

Viability and performance characteristics of using HIPify for CUDA projects

Why would that be needed?

CUDA is the most popular GPU computing platform. But it is also completely proprietary, supporting only Nvidia GPUs. Even though CUDA is based on Open Source LLVM compiler infrastructure, and LLVM can compile CUDA even without NVCC that is based on it, it still needs proprietary CUDA SDK.^[1]

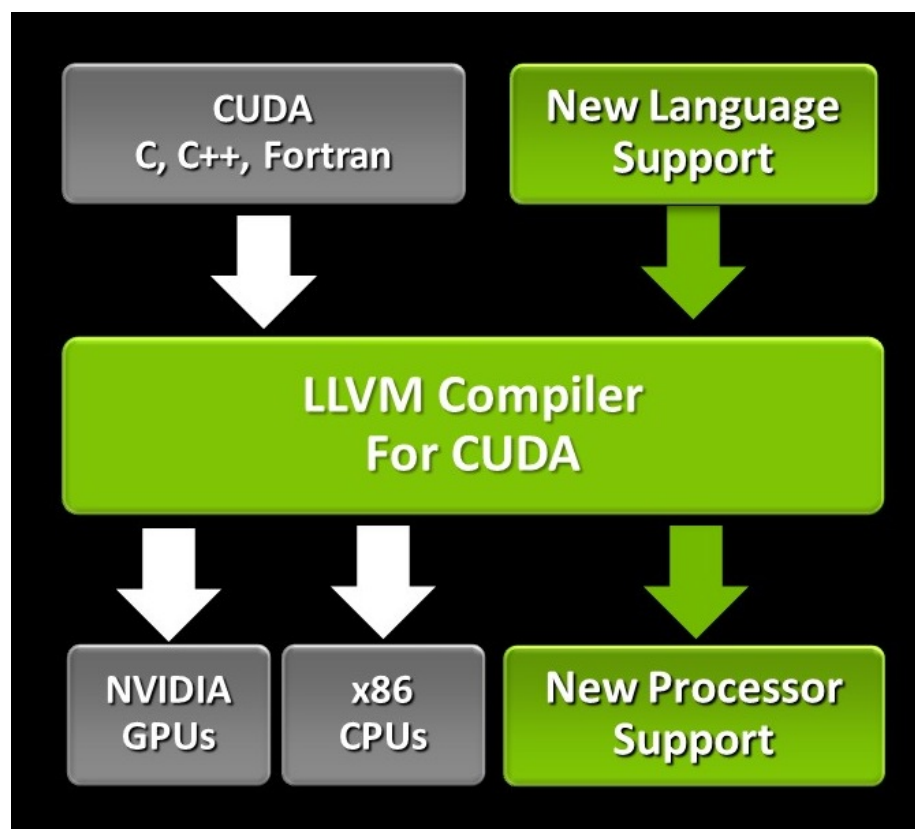


Figure 1: CUDA LLVM structure

There has been many approaches to make CUDA code run on other GPUs, but I will focus on the most mature HIP (Heterogeneous Interface for Portability) and ROCm software stack, which are both fully open source.

What is HIP?

HIP is a portable source code that can be compiled for both AMD and Nvidia GPUs using HCC or NVCC respectively. HIP code can be extremely similar to CUDA C++, with only CUDA function names being replaced with HIP ones.

HCC is based on LLVM same as NVCC, I assume not only because it is so powerful, but also because LLVM already had support for CUDA, so AMD could reuse a lot of that code. This means LLVM is now used for CUDA, while AMD and Intel use it for both graphics (shader) and compute compiler. This will hopefully mean more vendor compatibility in the future.

What is HIPify?

HIPify-clang is a tool to convert CUDA to Portable C++ Code. It is utilising LLVM's CUDA support and "It translates CUDA source into an abstract syntax tree, which is traversed by transformation matchers. After applying all the matchers, the output HIP source is produced."¹

From the start this approach could mean some incompatibility, because even LLVM themselves are warning that NVCC is parsing code slightly differently than mainline LLVM.²

This approach supports all the CUDA versions LLVM does, so it is only a few months behind what Nvidia releases, and "even very complicated constructs will be parsed successfully".³

But I found out this approach brings a lot of its own incompatibilities with it. It is a standalone package that doesn't require the rest of ROCm stack, but it does require a new LLVM (the newer the higher CUDA support), as well as a full CUDA SDK installation.

That fact immediately breaks the idea of fully open source running of CUDA code. Also because of its license many Linux distributions can't package CUDA and include it in its repositories, including mine. I tried in many ways manually extracting the CUDA installer, to get the libraries and headers. But it turns out that HIPify still uses LLVM's full CUDA detection, which means LLVM checks all components of CUDA and if they were properly installed, including e.g. device libraries.⁴

After a lot of trying I had to use a different OS for just translation to HIP, I used Ubuntu 20.04 with CUDA SDK straight from Nvidia, and then I could use that translated code on my OS with fully build from source software stack.

¹<https://github.com/ROCm-Developer-Tools/HIPIFY>

²<https://www.llvm.org/docs/CompileCudaWithLLVM.html#dialect-differences-between-clang-and-nvcc>

³<https://github.com/ROCm-Developer-Tools/HIPIFY>

⁴<https://github.com/llvm/llvm-project/blob/7e629e4e888262abd8f076512b252208acd72d62/clang/lib/Driver/ToolChains/Cuda.c>

```
~/Downloads/HIPIFY-rocm-5.4.2/dist/hipify-clang vectorAdd.cu -I ~/Downloads/cuda-samples-11
```

Differences between HIP and CUDA C++ language

HIP is very similar to CUDA. After I converted vectorAdd.cu all that got changed was all the occurrences of CUDA includes or function names got changed to HIP.

So for example `#include <cuda_runtime.h>` to `#include <hip/hip_runtime.h>`, `err = cudaMalloc((void **)&d_A, size);` to `err = hipMalloc((void **)&d_A, size);`, etc. The whole rest was the same.

All the libraries need to be converted too, you can use HIPify for that too, and for the big CUDA libraries AMD has created their own versions, optimised for their GPUs, to the programmer the functions are very similar but unfortunately not the same.

Incompatibilities when converting using HIPify

Only now we can deal with actual conversion and its shortcomings. I have decided to use Nvidia's CUDA Samples⁵ because they offer a wide range of features to be tested, as well as I expected they would be better supported because they are the official reference code. And yet I still encountered many problems with them

- You can't specify which c++ version it is using for converting, like you can when compiling CUDA, but you can when compiling. This meant I had to replace c++ 11's `sprintf_s` with something supported in previous version, even when HIPify should be using C++ 11 by default

```
/tmp/deviceQuery.cpp-040229.hip:308:3: error: use of undeclared identifier 'sprintf_s'
sprintf_s(cTemp, 10, "%d.%d", driverVersion / 1000,
```

- some device properties from CUDA are not implemented in HIP's `hipDeviceProp_t`

```
deviceQuery.cpp.hip:180:20: error: no member named 'maxTexture1DLayered' in 'hipDeviceProp_t'
    deviceProp.maxTexture1DLayered[0], deviceProp.maxTexture1DLayered[1]);
```

```
    ~~~~~
    maxTexture1DLinear
```

```
/usr/include/hip/hip_runtime_api.h:122:9: note: 'maxTexture1DLinear' declared here
    int maxTexture1DLinear;    ///< Maximum size for 1D textures bound to linear memory
    ~
```

```
deviceQuery.cpp.hip:180:39: error: subscripted value is not an array, pointer, or vector
    deviceProp.maxTexture1DLayered[0], deviceProp.maxTexture1DLayered[1]);
```

```
    ~~~~~~
```

```
deviceQuery.cpp.hip:180:55: error: no member named 'maxTexture1DLayered' in 'hipDeviceProp_t'
    deviceProp.maxTexture1DLayered[0], deviceProp.maxTexture1DLayered[1]);
```

⁵<https://github.com/NVIDIA/cuda-samples/archive/refs/tags/v11.6.tar.gz>

```

~~~~~~
maxTexture1DLinear
/usr/include/hip/hip_runtime_api.h:122:9: note: 'maxTexture1DLinear' declared here
int maxTexture1DLinear;    ///< Maximum size for 1D textures bound to linear memory
~
deviceQuery.cpp.hip:180:74: error: subscripted value is not an array, pointer, or vector
deviceProp.maxTexture1DLayered[0], deviceProp.maxTexture1DLayered[1]);
~~~~~~
deviceQuery.cpp.hip:184:20: error: no member named 'maxTexture2DLayered' in 'hipDeviceProp_t'
deviceProp.maxTexture2DLayered[0], deviceProp.maxTexture2DLayered[1],
~~~~~~
maxTexture1DLinear
/usr/include/hip/hip_runtime_api.h:122:9: note: 'maxTexture1DLinear' declared here
int maxTexture1DLinear;    ///< Maximum size for 1D textures bound to linear memory
~
deviceQuery.cpp.hip:184:39: error: subscripted value is not an array, pointer, or vector
deviceProp.maxTexture2DLayered[0], deviceProp.maxTexture2DLayered[1],
~~~~~~
deviceQuery.cpp.hip:184:55: error: no member named 'maxTexture2DLayered' in 'hipDeviceProp_t'
deviceProp.maxTexture2DLayered[0], deviceProp.maxTexture2DLayered[1],
~~~~~~
maxTexture1DLinear
/usr/include/hip/hip_runtime_api.h:122:9: note: 'maxTexture1DLinear' declared here
int maxTexture1DLinear;    ///< Maximum size for 1D textures bound to linear memory
~
deviceQuery.cpp.hip:184:74: error: subscripted value is not an array, pointer, or vector
deviceProp.maxTexture2DLayered[0], deviceProp.maxTexture2DLayered[1],
~~~~~~
deviceQuery.cpp.hip:185:20: error: no member named 'maxTexture2DLayered' in 'hipDeviceProp_t'
deviceProp.maxTexture2DLayered[2]);
~~~~~~
maxTexture1DLinear
/usr/include/hip/hip_runtime_api.h:122:9: note: 'maxTexture1DLinear' declared here
int maxTexture1DLinear;    ///< Maximum size for 1D textures bound to linear memory
~
deviceQuery.cpp.hip:185:39: error: subscripted value is not an array, pointer, or vector
deviceProp.maxTexture2DLayered[2]);
~~~~~~
deviceQuery.cpp.hip:192:23: error: no member named 'sharedMemPerMultiprocessor' in 'hipDeviceProp_t'
deviceProp.sharedMemPerMultiprocessor);
~~~~~~
maxSharedMemoryPerMultiProcessor
/usr/include/hip/hip_runtime_api.h:114:12: note: 'maxSharedMemoryPerMultiProcessor' declared here
size_t maxSharedMemoryPerMultiProcessor;    ///< Maximum Shared Memory Per Multiprocessor
~
deviceQuery.cpp.hip:214:21: error: no member named 'deviceOverlap' in 'hipDeviceProp_t'

```

```

        (deviceProp.deviceOverlap ? "Yes" : "No"), deviceProp.asyncEngineCount);
        ~~~~~ ^
deviceQuery.cpp.hip:214:63: error: no member named 'asyncEngineCount' in 'hipDeviceProp_t'
        (deviceProp.deviceOverlap ? "Yes" : "No"), deviceProp.asyncEngineCount);
        ~~~~~ ^
deviceQuery.cpp.hip:222:23: error: no member named 'surfaceAlignment' in 'hipDeviceProp_t'
        deviceProp.surfaceAlignment ? "Yes" : "No");
        ~~~~~ ^
deviceQuery.cpp.hip:231:23: error: no member named 'unifiedAddressing' in 'hipDeviceProp_t'
        deviceProp.unifiedAddressing ? "Yes" : "No");
        ~~~~~ ^
deviceQuery.cpp.hip:235:23: error: no member named 'computePreemptionSupported' in 'hipDeviceProp_t'
        deviceProp.computePreemptionSupported ? "Yes" : "No");
        ~~~~~ ^

```

- Converting helper_cuda.h was the hardest as HIPify can't handle cuBLAS, cuRAND, cuFFT, cuPARSE, even though those APIs should be supported⁶
- It ignores preprocessor statements, so for example when a file contains `#ifdefs` it will throw errors about e.g. redefinition
- Oddities, like `__DRIVER_TYPES_H__` never being defined by HIP, nor HIPify will it to `HIP_INCLUDE_HIP_DRIVER_TYPES_H`. This down the line makes compilation not work
- There is no easy way to convert makefiles, all have to be done manually

The ROCm stack

ROCm

The whole of the stack is Open Source, from kernel driver up, most released under MIT or GPL-2 license. The only proprietary part is the GPU firmware.

You can of course download packages for supported OSes from AMD, but the whole differentiating factor from CUDA is it being open, so let's review how easy it is to compile it from source and package it for a distribution.

Building and packaging ROCm from source

<https://github.com/JacekJagosz/rocm-Solus>

Problems when compiling

```
hipcc vectorAdd.cu.hip -o vectorAddRX580 --rocm-device-lib-path=/usr/lib64/amdgc/bitcode/ -
```

```
hipcc vectorAdd.cu.hip -o vectorAddRX580v2 --rocm-device-lib-path=/usr/lib64/amdgc/bitcode/ -
```

```
hipcc commonKernels.cu.hip helperFunctions.cpp.hip matrixMultiplyPerf.cu.hip --rocm-device-lib-path=/usr/lib64/amdgc/bitcode/ -
```

⁶<https://github.com/ROCm-Developer-Tools/HIPIFY#supported-cuda-apis>

- Even odder things

```
matrixMultiplyPerf.cu.hip:318:23: error: no matching function for call to 'hipHostGetDevice'
    checkCudaErrors(hipHostGetDevicePointer(&dptra, hptrA, 0));
```

- Can't find an identifier for some reason:

```
error: use of undeclared identifier 'findCudaDevice'
int dev = findCudaDevice(argc, (const char **)argv);
```

Problems when running

- cudaProfilerStart and cudaProfilerStop are deprecated but exposed by torch.cuda.cudart(). HIP has corresponding functions stubbed out, hipProfilerStart and hipProfilerStop, but they return hipErrorNotSupported, which causes the program to stop. I think if they are stub functions anyways they should return success. `code=801(hipErrorNotSupported)` "hipProfilerStart()": <https://github.com/pytorch/pytorch/pull/82778>
It is CUDA's problem too, as even their own samples are using a deprecated API, which is still implemented in CUDA but AMD didn't bother.

Performance

First APU results, 37% performance of the RTX 4000, with only 27.4% of the theoretical TFLOPs, and about 10x smaller power consumption (0,4482,1802/7.119) matrixMul:

```
MatrixA(320,320), MatrixB(640,320)
Computing result using CUDA Kernel...
done
Performance= 216.03 GFlop/s, Time= 0.607 msec, Size= 131072000 Ops, WorkgroupSize= 1024 thr
Checking computed result for correctness: Result = PASS
```

How AMD GPUs are seen by CUDA

As mentioned earlier, `hipDeviceProp_t` doesn't contain some parameters CUDA expects, which does necessitate changes to the code. Still most parameters work. `deviceQuery`:

```
Detected 1 CUDA Capable device(s)

Device 0: ""
  CUDA Driver Version / Runtime Version      50120.3 / 50120.3
  CUDA Capability Major/Minor version number: 9.0
  Total amount of global memory:             4096 MBytes (4294967296 bytes)
MapSMtoCores for SM 9.0 is undefined. Default to use 128 Cores/SM
MapSMtoCores for SM 9.0 is undefined. Default to use 128 Cores/SM
(007) Multiprocessors, (128) CUDA Cores/MP: 896 CUDADetected 1 CUDA Capable device(s)
```

```

Device 0: ""
  CUDA Driver Version / Runtime Version      50120.3 / 50120.3
  CUDA Capability Major/Minor version number: 9.0
  Total amount of global memory:             4096 MBytes (4294967296 bytes)
MapSMtoCores for SM 9.0 is undefined. Default to use 128 Cores/SM
MapSMtoCores for SM 9.0 is undefined. Default to use 128 Cores/SM
  (007) Multiprocessors, (128) CUDA Cores/MP: 896 CUDA Cores
  GPU Max Clock rate:                       1900 MHz (1.90 GHz)
  Memory Clock rate:                        1933 Mhz
  Memory Bus Width:                         128-bit
  L2 Cache Size:                           1048576 bytes
  Maximum Texture Dimension Size (x,y,z)    1D=(16384), 2D=(16384, 16384), 3D=(16384, 16384, 16384)
  Total amount of constant memory:          4294967296 bytes
  Total amount of shared memory per block:   65536 bytes
  Total number of registers available per block: 65536
  Warp size:                               64
  Maximum number of threads per multiprocessor: 2560
  Maximum number of threads per block:      1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 1024)
  Max dimension size of a grid size (x,y,z): (2147483647, 2147483647, 2147483647)
  Maximum memory pitch:                     4294967296 bytes
  Texture alignment:                        256 bytes
  Run time limit on kernels:                 No
  Integrated GPU sharing Host Memory:        No
  Support host page-locked memory mapping:   Yes
  Device has ECC support:                    Disabled
  Device supports Managed Memory:            No
  Supports Cooperative Kernel Launch:        Yes
  Supports MultiDevice Co-op Kernel Launch:  Yes
  Device PCI Domain ID / Bus ID / location ID: 0 / 4 / 0
  Compute Mode:
    < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously)

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 50120.3, CUDA Runtime Version = 50120.3
Result = PASS Cores
  GPU Max Clock rate:                       1900 MHz (1.90 GHz)
  Memory Clock rate:                        1933 Mhz
  Memory Bus Width:                         128-bit
  L2 Cache Size:                           1048576 bytes
  Maximum Texture Dimension Size (x,y,z)    1D=(16384), 2D=(16384, 16384), 3D=(16384, 16384, 16384)
  Total amount of constant memory:          4294967296 bytes
  Total amount of shared memory per block:   65536 bytes
  Total number of registers available per block: 65536
  Warp size:                               64
  Maximum number of threads per multiprocessor: 2560
  Maximum number of threads per block:      1024

```

```

Max dimension size of a thread block (x,y,z): (1024, 1024, 1024)
Max dimension size of a grid size      (x,y,z): (2147483647, 2147483647, 2147483647)
Maximum memory pitch:                  4294967296 bytes
Texture alignment:                     256 bytes
Run time limit on kernels:             No
Integrated GPU sharing Host Memory:    No
Support host page-locked memory mapping: Yes
Device has ECC support:                Disabled
Device supports Managed Memory:        No
Supports Cooperative Kernel Launch:    Yes
Supports MultiDevice Co-op Kernel Launch: Yes
Device PCI Domain ID / Bus ID / location ID: 0 / 4 / 0
Compute Mode:
    < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously)

```

```

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 50120.3, CUDA Runtime Version = 501
Result = PASS

```

We can see because I am running an APU, the total amount of global memory is 4GB, the same as I assigned for the iGPU in the BIOS, and L2 Cache size is 1GB, which would be impossible on a dedicated GPU, but as here all memory is just RAM the number can be any chosen by the manufacturer. Most interesting result

Other notes about ROCm

HIPify is having a crazy high development pace, as seen by number of commits

- /tmp/helper_cuda.h-978032.hip:63:3: warning: 'cuGetErrorName' is experimental in 'HIP'; to hipify it, use the '--experimental' option.

Hardware compatibility is far behind CUDA

Nvidia has been really decent with supporting older architectures. As of writing, the latest CUDA 12 has support from Maxwell to Hopper, the oldest supported GPU is GTX 750 family which is 9 years old. And all consumer GPUs in between are supported. Another benefit is that similarly as with CPUs, you can compile for specific compute capability, and all GPUs with same or higher compute capability will work with the same binary. This really helps with build times as well as resulting application size.

In comparison, AMD has a lot of work to do. The latest release of ROCm 5.4 officially supports... only professional grade GPUs, and only from 3 architectures, Vega, RDNA 2 and CDNA. The oldest officially supported GPU, MI50, is a bit more than 4 years old. Of course it does support consumer GPUs, but the

experience is surprisingly spotty. Support for new architectures and OS releases is often delayed. For example good support for RDNA 1 came only around release of RDNA 2.

Moreover when a specific architecture over all should be working, this doesn't mean all GPUs do. Often laptop chips didn't work, same with some APUs. But most egregious was the case of Navi 22, featured in for example 6700 XT. Both higher and lower end models got support since release, but this one didn't. So you either had to build the whole stack from source with this support patched in, or you had to force the Navi 21 kernel.⁷ Compiler supported it from the start, all it needed was adding of IDs in a few places in the code, and yet AMD didn't bother.

Which brings us to yet another pitfall, there is no intermediate representation, each kernel only supports one GFX ID. And that doesn't mean support for one architecture, but one physical GPU die, so just one GPU family can need 3-4 different kernels (like RDNA 2 had Navi 21, 22 and 23).

This means that if you want to build for all supported architectures the build times are an order of magnitude longer than if you just supported one GPU, same for file sizes.

Conclusions

AMD touts its software stack as great for servers, HPC and AI. What it certainly isn't optimised yet for is for smaller scale or personal use. The GPU and OS compatibility is not great, the HIPify has its problems, and packaging from source is really hard or in some cases impossible.

Also no matter how good the CUDA to HIP conversion process is, it is still not a replacement for running CUDA. Almost all the documentation out there is for CUDA, not HIP. Even if it is really similar, it is much easier to write CUDA. So you can either keep writing in CUDA, then convert it to HIP, and then fix some problems, and repeat that with every single update of your CUDA code. Or one can continue working on the HIP code, which works on both GPU vendors, but is less supported and has very little documentation.

But once you have the HIP code, installed stack and supported GPU, the performance is great, AMD offers form factors not available anywhere else, like mobile, desktop or even server APUs. Also the whole stack is open source which is great for security, maintainability and future hardware support, and in that regard, AMD has no competition.

Ideas for improvements:

- HIPify only changes function names, HIP is reimplementing CUDA functions anyways. It shouldn't need installation of CUDA SDK, it necessitates

⁷<https://github.com/RadeonOpenCompute/ROCm/issues/1668>

proprietary software, limits OS compatibility and makes the whole process more complicated for no good reason. At least only the headers should be necessary, not relying on LLVM and its CUDA detection

- ROCm needs to be steadily improved to help with packaging on all distros, not just by Debian and Arch maintainers, but AMD devs need to help too (more directly)
- Official hardware compatibility is a lot worse than with CUDA, for both brand new and aging hardware. Meanwhile versions by community already do fix some incompatibilities, bugs and even keep support for dropped architectures. But then we come back to the problem of how hard the stack is to package
- With how few architectures a specific release supports both distributions and software would have to support multiple ones. While the final nail in the coffin is necessity for separate kernels for every possible GPU die.

Sources: [^1]: <https://developer.nvidia.com/cuda-llvm-compiler>