp target
    #pragma omp parallel num_threads(4)
    fprintf(stdout, "Thread id: ", omp_get_thread_num());
}
```

$ clang++ openmp.cpp -fopenmp --offload-arch=native \ -lcgpu
$ ./a.out
Thread id: 0
Thread id: 1
Thread id: 2
Thread id: 3

$ clang++ direct.cpp --target=amdgcn-amd-amdhsa \ -mcpu=native -lc crt1.o
$ ./amdhsa_loader --threads 4 a.out
Thread id: 0
Thread id: 1
Thread id: 2
Thread id: 3
## What’s Working — Build Bots

<table>
<thead>
<tr>
<th>Build request</th>
<th>Build times</th>
<th>Success Rate</th>
<th>Owners</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Builds:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>#1</td>
<td>23 minutes</td>
<td>8 minutes</td>
<td>Peter Kasting <a href="mailto:pkeasting@chromium.org">pkeasting@chromium.org</a></td>
</tr>
<tr>
<td>#2</td>
<td>31 minutes</td>
<td>8 minutes</td>
<td>Mark de Wever <a href="mailto:kronq@xs4all.nl">kronq@xs4all.nl</a></td>
</tr>
<tr>
<td>#3</td>
<td>42 minutes</td>
<td>11 minutes</td>
<td>Simon Pilgrim <a href="mailto:llvm-dev@redging.me.uk">llvm-dev@redging.me.uk</a></td>
</tr>
<tr>
<td>#4</td>
<td>an hour</td>
<td>8 minutes</td>
<td>Jay Foad <a href="mailto:jay.foad@amd.com">jay.foad@amd.com</a></td>
</tr>
<tr>
<td>#5</td>
<td>2 hours</td>
<td>9 minutes</td>
<td>Jay Foad <a href="mailto:jay.foad@amd.com">jay.foad@amd.com</a></td>
</tr>
<tr>
<td>#6</td>
<td>2 hours</td>
<td>8 minutes</td>
<td>Michael Liao <a href="mailto:michael.liiao@gmail.com">michael.liiao@gmail.com</a></td>
</tr>
<tr>
<td>#7</td>
<td>3 hours</td>
<td>8 minutes</td>
<td>Simon Pilgrim <a href="mailto:llvm-dev@redging.me.uk">llvm-dev@redging.me.uk</a></td>
</tr>
</tbody>
</table>

**LLVM Staging**

**Builds:**
- **AMDGPU builder targeting** **gfx906 architecture**
- **NVIDIA builder targeting the** **sm_75 architecture**
- **NVIDIA builder targeting the** **sm_60 architecture**
The LLVM C Library — Overview

- Written entirely in freestanding C++
- Designed to be easily decomposed and ported
  - Generates custom system headers (e.g. string.h)
  - Defines which functions are supported (e.g. memcpy)
  - Configured through function specifications (e.g. the C standard)

```cpp
set(TARGET_PUBLIC_HEADERS
    libc.include.ctype
    libc.include.string
    libc.include.fenv
    libc.include.errno
    libc.include.stdlib
)

set(TARGET_LIBC_ENTRYPOINTS
    # ctype.h entrypoints
    libc.src.ctype.isalnum
    libc.src.ctype.isspace
    ...
    # string.h entrypoints
    libc.src.string.memccpy
    libc.src.string.memcmp
    libc.src.string.memcpy
    ...
)

HeaderSpec CType = HeaderSpec<
"ctype.h",
[], // Macros
[], // Types
[
    FunctionSpec<
        "isalnum",
        RetValSpec<IntType>,
        [ArgSpec<IntType>]
    >,
]
>;
```
# The LLVM C Library — Targeting GPUs

- GPUs are treated as regular targets
  - `--target=amdgcn-amd-amdhsa -mcpu=gfx90a`
- Architecture specific operations handled with macros and builtin functions
  - `__AMDGPU__` and `__NVPTX__`
- Packaged as a “bundled” static library (just link it!)

```c
#include "src/string/strcmp.h"
#include "src/__support/common.h"

namespace LIBC_NAMESPACE {

template <typename Comp>
constexpr static int strcmp_implementation(
  const char *left, const char *right, Comp &comp) {
  for (; *left && !comp(*left, *right); ++left, ++right) {
    return comp(reinterpret_cast<unsigned char *>(left),
                reinterpret_cast<unsigned char *>(right));
  }

  LLVM_LIBC_FUNCTION(int, strcmp, (const char *left,
                                 const char *right)) {
    auto comp = []((char l, char r) -> int { return l - r; });
    return strcmp_implementation(left, right, comp);
  }
} // namespace LIBC_NAMESPACE
```
The LLVM C Library — Targeting GPUs
The LLVM C Library — GPU Loader Utility and Startup

- Write a loader utility to launch the GPU program
  - amdhsa_loader and nvptx_loader
- Standard libc implementations use a startup object (i.e. crt1.o)
  - Just write one for the GPU
- Export kernels that handle global ctors / dtors and call the main function

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

void call_fini_array_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_array_callbacks);
        call_init_array_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);
    }
}
```
Now we can run anything on the GPU right?
The LLVM C Library — Syscalls

- Some functions require intervention from the operating system
  - E.g. `exit`, `printf`, or `malloc`
- The GPU doesn’t have a usable operating system
  - Treat the host machine as the operating system
- Implement **Remote Procedure Calls** to function as syscalls
Remote Procedure Calls — Implementation

- GPUs support device-accessible host memory
  - E.g. hipMallocHost or cudaMallocHost
- Atomically swap ownership of a fixed size buffer between a client and server
  - Each process has a write-only outbox and a read-only inbox to indicate ownership of the buffer
- Exposes some primitive operations
  - Wait for ownership
  - Use the buffer
  - Give away ownership
Remote Procedure Calls — Implementation

- Most GPUs support unified memory addressing
  - E.g. hipMallocHost or cudaMallocHost
- Atomically swap ownership of a fixed size buffer between a client and server
  - Each process has a write-only outbox and a read-only inbox to indicate ownership of the buffer
- Exposes some primitive operations
  - Wait for ownership
  - Use the buffer
  - Give away ownership
- Add a header for the desired “syscall”
Remote Procedure Calls — Handling multiple threads

- Abstract access to the buffer into a port
- Provide mutual exclusion on the port using a `test` and `set` lock
- We can open a port if the process owns the buffer and the lock is available
Remote Procedure Calls — GPU Considerations

- The smallest unit of independent parallelism on GPUs is not a thread
  - SIMD execution on a warp / wave
- The interface needs to handle a whole warp or wavefront at a time
  - Supports partial / masked usage
Remote Procedure Calls — Implementation

- Provide multiple ports to increase concurrent access
  - Required to prevent deadlocks on some GPU hardware

Number of Ports
The LLVM C Library — Remote Procedure Calls

- Simplify this interface into arbitrary* `send` and `recv` of packets
- Requires a server on the host listening to the ports
- Provides the ability to call host functions from the GPU

```c
#include "rpc.h"

// Running on GPU.
uint64_t increment_on_cpu(uint64_t count) {
    rpc::Client::Port port = rpc::client.open<RPC_INC>();
    port.send([=](uint64_t *buffer) { *buffer = count; });
    port.recv([&](uint64_t *buffer) { count = *buffer; });
    port.close();
    return count;
}

// Running on CPU.
int rpc_server() {
    rpc::Server::Port port = rpc::server.open();
    switch (port.opcode) {
    case RPC_INC: {
        port.recv([&](uint64_t *buffer) { *buffer += 1; });
        port.send([](uint64_t *) { /* no-op */ });
    } break;
    default: break;
    }
    port.close();
    return 0;
}
```
Remote Procedure Calls — Overhead

- Comparing the latency of a no-op function call on the host via RPC versus a kernel launch
  - Calling an RPC function is roughly equivalent to launching a kernel
The LLVM C Library — Testing

- The existing LLVM C library tests run on a CPU in a self-hosted environment
- Execute these tests directly on the GPU as if we were cross compiling
- Running 125 GPU tests across three buildbots
  - Looking into running parts of the LLVM test suite on the GPU as well

```c
#include "src/string/strcmp.h"
#include "test/UnitTest/Test.h"

TEST(LlvmLibcStrCmpTest,
     CapitalizedLetterShouldNotBeEqual) {
    const char *s1 = "abcd";
    const char *s2 = "abCd";
    int result = __llvm_libc::strcmp(s1, s2);
    // 'c' - 'C' = 32.
    ASSERT_EQ(result, 32);

    // Verify operands reversed.
    result = __llvm_libc::strcmp(s2, s1);
    // 'C' - 'c' = -32.
    ASSERT_EQ(result, -32);
}
```
The LLVM C Library — Pain Points

- Currently cannot support `thread_local` keywords
  - Implementing `errno` or `rand` is difficult without it
- Linking `libc` with LTO is problematic
  - Functions cannot be inlined without `-fno-builtin`
- Linking `libc` with LTO is problematic
  - Functions cannot be inlined without `-fno-builtin`
- Linking `libc` with LTO is problematic
  - Functions cannot be inlined without `-fno-builtin`
- Linking `libc` with LTO is problematic
  - Functions cannot be inlined without `-fno-builtin`
- CMake cannot build the CPU `libc` and GPU `libc` at the same time
- The `libc` headers and host headers need to match and cooperate
- The NVIDIA toolchain
  - No static library support in `nvlink`
  - Cannot emit variables to sections, the `ptx` keyword `section` only works in debug mode
- Multi-architecture support should be done with a single bitcode file
  - We currently just build the entire library for each supported architecture
LibC for GPUs — Conclusion

- Built the LLVM C library targeting GPU
  - Compiling freestanding C++ for the GPU works very well and is portable
- Created `libcgpu.a` and `libmgpu.a` that is fully integrated into OpenMP offloading
  - Using with CUDA / HIP is functional but opt-in
- Parallel and extensible RPC interface for host execution
- Available now upstream and tested via buildbots
- Can compile and run standard C++ directly on the GPU
Thanks to

- Jon Chesterfield
- Johannes Doerfert
- Brian Sumner
- Tian Shilei
- Siva Chandra
- Tue Ly
- Michael Jones