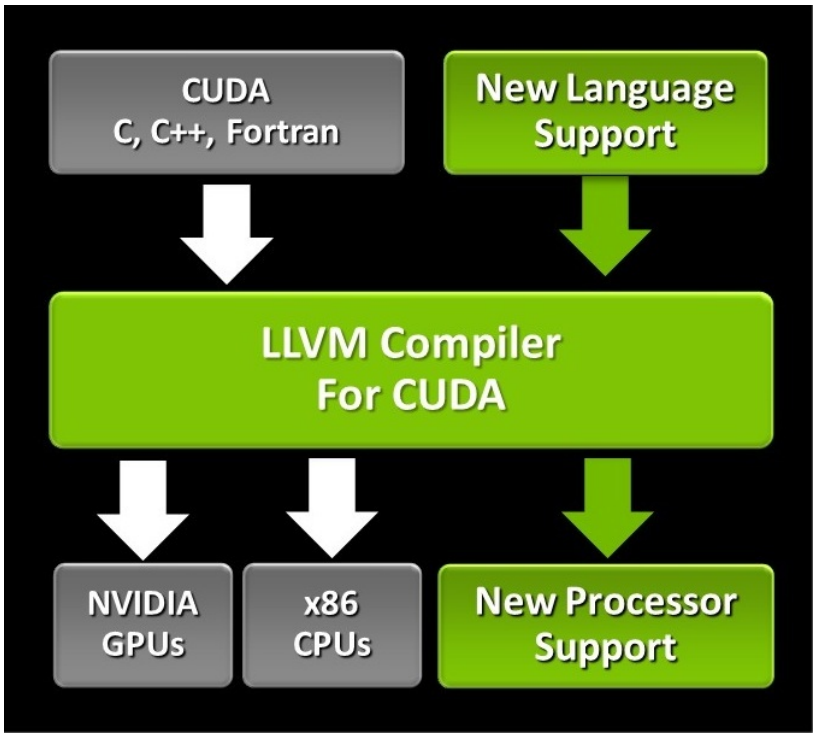
Viability and performance characteristics of using HIPify for CUDA projects

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]



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."[2]

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.[3]

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".[4]

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 one I used. 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.[5]

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.

Here is the command I used on Ubuntu:

```
~/Downloads/HIPIFY-rocm-5.4.2/dist/hipify-clang vectorAdd.cu -I ~/Downloads/cuda-samples-11.6/Common/ --experimental
```

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 occurences 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, 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

Time to look at conversion process and its shortcomings. I used latest version of HIPify compiled from source. I have decided to use Nvidia's CUDA Samples[6] 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'; did you mean
       'maxTexture1DLinear'?
       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'; did you mean
       'maxTexture1DLinear'?
         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:180:74: 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'; did you mean
       'maxTexture1DLinear'?
         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'; did you mean
       'maxTexture1DLinear'?
         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'; did you mean
       'maxTexture1DLinear'?
```

```
        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'; did you mean
        'maxSharedMemoryPerMultiProcessor'?
             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");
            ~~~~~~~~~~ ^
```

- It had problems with finding matching function for very simple types:

```
/tmp/saxpy.cu-e011bd.hip:58:16: error: no matching function for call to 'max'
   maxError = max(maxError, abs(y[i]-4.0f));
              ^~~
```

even in such simple piece of code:

```
float maxError = 0.0f;
for (int i = 0; i < N; i++) {
  maxError = max(maxError, abs(y[i]-4.0f));
}
```

- Converting helper_cuda.h was the hardest as HIPify can't handle `cuBLAS`, `cuRAND`, `cuFFT`, `cuPARSE`, even though those APIs should be supported[7]
- It ignores preprocesor statements, so for example when a file contains `#ifdefs` it will throw errors about e.g. redefinition

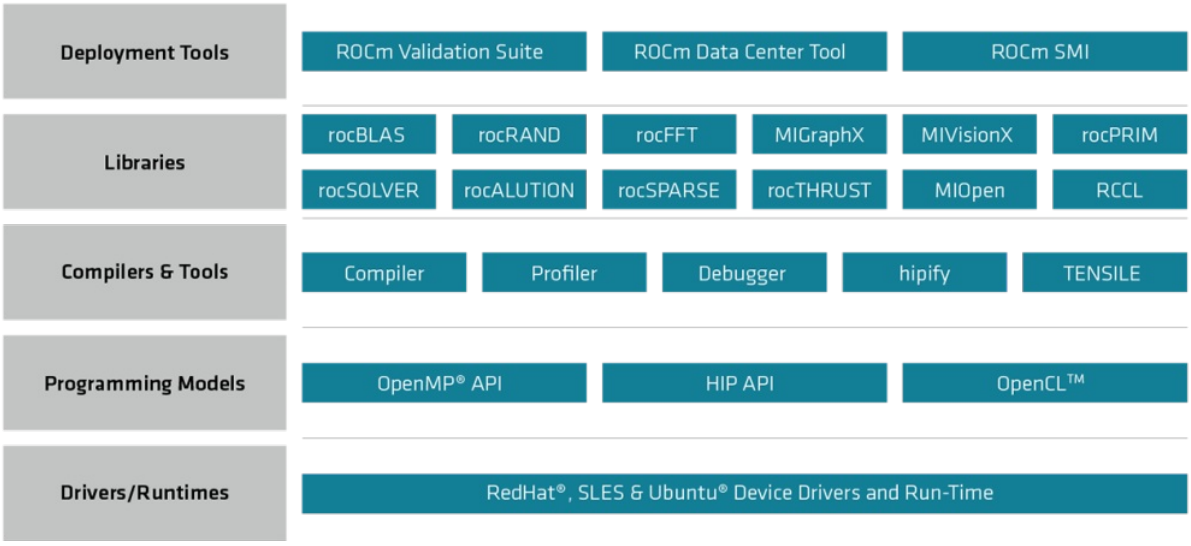
```
#if defined(WIN32) || defined(_WIN32) || defined(WIN64) || defined(_WIN64)
   // Windows path delimiter
   size_t delimiter_pos = executable_name.find_last_of('\\');

#else
   // Linux & OSX path delimiter
   size_t delimiter_pos = executable_name.find_last_of('/');
#endif
```

In the code above it will complain `delimiter_pos` was redefined, because it doesn't understand only `if` OR `else` can be entered.

- 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



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.

Compiling HIP

After the CUDA code has been compiled to HIP, you can use `hipcc` to compile them. The format is the same as when using NVCC or clang

The following command won't work, as even the libraries have to be converted to HIP

```
hipcc vectorAdd.cu.hip -o vectorAddRX580 --rocm-device-lib-path=/usr/lib64/amdgcn/bitcode/ -DROCM_PATH=/usr -I
        ~/Pobrane/cuda-samples-11.6/Common/
```

So you need to HIPify the include folder and then include it

```
hipcc vectorAdd.cu.hip -o vectorAddRX580v2 --rocm-device-lib-path=/usr/lib64/amdgcn/bitcode/ -DROCM_PATH=/usr -I
        ../../CommonConvertedToHIP_5.4.2/
```

And here is how you compile binary from multiple source files

```
hipcc commonKernels.cu.hip helperFunctions.cpp.hip matrixMultiplyPerf.cu.hip --rocm-device-lib-
        path=/usr/lib64/amdgcn/bitcode/ -DROCM_PATH=/usr -I ../CommonConvertedToHIP_5.4.2/
```

Problems when compiling HIP

- It can't find some function calls, that clearly are part of HIP

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

- Compiler can't find an identifier for some reason:

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

Problems when running

Fortunately there weren't many, if you managed to convert and compile successfully, it did work.

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()"`: Here is the related post discussing it.

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.

When I increased saxpy's element number 100 times after a few seconds the GPU hang and I got artifacts all over the screen. I must have filled the GPU memory, but it is bad the GPU driver couldn't recover and the PC required a hard reset.

Vega's capability version is too high for its own good, and AMD's own functions don't know what to return `MapSMtoCores for SM 9.0 is undefined.  Default to use 128 Cores/SM`. This in turn returns the wrong value, as for GCN (and almost any AMD GPU) it should be 64.

Performance

Test systems

I had access to 3 systems to compare. First was lab computer running Ubuntu 22.04, with Nvidia Quadro 4000, which features 36 SMs, 2304 CUDA cores, boost clock 1545 MHz and its theoretical FP32 performance is 7.119 TFLOPS, and memory subsystem with 416.0 GB/s of bandwidth.[8] In performance it is comparable to RTX 2060 Super.

2 of my AMD systems were running Solus 4.3 with ROCm 5.1.3 packaged by me, now available from the main repository.

First one was Radeon RX 580, a very affordable and popular GPU, with 36CUs, 2304 shading units, boost clock 1340 MHz and theoretical FP32 performance of 6.175 TFLOPS, and 256.0 GB/s memory.[9]

Even though the 2 cards are completely different architectures, with Radeon being much cheaper and slower in most workloads, they have exactly the same number of "cores", and the difference in TFLOPS is only because different boost clocks. So it is a great comparison how theoretically equivalent devices can handle different workloads completely differently.

The 3rd system is equipped with Ryzen PRO 4650G, an APU. Its integrated GPU is Vega 7, with 7 CUs, 448 SUs[10], that I manually overclocked as far as it could handle, to 2180MHz, which raises its FP32 performance to 448 * 2,180 * 2 = 1.95 TFLOPS. I also overclocked the RAM to 3866MHz (61.8 GB/s) to try to reduce the memory speed as the bottleneck as much as possible. This iGPU is very interesting for 3 reasons.

For one because it is an integrated GPU, which means it is a very unique form factor that can be really useful for some use cases. In a very small size and power envelope you can get very powerful CPU and in theory GPU. Nvidia's only only competition is their Jetson series, but the software is much more locked down, the CPU performance much worse, and the price much higher.

Second is because it uses Vega architecture, the last before AMD split their GPUs into gaming focused RDNA and compute focused CDNA. While RDNA changed a lot in its architecture to optimise for gaming performance, sacrificing compute, CDNA is direct successor to already very great at compute Vega. So it is a good look at performance of GPUs only available for servers and supercomputers, as well as it should be the best supported consumer GPU family by ROCm.

And lastly APUs over 10 years ago started being pushed by AMD for their combined compute capabilities with HSA (Heterogeneous System Architecture). And while that didn't get much traction, now the compute stack seems to be more ready then ever.

Tests

First test I performed was with `matrixMul` from CUDA samples, with default settings, which I previously checked were the already the best for the Nvidia GPU.

As this test partly uses the shared memory, the GPU performance shouldn't be as bottlenecked by memory performance, as that data should be residing in cache.

Quadro 4000:

```
GPU Device 0: "Turing" with compute capability 7.5

MatrixA(320,320), MatrixB(640,320)
Computing result using CUDA Kernel...
done
Performance= 591.94 GFlop/s, Time= 0.221 msec, Size= 131072000 Ops, WorkgroupSize= 1024 threads/block
Checking computed result for correctness: Result = PASS
```

RX 580

```
Performance= 1029.97 GFlop/s, Time= 0.127 msec
```

Vega 7

```
Performance= 216.03 GFlop/s, Time= 0.607 msec
```

Here the RX 580 really surprised, it was 74% percent faster than much more expensive professional GPU.

APU was also really impressive 36.4% performance of the RTX 4000, with only 27.4% of the theoretical TFLOPs, and about 10x smaller power consumption.

Next I tried running saxpy, an simple exercise program from one of our classes, with no manual optimisations.

Quadro 4000

```
Effective Bandwidth (GB/s): 380.884864
```

RX 580

```
Effective Bandwidth (GB/s): 174.323280
```

Vega 7

```
Effective Bandwidth (GB/s): 51.354900
```

Here the Polaris GPU's performance was much less impressive, achieving only 45.8% performance of the Nvidia's GPU, but it is still not bad compared to the price of the GPU. Maybe that is because Radeon's memory has 61.5% throughput of the Quadro, but I don't think that is the case.

APU's bandwidth was only 13.5% performance of the Quadro, while compared to the 580 its performance scaled almost linearly with TFLOPS - 29.5% of Polaris' performance with 31.5% of TFLOPS. Or if you look at the memory it has 14.9% bandwidth of the Quadro, so the scaling would also make sense here.

So in this workload either AMD GPUs perform 2 times worse than Nvidia for the same TFLOPS, or the performance scales closely with the memory bandwidth.

The last program I benchmark was vectorAdd from CUDA samples, where I manually added execution time counting.

Quadro 4000

```
[Vector addition of 50000 elements]
Copy input data from the host memory to the CUDA device
CUDA kernel launch with 196 blocks of 256 threads
Time= 0.006 msec
Copy output data from the CUDA device to the host memory
Test PASSED
Done
```

RX 580

```
Time= 0.354 msec
```

Vega 7

```
Time= 0.263 msec
```

Here the Radeon GPUs did really bad, and I am not sure why, RX 580 was 59 times slower than the Nvidia GPU. My guess is that this is all because the kernel execution time is so low, most of it are not the calculations but the time to launch it itself. And maybe that time is much higher with HIP?

To test that theory I increased the element size 100 times and rerun the tests: Quadro 4000

```
[Vector addition of 5000000 elements]
Copy input data from the host memory to the CUDA device
CUDA kernel launch with 19532 blocks of 256 threads
Time= 0.163 msec
Copy output data from the CUDA device to the host memory
Test PASSED
Done
```

RX 580

```
Time= 2.559 msec
```

Vega 7

```
Time= 1.402 msec
```

Polaris was still 15.7 times slower, so I even though the kernel launch time could be a factor, it clearly isn't the only factory why AMD GPU is slower. Maybe Nvidia GPU has additional optimisation, but that is unlikely as both compilers are based on LLVM. Or maybe this is an AMD bug?

I don't think it is a big problem though, as we saw previously longer workloads do perform in line with expectations.

How AMD GPUs are seen by CUDA

As AMD architectures are so different from Nvidia's I reall wanted to see how they are seen by CUDA programs. So I converted `deviceQuery` from CUDA Samples to HIP and ran it. As mentioned earlier, `hipDeviceProp_t` doesn't contain some parameters CUDA expects, which does necessitate changes to the code. Still, most parameters do work. Quadro 4000

```
Detected 1 CUDA Capable device(s)

Device 0: "Quadro RTX 4000"
  CUDA Driver Version / Runtime Version          11.7 / 11.7
  CUDA Capability Major/Minor version number:    7.5
  Total amount of global memory:                 7982 MBytes (8370192384 bytes)
  (036) Multiprocessors, (064) CUDA Cores/MP:    2304 CUDA Cores
```

```
  GPU Max Clock rate:                            1545 MHz (1.54 GHz)
  Memory Clock rate:                             6501 Mhz
  Memory Bus Width:                              256-bit
  L2 Cache Size:                                 4194304 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total shared memory per multiprocessor:        65536 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1024
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 3 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Managed Memory:                Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 11.7, CUDA Runtime Version = 11.7, NumDevs = 1
Result = PASS
```

RX 580

```
Detected 1 CUDA Capable device(s)

Device 0: "AMD Radeon RX 580 Series"
  CUDA Driver Version / Runtime Version          50120.3 / 50120.3
  CUDA Capability Major/Minor version number:    8.0
  Total amount of global memory:                 4096 MBytes (4294967296 bytes)
  (036) Multiprocessors, (064) CUDA Cores/MP:    2304 CUDA Cores
  GPU Max Clock rate:                            1340 MHz (1.34 GHz)
  Memory Clock rate:                             1750 Mhz
  Memory Bus Width:                              256-bit
  Maximum Texture Dimension Size (x,y,z)         1D=(16384), 2D=(16384, 16384), 3D=(16384, 16384, 8192)
  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:            No
  Supports MultiDevice Co-op Kernel Launch:      No
  Device PCI Domain ID / Bus ID / location ID:   0 / 38 / 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, NumDevs = 1
Result = PASS
```

Vega 7

```
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 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, 8192)
  Total amount of constant memory:               4294967296 bytes
```

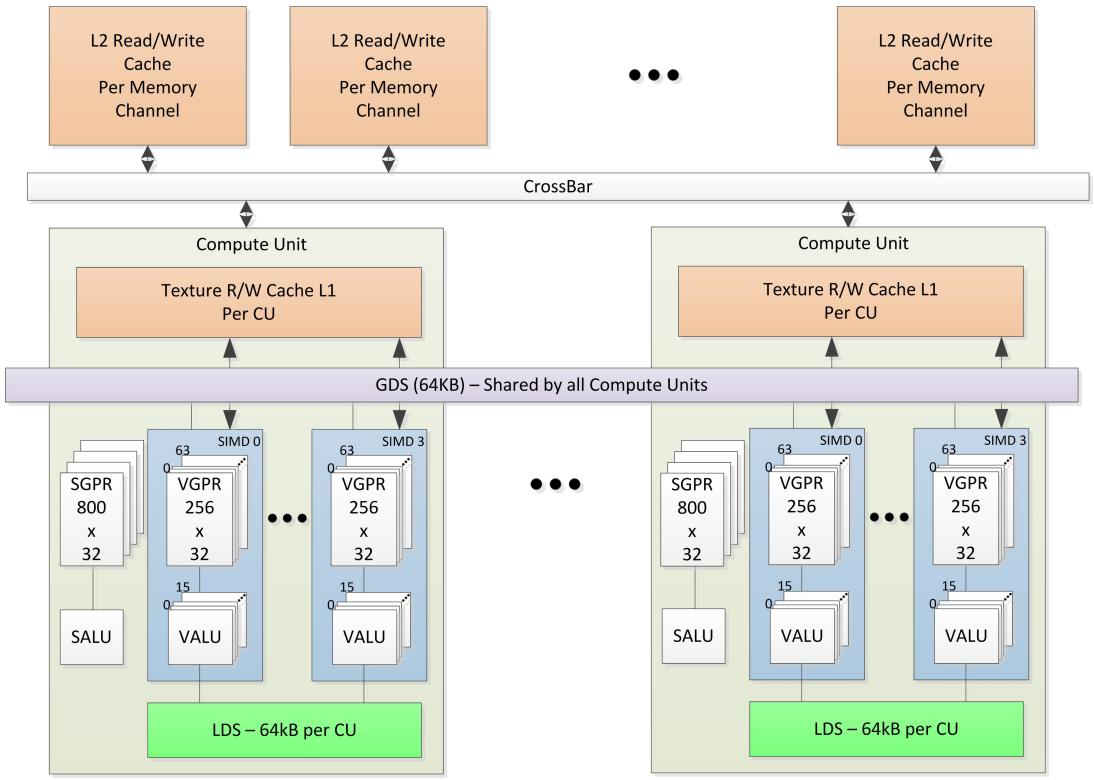
```
  Total amount of shared memory per block:       65536 bytes
  Total number of registers available per block: 65536
  Warp size:                                     64
  Maximum number of threads per multiprocessor:  2560
  Maximum number of threads per block:           1024
  Max dimension size 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, NumDevs = 1
Result = PASS
```

Interesting that even though both AMD architectures are older than Turing, they report higher CUDA capability than it does.

On AMD GPUs the warp size is 64 instead of 32 common for all Nvidia GPUs. Some programs might not expect that being anything other than 32. All GCN GPUs execute instructions in wave64, while RDNA moved to wave32.[11]

I was also wondering why Polaris reports to have more shared memory per block, even though it has half the L2 and 4 times less L1. If any of that memory is by default put in cache, or all of it in RAM and you have no control of what goes into cache. I have managed to link this value to HSA_AMD_MEMORY_POOL_INFO_SIZE, and then to the function it gets this value from. I finally managed to trace those values, turns out all GCN have 64KB of local and shared memory, and that is more than it has cache.[12]



Shared Memory Hierarchy

Also interesting that it didn't return L2 cache size for Polaris but did for Vega.

On the 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.

Because of the Open Source nature I was able to dig through layers of ROCm stack and find how those values are calculated, like this one.

Building and packaging ROCm from source

The fact ROCm is fully open source is a big advantage of AMD's solution. But the fact the source is viewable is one thing, but how useful

that fact is another thing.

While on Windows you get programs straight from the creator, on Linux almost every piece you use has been packaged from source by distribution maintainers. This approach greatly improves flexibility and compatibility, also allows for customizing or fixing specific bugs by distributions themselves. But this approach needs some work by the developer to make a flexible build process for an app.

A notable exception from all the other Linux packages are Nvidia drivers which are only released as binaries from Nvidia. This many times caused incompatibilities with new kernels, and for full GPU support distributions have to support multiple driver versions.

So I decided to package ROCm for Solus, an independent distribution, and I wanted to get it into repositories for everyone to use. It took me 3 months to get most of the stack working, and I encountered a ton of problems along the way. It can be clearly seen that the whole code was mainly expected to be build into binaries AMD shares, and not to be packaged by distributions.

Here are most notable of the issues I found:

- many paths were hardcoded in the source files, so often files are expected to be found in `/opt/rocm` and not in standard unix folders distributions want to install them, like `/usr/share`, or `/etc/bin`
- there are variables for specifying where you want to put files, but there are way too many, there is no pattern to them, and they don't always work. Like `ROCM_PATH`, `ROCM_DIR`, `HIP_PATH`, `HIP_CLANG_PATH`, `DEVICE_LIB_PATH`, `HSA_PATH`, `ROCM_ROOT_DIR`, `amd_comgr_DIR`, and more
- there are a lot of packages, all depending on each other, but there is very little consistency in the build system, which can be seen by the number of variables
- AMD has its own fork of LLVM. Most of the time all the things introduced in it come to mainline LLVM, but not always. So the main system LLVM needs to include some patches from AMD fork, which can be hard to convince distribution maintainers to include. And even then when using current LLVM the ROCm version has to be multiple months old, as the LLVM release cycle is quite long
- to use newest version of ROCm you have to package AMD's fork of LLVM, which is additional work. And even then when my friend tried that, some packages would get confused which LLVM to use and refuse to build
- all the above require a ton of patches, for every single package, which need to be rebased and updated with each ROCm or LLVM upgrade, which makes the whole process even more difficult

So far the only mostly complete Open Source ROCm can be found in Debian repos, while for example AMD developer has packaged a part of the stack for Fedora, so at least the OpenCL support is working (but no HIP). And Arch has build scripts for complete stack, but those have to be compiled yourself, they are not in the repositories.

Those community are projects are the best chance of improving the build process, as when they have to patch things, they try to get that change included upstream as well. They also serve as a great reference for other distributions.

The package sources can be find on my Github or Solus' dev tracker.

Other notes about ROCm

Whole ROCm has a very high development pace, as can be seen by number of commits to repositories as well as number of releases

For example library compatibilities is getting better - `/tmp/helper_cuda.h-978032.hip:63:3: warning: 'cuGetErrorName' is experimental in 'HIP'; to hipify it, use the '--experimental' option.`

Or not too long ago launching kernels became the same as in CUDA, while before the syntax was completely different.

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.[13] 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.[14] So even the RX 480 I was testing on is not supported and I had to patch in the support for it.

Of course it does work with 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 belonging to that family 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 it to use Navi 21 kernels.[15] 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 3rd party documentation.

AMD is also in a weird place with GPUs. In 2011 they have introduced GCN that was "Geared for compute"[16], for years their GPUs had great compute capabilities, but almost no software could use them, while the gaming performance was lacking. So finally they decided to split the development into gaming-focused RDNA, and continue GCN legacy with CDNA. So now all consumer GPUs are not as great for compute, and they are not well supported either as all datacenters use GPUs with different architecture. And you can't buy any CDNA GPUs from retail. So if you want a true compute GPU you need to go back to Vega.

But once you have the HIP code, installed stack and supported GPU, the performance can be great, minus some weird edge cases. 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 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. Also there is only so long community can keep older architectures working, as at some point things break, and no one will ever fix them
- AMD needs to release CDNA for consumers, or support RDNA better for compute
- 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:

The repository with the test samples can be found here.

---

1. https://developer.nvidia.com/cuda-llvm-compiler__
2. https://github.com/ROCm-Developer-Tools/HIPIFY__
3. https://www.llvm.org/docs/CompileCudaWithLLVM.html#dialect-differences-between-clang-and-nvcc__
4. https://github.com/ROCm-Developer-Tools/HIPIFY__
5. https://github.com/llvm/llvm-project/blob/7e629e4e888262abd8f076512b252208acd72d62/clang/lib/Driver/ToolChains/Cuda.cpp#L123__
6. https://github.com/NVIDIA/cuda-samples/archive/refs/tags/v11.6.tar.gz__
7. https://github.com/ROCm-Developer-Tools/HIPIFY#-supported-cuda-apis__
8. https://www.techpowerup.com/gpu-specs/quadro-rtx-4000.c3336__
9. https://www.techpowerup.com/gpu-specs/radeon-rx-580.c2938__
10. https://www.notebookcheck.net/AMD-Radeon-RX-Vega-7-Graphics-Card-Benchmarks-and-Specs.450004.0.html__
11. https://gpuopen.com/performance/__
12. https://rocmdocs.amd.com/en/latest/GCN_ISA_Manuals/testdocbook.html#data-sharing__
13. https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units__

14. https://en.wikipedia.org/wiki/List_of_AMD_graphics_processing_units
15. https://github.com/RadeonOpenCompute/ROCm/issues/1668
16. https://www.anandtech.com/show/4455/amds-graphics-core-next-preview-amd-architects-for-compute/3