GPU Tutorial 2023 Questions and Notes

Please use this document to write any questions or comments you have for the presenters

Would you mind sharing the event code? Registered late and didn't receive the email. Sorry.
Code: `CERN_XLAB_YR23`

Day 1 Questions

Do device and host memory associated with the RAM or is it entirely different and linked in particular with GPU and CPU itself?

Host memory == RAM. That is the CPUs main memory. Device memory we refer to as 'global memory' on the GPU. That is the globally accessible VRAM on the device. It is physically separate from the CPU's RAM. For example, a GPU with "96GB of VRAM" is a GPU with 96GB of global memory that can be accessed by all the SMs on that GPU. You have to move data between RAM and VRAM over the PCIE (or SXM2) bus. The CPU cannot access VRAM, and the GPU cannot access RAM directly.
(Thanks)

Can we run the entire program on GPU only? E.g. can we convert main() into a kernel?

No. As Matt mentioned earlier, the GPU is a supplemental device. You offload computationally intensive portions of code to the GPU. The GPU cannot do things like file I/O that the CPU can do. It's a simpler, but more parallel device. Typically, you offload a small portion of your actual codebase. The performance implication is large because those portions are typically the ones that result in a lot of runtime. Parallelizing accelerates those regions. (Thanks)

In the blockid exercise: Do we expect the order of block number printed to always be 0, 1?
I am getting them in the order 1, 0.
Order is not guaranteed. In fact the order of execution of threads on the device is not deterministic.

Is there a maximum number acceptable for N?
there are a max of 1024 threads per block

Do these blocks run in different cores of GPU parallelly?
usually. you can oversubscribe the GPU, in which case not all will run concurrently. it's actually considerably more complicated, as it depends on what other operations are in the kernel, eg do you need to load memory, etc. all this is handled by the scheduler, which we'll get into in a bit. but in general, you can think that each thread executes concurrently.

Why are threads ordered?
not sure what you mean by "ordered". they are *indexed*.

How many cores a SM can have?
Depends on the GPU architecture

Who schedules them? Is it OS?
CUDA Runtime

Is the entire code inside the "__global__ void hello()" running on the GPU? or parts of it run on the CPU and parts on the GPU. For example is the GPU handling the printf statement?
The entire code inside the kernel declaration is run on each thread that you spawn on the GPU. The total number of threads can be calculated by multiplying the number of blocks by the number of threads per block in your

execution config. So, if you had a launch config kerne<<<N,M>>>, your total number of threads would be N*M.

Why do we need block ID when there's already a pointer to the GPU memory? (copied from chat)
the thread/block ID are used to index a particular element in the array. you usually only have the address of the start of the array. this is done as you usually want individual threads to work on different elements of the memory.
You're basically calculating the offset for each thread starting from the GPU memory pointer.

Does the indentation of the blocks matter in CUDA?
no - it's like C++ or C. whitespace is not important.

Can index of thread or block the index of element we would use in an array?
It's a function of both IDs, as Matt showed. The canonical way of calculating a global array index based on a given thread's thread ID and block ID is below:
Int index = threadIdx.x + blockIdx.x*blockDim.x
Where threadIdx.x is the thread's ID, blockIdx.x is the ID of the block containing that thread, and blockDim.x is the block size (number of threads per block), which would be the "M" in the above launch configuration.

What is more efficient: hello<<<2, 512>>> or hello<<<512, 2>>>?
Neither. Ideally, we want to have many blocks AND many threads per block. Both of these configs will result in inefficiencies that I described verbally.
In general though, 512,2 is worse. As Rob said, threads are executed in warps of 32 threads. let's say that your kernel has 10 instructions, then 512,2 would launch 512 warps, so 5120 instructions. 2,512 would launch 2*16 warps, so 320 instructions.

Would hello<<<2, 512>>> also provide more latitude to schedule other kernels on the unused SMs are the same time?
Yes, it would assume you have another kernel that operates on different data that can be launched asynchronously with no data race conditions. However, this is uncommon in my experience.

How about hello<<<32, 32>>> then?
Definitely better, but still leaves many SMs unused. On the A100 GPU, we have 108 SMs, each with 64 fp32 cores and 32 fp64 cores. So, we would ideally want a launch config of at least <<<108,64>>> for a single precision problem to actually use all the hardware on the GPU. The best case would be many more than 108 blocks because the more work you feed to the GPU, the better load balancing you'll get, and obviously more speedup :)

technically: can you have a single kernel launch with a different number of threads per block? (not that that necessarily would make any sense to do) (copied from chat)
You mean different numbers of threads across blocks in the same kernel launch? No you cannot. Every block will have the same number of threads. You can launch the kernel a second time with a different configuration if you wish. But in one given kernel launch, all blocks will always have the same number of threads specified in your launch configuration.

Is it possible to run several different kernels concurrently on the GPU - so as to not keep parts of the hardware unutilised? (maybe this is covered later in the course?)
yes. there are several ways of doing this, such as using cuda streams and launching kernels from different threads, or launching kernels asynchronously from the same thread without forcing a synchronisation, or using the MPS service to run kernels from different processes concurrently.

Isn't a direction already determined by the order of arguments (the 1st is destination, the 2nd is source) of cudaMemcpy? Or the compiler cannot know which pointer is which?
yes, but it helps with clarity of code. and helps you find errors more easily.

It also eliminates runtime overhead by not making the runtime figure out what types of pointers you're passing for source and destination.

Why aren't host and device pointers differentiated in the type system?
They are in the sense of how they are allocated (malloc vs cudamalloc). As for the type, the data type is the same whether it resides in host memory or device memory. A 32-bit float is the same whether it resides in CPU RAM or GPU Global Memory. Best practice here is to append some kind of signifier to the variable name. So, if i allocate array 'arr' on the CPU, I'd allocate something called 'd_arr' to signify that this is the 'device' space for moving 'arr' over to the device as needed.

What happens if one forgets to free the device memory?
Typically nothing. It will be automatically freed when execution completes. It's similar to forgetting to free CPU memory. It is good practice to do it to avoid any possible memory leaks.

These API look very much C, and not so much C++, especially not modern C++. In particular one should never use malloc, nor see a raw pointer, but rather use make_unique and references. Does cuda code have similar "more modern" APIs ? I feel like I go 25 years back in time here… Not mentioning the error handling. Any exception mechanism ?
It is very "C" like. There are no exception mechanisms. There are higher level tools that are available, such as linear algebra solvers, FFTs, RNGs, etc, or directive based programming like OpenMP or OpenACC. There's also the Thrust library, and more recently nvc++ supports std::par executing on GPUs for standard C++ syntax.

As seen from example we are jumping around pointers from host to device or vice a versa through cudaMemcpy. But if there is an array with a large number of elements then how des it work? Where all the data of an array sits? Will it do a complete copy of an array?
It will copy however much data you specify in the size argument. You can copy parts of an array or you can copy the entire array.

How do all these cuda functions work when we have more devices (several GPU cards) on one host?
A Kernel is launched on a single device only. So, if you have multiple devices, you'll need to decompose your problem such that you launch a portion of the work onto each available device. So, for example, if you have 4 GPUs on your machine and you want to make use of them all, you'd want to decompose your

problem by a factor of 4 (into 4 'slices') and then send a slice to each GPU and launch a separate kernel on each device to compute its portion of the work. In most HPC applications, this decomposition is done using MPI because you can launch an MPI rank for each device on a given node. MPI makes it possible to scale across devices AND nodes at the same time. So, if you have 10 nodes, each with 4 GPUs, youd want 4 MPI ranks per node (40 ranks total) and have each MPI rank bind to a different GPU for its portion of the total work.

I'm using Quadro rtx 4000 on my local server. I find it hard to find the data sheet for how many SM units it has. However there's a terminology called RT core. It looks like the terminology between architecture could be different?
CUDA core/FP32 = 8 * Tensor core = 64 * SM/RT core?
"RT" refers to "Ray Tracing", which is a specialized bit of hardware that accelerates ray tracing calculations. It appeared relatively recently (around 2019) in the design of NVIDIA GPUs. Tensor cores are another type of specialized hardware that accelerates low/mixed precision matrix manipulations (useful for ML). The RTX 4000 has 36 SMs and one RT core per SM. You can query the GPU directly with the following (using nvcc):

```
#include <iostream>
int main() {
    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, 0);
```

```
        std::cout << deviceProp.multiProcessorCount <<
std::endl;
        return 0;
    }11
```
On the NVIDIA DLI instances (Tesla T4 GPUs), this will print out "40".

Day 2 Questions

Is that 48 KB (shared memory) per SM or total?
Which memory section were you asking about?
48 kb shared memory per threadblock in this case, and that changes with GPU generation: check out this link if you would like to know about other generations
https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities Compute capabilities is a great table to look at

80 GB A100 means that memory is Global Memory?
Dram (global mem) thats right

would you then not need a synchronization after the for loop?
No - the cudaMemcpy (line 63 of vector_add.cu) is a blocking operation, which requires the CUDA kernel in the same stream to complete
The stream synchronization Robbie mentioned relates to the gpu scheduling of the kernels, from the CPUs perspective it will launch all those kernels one after another, and then just return control to CPU program. The GPU kernels will wait for each other to finish, but without synch with the host

For small values of DSIZE I see both dram__bytes_read & write go to 0 for the fifth kernel launch. Is it obvious what that means?

that means that the whole vector is fitting in L2 cache, so global memory doesn't need to be accessed.

^ so then cudaMemcpy device -> host is served straight from the L2 for C?
Typically nsight compute is profiling the kernel itself. So cudaMemcpy is going to dram (global), and its likely that the cuda runtime is smart enough that if the data is small enough, it will move it to the L2 for you. All global mem data accesses on a nvidia gpu goes through L2, so when the kernel goes to look for the global mem, it checks first in L2, finds it, and services the request from there, for this kernel. As far as the programmer is concerned, cudamemcpy goes to global memory, and whether that ends up in dram or L2 is a compiler/runtime decision based on size of allocation.

FYI cudaDevAttrMaxPersistingL2CacheSize from cudaDeviceGetAttribute() and persistingL2CacheMaxSize from cudaGetDeviceProperties() both return 0 on the T4s.

Behaviour seems to change around const int DSIZE = 32*8 * 1024 * 1
which would suggest 1 MB of cache, right?
(32*8*4 byte floats gives 1 KB)

Will every cudaMalloc be aligned to a cache line boundary?
Yes

The code works, but the profiler is returning a LaunchFailed (first profiler cell)

```
==ERROR== LaunchFailed
==PROF== Trying to shutdown target application
==ERROR== The application returned an error code (9).
==ERROR== An error occurred while trying to profile.
```

Try adding `--replay-mode application` to the ncu command line, that worked for me in some case

For me the following worked:

```
!ncu --replay-mode application ./matrix_sums
```

Should row access be more efficient given the indices are next to each other,
while column indices are separated by the dimension of the matrix?
no, because each thread will be requesting a new cache line. with column access, each thread will request the same cache line (up to a limit of the size of the matrix)

In principle one can download all notebooks and source files with in START_HERE.ipynb:

```
!tar --exclude=all.tar.gz -zcf all.tar.gz .
```

And then download all.tar.gz from the sidebar.

OK, who messed up the formatting ;-)
The notebook is also available as a direct download from the Indico

---

Day 3 Questions

Right here: 11111111111111111111111111111
(that's how many people were set up to be able to ask questions in this Google Doc on day 3)

how would you profile a kernel with nsys from python?
=> demonstrated an example

But this events is in GPU, no? 🙂
=> yes, and by iterating over them in Python, it's pulling each value from the GPU, one at a time. Very inefficient, but it's a correctness test of 5 events.

Working on the project until 30 minutes past the hour, and then a 5 minute break (35 minutes past the hour). Then, we'll review solutions.

I have a problem installing md5sum, will check the DLI ⇒ Can you please provide the miniforge for MacOS, similar to link
Is the github.com/conda-forge/miniforge page what you're looking for?
⇒ ups, found it 🙂

Miniforge and mambaforge have coalesced; a month or so ago, they became the same thing. (I still call mine "mambaforge" so that badly written scripts still work, but what I've installed is "miniforge." I just use the "mamba" command within miniforge, because it now works.)

Also see: https://scikit-hep.org/user/installing-conda

I have an issue with JuptyerLab on DLI: 100% disk space usage?
What can I do?
Could be the Conda environment
=> conda environment uses 14 GB, but the DLI system should give you 250 GB. Maybe there's something left over from days 1 and 2.

I observed that I can't do something like 92.34//1 with cuda
=> you can compute 92.34 // 1, but the result is technically floating point (even though it lands on a whole number). That can't be used as an array index.

One thing that confused me: in the beginning you say "create an array with dimensions (10000000, 120) and have each thread fill its own row of the two-dimensional array."
but then on your solution:

`num_threads = 1024`
which means a thread will fill multiple rows..
Got it!

math.floor in numba returns float, but math.floor returns int in pure python though
Do you consider this an incorrect implementation
=> This page in the Numba docs suggests that they prefer the type signature conventions set by libm etc. over Python's behavior. Also relevant is the fact that Numba interprets Python's "int", which is an arbitrary precision "BigInt" (in SQL's terminology), as though it were int64_t. That's not out of place with interpreting math.floor as having the signature of the floor functions in libm. So, it's probably not considered a bug. I wish they would accept np.* functions in numba.cuda, since np.* functions have the signatures you'd expect from libm.

will closing a jupyter window delete all the memory on the gpu that was allocated on it? yes - i meant the tab.

Does nb.cuda have something like xxhash (so you can hash the thread id)
=> Not quite the same thing, but somebody's written SHA256 in numba.cuda device functions. There are xxhash.h implementations for CUDA, and this declares external C functions that can be called in numba.cuda. The interface is a lot like ctypes/cffi in that the external function needs to have a C API ('extern "C"' in C++). The compiler-research.org group is working on calling C++ directly from Numba-compiled functions using cppyy, although I'm sure that's only for CPU functions at this stage. I don't know if it would be an easy extension to do that for CUDA later…

We'll do project 2 until 40 minutes past the hour, at which point, I'll walk through solutions.

Ask any questions! During the project time, I'll type responses to not disrupt people's concentration.

x = xoroshiro128p_uniform_float32(rng_states, thread_idx)*2 - 1, why -1?

Why don’t we get the same values for x and y when using
x = xoroshiro128p_uniform_float32(rng_states, thread_idx)*2 - 1
y = xoroshiro128p_uniform_float32(rng_states, thread_idx)*2 - 1
Since we use the same thread_idx?

OK, I understand that rng_states is mutable. Thanks!

I sample 100000000 points can get out of memory. Is it expected?

This was the Zoom chat at the end of day 3:

09:05:22 From Matt Stack_Nvidia To Everyone:
    https://github.com/jpivarski-talks/2023-11-02-atlas-gpu-python-tutorial
09:05:29 From Matt Stack_Nvidia To Everyone:

https://docs.google.com/document/d/1V0qfPrj710XG4d4eceqE0bD28V7tdtt
PT2tv8EOMOZQ/edit?pli=1#heading=h.k6ub9gkdfd48
10:45:33 From Rongkun Wang To Everyone:
    it supports math.floor
10:48:32 From Vangelis Kourlitis To Everyone:
    this is what I also used on my “solution”
10:49:08 From Rongkun Wang To Everyone:
    it should support operator.floordiv though..
10:50:03 From Rongkun Wang To Everyone:
    python3 😄
11:27:44 From Vangelis Kourlitis To Everyone:
    Thanks for this tutorial Jim! I know have to disconnect unfortunatelly,
meeting with student.
12:09:14 From Ehizojie Ali To Everyone:
    Great Session
12:09:46 From Saurabh Saini To Everyone:
    Thanks for the tutorial. It was really nice and learnt a lot.
12:10:31 From Ehizojie Ali To Everyone:

Great session today. Could there be an extended session on C++ GPU Programming [without CUDA] ?
12:15:35 From Jim Pivarski To Everyone:
nvidia-smi
12:16:30 From Jim Pivarski To Everyone:
cudatoolkit
12:16:40 From Jim Pivarski To Everyone:
numba -s
12:17:25 From Ehizojie Ali To Everyone:
Great troubleshooting tips.
12:18:26 From Jim Pivarski To Everyone:
Thrust?
12:19:33 From Ehizojie Ali To Everyone:
Yes. Thanks