Argonne NATIONAL LABORATORY

Introduction to GPU “Low-level” Programming

How does that even work?

Thomas Applencourt, Nathan Nichols ({apl,nnichols}@anl.gov)

July 28, 2025

Disclaimer

- “If it’s simple, it’s always false. If it’s not, it’s unusable.” Paul Valéry
- “Trust, but verify.” Russian proverb
- “Stay awhile and listen...” Deckard Cain [1]

[1] And sorry in advance for the 2h long lecture

ToC

Introduction

Non-Goals of this lecture

- I will not teach you CUDA[2], HIP[3], Level Zero[4]
- You are all smart, if you need to learn it you can find super nice tutorial online

[2] Compute Unified Device Architecture,
[3] Heterogeneous-Compute Interface
[4] But maybe I will teach you some OpenCL – Open Compute Language...

Goal of this Lecture: How This code is Implemented

OpenMP[5,6]

```
1 #pragma omp target parallel for map(to: B[0:N]) map(from: A[0:N])
2 for (int i=0; i < N; i++)
3   A[i] = B[i];
```

I'm going to ask someone at the end, so listen carefully!

[5] Open Multi-Processing... Wait what?!

[6] I use OpenMP – as example because after me, you have the chance, the privilege!, to get a OpenMP tutorial. But same goes for Kokkos, SYCL, Python

Goal of this Lecture

- Give you some foundation to understand the difference and similitude between multiple low-level programming models (“Any fool can know. The point is to understand.” Ernest Kinoy)
- Make clear the layering approach of current toolchains

Some big questions

- Why the “need” to use CUDA for NVIDIA hardware, HIP for AMD, and L0 for Intel?[7]

(a) Level Zero

(b) CUDA

(c) HIP

[7] But then how can OpenCL be portable?

GPU-programming consist on three things:

- Programming Model for the GPUs Kernel (OpenCL C, CUDA C, ...).
- Execution Model (SPMD / “MPI-like”)
- Programming Model for the “CPU Runtime” (CUDA, L0, OpenCL)[8]

[8] This is the only thing who matter, this rest is trivial

Programming Model for the GPUs Kernel

GPU is a different Hardware

- GPU Doesn't execute the same machine code as your CPU
- You need and additional compiler[9] to lower your high-level language to GPU machine code

[9] or compiler backend

Old School: Double Source Approach

- **Host Code**:
  - Written in **C/C++/Python** for CPU tasks like memory management and kernel launches.
- **Device Code**:
  - Written in GPU-specific languages like CUDA C/C++, HIP C, or OpenCL C[10]
  - Contains the GPU kernels.

[10] PTX, SPIR-V, ...

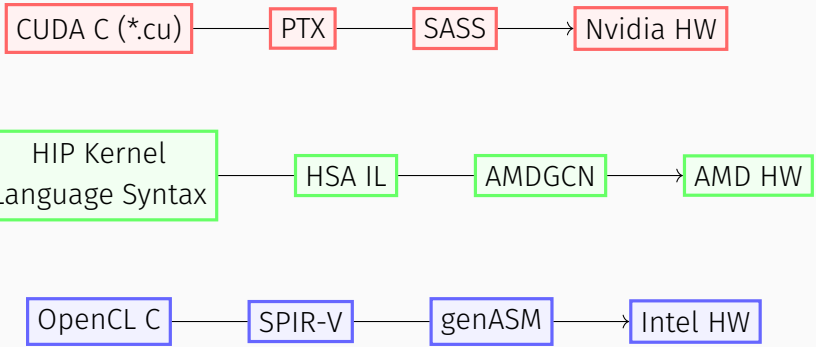
GPU program

```
1  // CUDA/HIP
2  extern __shared__ float array[];
3  __global__ void cuda_or_hip_hello() {
4      printf("Hello World from GPU!\n");
5  }
6  // OpenCL
7  __kernel void hello_world() {
8      printf("Hello World from GPU!\n");
9      local int a;
10 }
```

So much difference!

- Please note the `__global__`, or `__kernel`, those are not standard C keyword.
- What can you put in device code has limitation depending on the Hardware / Compiler
- No dynamic allocation, no throw, no recursion, no virtual Functions, ...

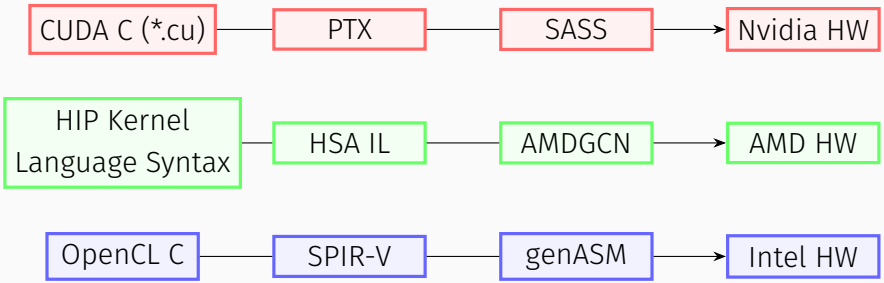
Compiling: Simple first steps



SPIR-V is an Open-Source Standard[11]

[11] The other are not, just saying

Compiling: Simple first steps



SPIR-V is an Open-Source Standard[12]

[12] The others are not, just saying

Double Source Compilation Process

- **Separate Compilation**:
  - Host code -> x86/ARM machine code.
  - Device code -> Intermediate code (PTX, SPIR-V) -> GPU-specific machine code.
- **Potentially Linking**: Combines host and device code into the final executable (see next slide single source).

Hiding Compilation: Emergence of Single Source

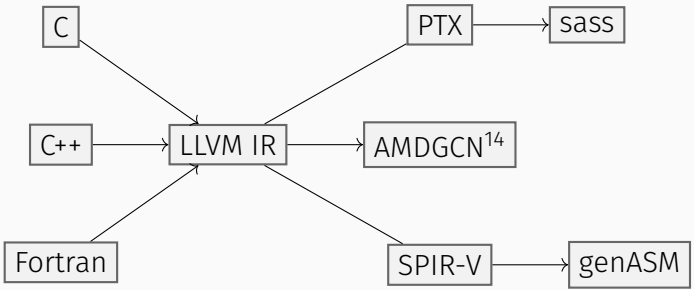
```
1  // OMP
2  printf("Hello World CPU!\n"); // CPU code
3  #pragma omp target            // GPU code
4  printf("Hello World GPU!\n");
5
6  // SYCL
7  printf("Hello World CPU!\n");   // CPU code
8  Q.single_task([](){printf(Hello World!\n");}}.wait(); // GPU code
9
```

Compiler is smart enough to understand which code will be executed on the CPU, and which one on the GPU.

- You can use a subset of C/C++/Python as a GPU kernel[13]

[13] Remember previous limitation: no recursion*, no exceptions, no syscall, no io...

Compiling: A Better(?) method – LLVM example



(you have bridge between SPIR-V and PTX)

[14] or nobody generate AMDGCN and everybody HSA IL, not clear...

Recap

- Two compilation path, always
- Can do it at runtime (JIT) or at compile time (AOT)
- GPU is fast, because it's stupid and have restriction

Execution Model

Execution Model

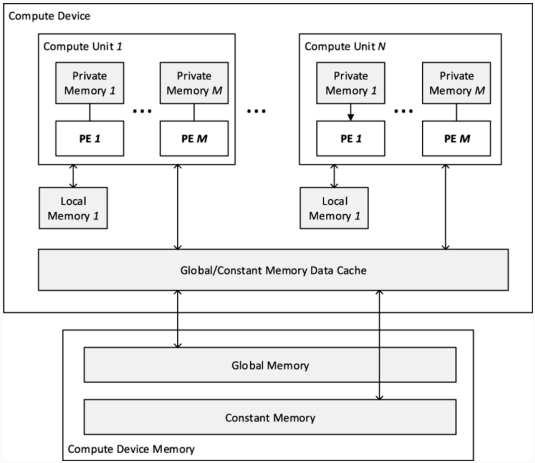
Kernel Language: GPU programming 101

Memory Hierarchy in GPUs

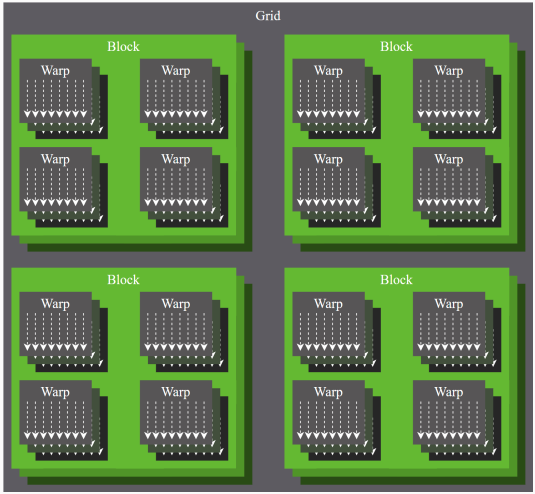
Overview of GPU Memory Architecture

- **Compute Device**:
  - Comprises multiple **Compute Units** (CU), each containing several **Processing Elements** (PE).
- **Private Memory**:
  - Dedicated to each PE, storing element-specific data.
- **Local Memory**:
  - Shared among PEs within the same CU, facilitating intra-unit communication.
- **Global/Constant Memory**:
  - **Global Memory**: Large, accessible by all PEs, used for cross-CU data sharing.
  - **Constant Memory**: Read-only, stores constants and configuration data.
- **Memory Caches**:
  - Intermediate cache layers improve access times for frequently used data.

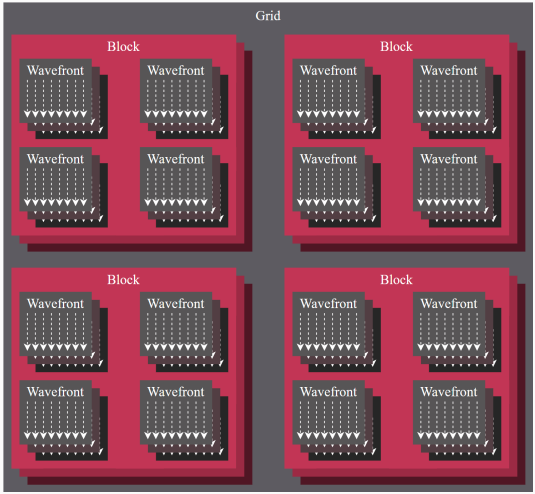
GPU Memory Architecture



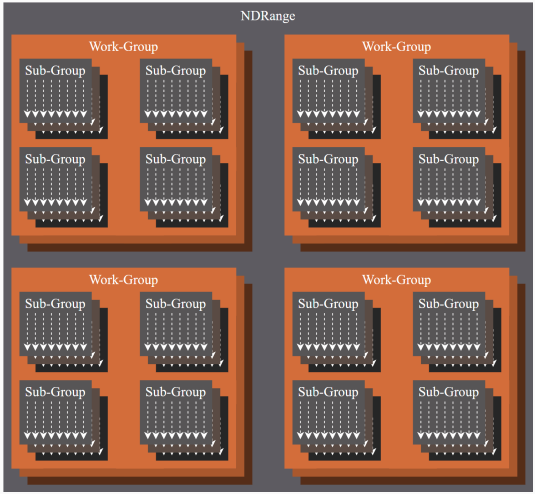
Execution Hierarchy: CUDA



Execution Hierarchy: HIP



Execution Hierarchy: OpenCL



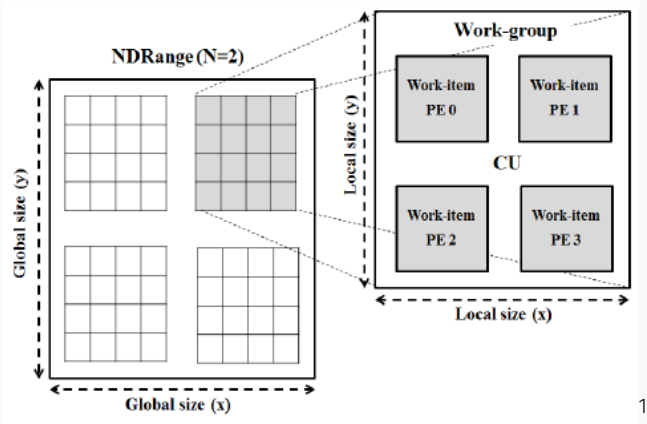
Execution Hierarchy in GPU Programming

- **NDRange/Grid**:
  - Defines the total computation space as a grid of threads.
  - Specifies the overall dimensions and workload size.
- **Work-Group/Block**:
  - Divides the NDRange into manageable chunks.
  - Each work-group operates independently, using shared local memory.
- **Sub-Group/Warp/Wavefront**:
  - Smaller units within a work-group that execute in lockstep.
  - Optimize data sharing and access patterns within the group.
- **Work-Items/Threads**:
  - The fundamental execution units in a GPU.
  - Each thread has private resources and can access shared local memory.

Key Points to Understanding the Execution Hierarchy

- **Efficient Parallelism**:
  - Optimize work-group sizes for load balancing.
  - Prevent resource underutilization with proper grid/block sizing.
- **Data Locality**:
  - Maximize shared memory usage to reduce global memory access.
  - Align data structures with hardware capabilities.
- **Synchronization**:
  - Use synchronization within work-groups to manage dependencies.
  - Minimize unnecessary synchronization to avoid bottlenecks.
- **Memory Access Patterns**:
  - Coalesced memory access enhances efficiency.
  - Optimize data layout for better performance.
- **Scalability**:
  - Design for scalability across different GPU architectures.
  - Ensure kernels can adapt to varying numbers of compute units.

Launching Kernel

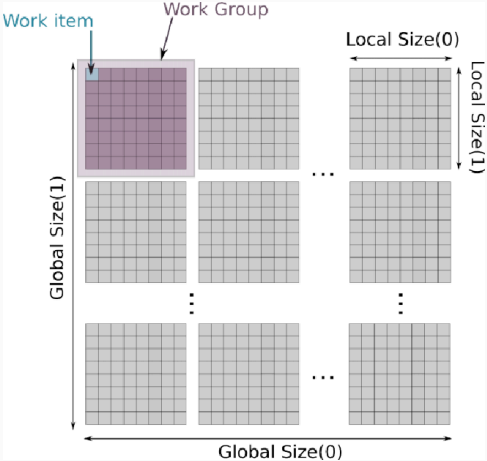


15

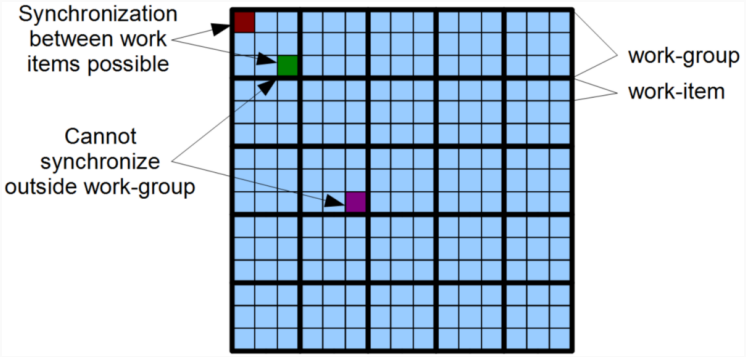
[15] From “Design of OpenCL Framework for Embedded Multi-core Processors”

Index Space

From “An Introduction to the OpenCL Programming Model” by Jonathan A. Thompson



Synchronization Possibilities



From An Introduction to the OpenCL Programming Model by Jonathan A. Thompson

Implementing atomic synchronization mechanisms is an advanced topic

GPU in a Nutshell

- Just think of the GPU as a CPU with lots of threads executing SIMD instruction
- GPU cannot allocate memory
- GPU programs are pretty boring:
  - Use Shared Local Memory[16] when possible to not read from the Main Memory
  - Be careful of register pressure
  - “Nothing Special” memory access[17]
  - GPU are fast because they force you to NOT synchronize between threads that live in difference work-group. Freedom versus Performance.
  - No branch prediction

[16] Shared Memory in CUDA, or just rely on cache oblivious algorithm

[17] “coalescing memory” == Don’t do random access / gather. “vectorization” strategies: contiguous versus strided access.

Programming Model API / Runtime

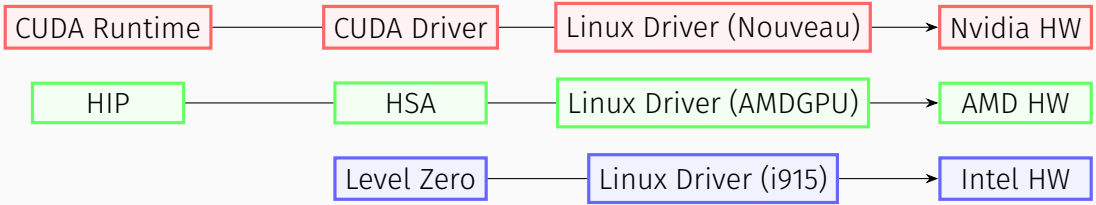
Programming Model API / Runtime

Runtime and Runtimes

Just a layer of Programming Model / Runtime!

"All problems in computer science can be solved by another level of indirection."
David Wheeler

Proprietary / Native Tool-chains (put Python Anywhere you like)

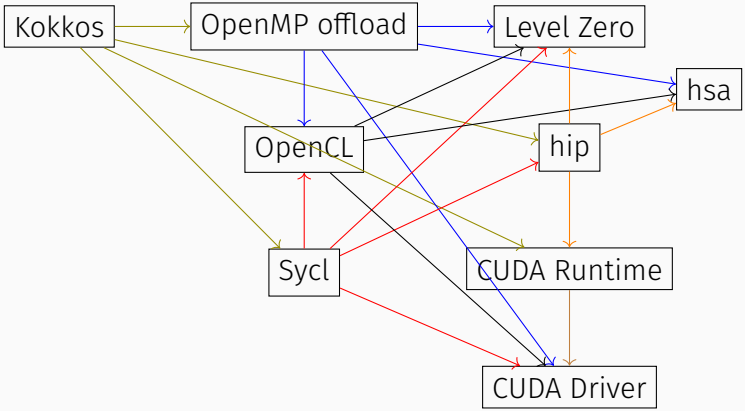


Comments on this diagram

- The Linux Drivers are the “lower-level” we will discuss. Huge effort to implement.[18]
- The level on top (CUDA Driver, HSA, Level Zero) abstracts away a little bit more of the hardware, but still provides a lot of control [19]
- The last level (CUDA Runtime, HIP) are “fully” hardware independent

[18] See nice blog post about the Linux M4 drivers
[19] Sweet spot to write higher-lever runtime

Taking About Higher-level Programming Model[20]



[20] Where is my Vulkan! Sorry gamer people... And thanks Valve for Direct3D -> Vulkan

Example of Paths

- OpenCL -> *[21]
- OpenMP Offload -> HSA
- Kokkos -> CUDA Runtime -> CUDA Driver
- HIP -> CUDA
- HIP, CUDA -> L0[22]

[21] Yes, I Like OpenCL... Soon you will too!
[22] Maybe more surprising, we will talk about this more at the end if we have time and interested

Hierarchy of Programming Model

- In short we have a “High Level” programming model. Used by Application.
- A “low-level” programming model that the high-level runtime is written with
- Each layer of abstraction is a trade-off between flexibility/performance and convenience/productivity

All of this is **relative** to who you are talking with.

“Sad” Truth

- No technical reason for having so much “intermediate” programming model
  - hipcc was a perl script that did ‘s/cu/hip/g‘ to avoid copyright infringement[23]
- Always hard to have a standard (*insert XKCD*)
- OpenCL is the standard, but low-adoption by vendors
- **Please don’t let vendors make the same mistake with new ML accelerators!**

[23] Not a lawyer, but the “recent” supreme court Google vs Oracle may help

Controversial Opinion

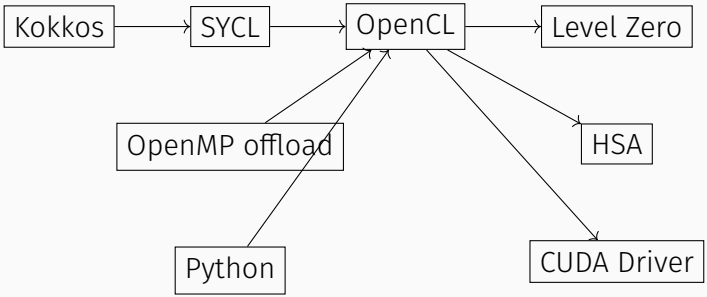
HIP and CUDA Runtime should not be used anymore.

- Too low-level for Applications that want to use nice C++ construct[24]
- Too high-level for people who have advance use-cases.

Kokkos, SYCL, OpenMP Offload already bypass HIP / CUDA runtime, so no "overhead" by using those programming models

[24] Come on, who wants to cast the output of malloc...

My Dream World



You are young and not yet totally jaded, so I share my dream with you!

Programming Model API / Runtime

Main concepts (shared)

Side note: They all have good old C API[25]

```
1  // Get number of platorm
2  cl_uint platformCount;
3  clGetPlatformIDs(0, NULL, &platformCount);
4  cl_platform_id* platforms = malloc(sizeof(cl_platform_id) * platformCount);
5  // Populate the newly allocated  array
6  clGetPlatformIDs(platformCount, platforms, NULL);
```

[25] I Don't check error code, because I'm not that of a great programmer...

What are a GPU Runtime's Main Goals?

1. Find devices
2. Load/Compile your kernels
3. Create Queue / Steam
4. Allocate GPU Memory
5. Execute commands (data-transfer, kernels)
6. Synchronize

Not that hard! Tedious but Simple[26]

[26] If only everything was like this...

Device. Easy

```
1  // OpenCL
2  cl_int clGetDeviceIDs(
3      cl_platform_id platform,
4      cl_device_type device_type,
5      cl_uint num_entries,
6      cl_device_id* devices,
7      cl_uint* num_devices);
8  // Level Zero
9  ze_result_t zeDeviceGet(ze_driver_handle_t hDriver,
10                         uint32_t *pCount,
11                         ze_device_handle_t *phDevices)
12 // Cuda Driver
13 CUresult cuDeviceGetCount ( int* count )
14 CUresult cuDeviceGet ( CUdevice* device, int ordinal)
15 // Cuda Runtime
16 cudaError_t cudaGetDeviceCount ( int* count )
17 cudaError_t cudaGetDevice ( int* device )
```

Queue / Stream

- Device are “explicit” / “separate object” / considered as accelerator.
- Queue/Stream[27] are the concept/object used to dispatch work to the device.

[27] Trivia, difference between queue and stream? Spoiler, answers on the next slide

Differences between programming models

- In L0, queues are out-of-order by default[28]
- In CUDA runtime and driver and HIP, streams are in-order[29]
- In OpenCL, they can be both
- In HSA, it's a ring buffer of packets

Out of order, you said?

- Out-of-order, mean kernel is free to re-order kernel execution... and concurrent execution is a re-ordering!
- Can be a source of error (and poor performance if not used)

[28] The latest L0 version we know have a `ZE_COMMAND_QUEUE_FLAG_IN_ORDER` flags.
[29] For more complex use cases, use cuda-graph

Queue / Stream: What to Submit?

Submit “command” to queue/stream. Commands can be

- Kernel Submission
- Memory Copy
- Synchronization (fence, barrier, event...)
- ...

Loading/Creating GPU Kernel

- Your code was split between hosts and GPU code (as we showed)
- Your kernels need to loaded by the GPU runtime!

```
1 clCreateProgramWithSource
2 clCreateProgramWithIL
3 clCreateProgramWithBinary
4 zeModuleCreate (   ZE_MODULE_FORMAT_IL_SPIRV | ZE_MODULE_FORMAT_NATIVE)
5 cuModuleLoad
```

Command Submission

On bad API (but convenient to use), commands can be submitted blocking manner

```
CUresult cuMemcpy ( CUdeviceptr dst, CUdeviceptr src,
                    size_t ByteCount ) // Where is my stream?!
}
```

On good one they asynchronously by default

```
CUresult cuMemcpyAsync (CUdeviceptr dst, CUdeviceptr src,
                        size_t ByteCount, CUstream hStream)

ze_result_t  zeCommandListAppendMemoryCopy(
  ze_command_list_handle_t hCommandList, void *dstptr, const void *srcptr,
  size_t size,
  ze_event_handle_t hSignalEvent,
  uint32_t numWaitEvents, ze_event_handle_t *phWaitEvents)
```

Hammering Down on the Async Concept

- Asynchronous execution just mean the CPU is free to continue to do other thing (like submitting other kernel). The call is not blocking / synchronous.
- The fact that the GPU can potentially execute multiple command concurrently is conceptually orthogonal to async. [30]
- Async mean you need to explicit synchronize, and handle dependency between commands.

[30] One Can use multiple MPI process, or Multiple Thread, ...

Command Submission Tips

- Async is a common source of error
- Blocking is a common source of poor performance

Synchronization: Event / Barrier

If it's asynchronous... you need to synchronize (insert "D'oh!" homer meme)

- via Event (specify dependencies for fine grained synchronization)
- via Barrier (for coarse synchronization)

Fine Grained: Event

OpenCL, L0[31]:

```
zeCommandListAppendMemoryCopy(..., &e1) // e1 will be signaled at completion
zeCommandListAppendMemoryCopy(..., &e2) // e2 will be signaled at completion
zeCommandListAppendMemoryCopy(..., 2, {e1,e2}, &e3) // wait for e1 and e2, e3 will
↪  be signaled at completion
zeEventHostSynchronize(e3) // And then finaly synchronization the leaf
```

CUDA, Hip:

```
hipMemcpyAsync // Hip, Cuda have optional Async, default blocking
hipEventRecord
hipEventSynchronize
```

(for more fancy use cases, use cuda-graph)

[31] So elegant

Coarse Grain: Barrier

Wait on queue / stream (wait until all the work has been done)

```
1  zeCommandQueueSynchronize
2  cudaStreamSynchronize
3  cudaDeviceSynchronize  // Whoa? Don't do that
```

Coarse grain. Use with caution because too much synchronization is bad.

Programming Model API / Runtime

Notes on Performance

Async is Key to good performance

- You want to keep the GPU busy
- When the GPU is computing something, the CPU should start preparing the next batch of work
- Importance of asynchronously

Async: [OpenMP][AMDGPU] Switch host-device memory copy to asynchronous version (real thing: `https://reviews.llvm.org/D115279`)

Overlapping is Key to good performance

- PCI is damn slow![32]
  - PCI 64 GB/s (unidirectional)
  - HBM 1 TB/s
  - GPU 50 TFlops+[33]
- Recompute is better than to load
- Overlap compute and data-transfers
- PCI is bidirectional so please do H2D and D2H at the same time!
- Avoid over-synchronization!

[32] And for integrated architectures, you have NUMA so same things... Data-movement will always be more expensive than compute

[33] And a few billions times more if you believe nvdia marketing slide and fantasy math

So how to achieve concurrency?

Importance of asynchronously![34]

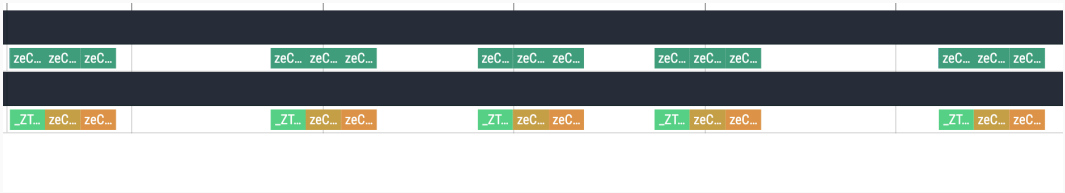
- Extract Maximum parallelism opportunity from your apps (this is the really hard part)
- Submit kernels to multiple stream / queue
- Submit kernels to an out-of-order queue.
- Synchronize: Not too much, not too little, just right.

[34] Or use multiple thread / process but this is cheating

Example

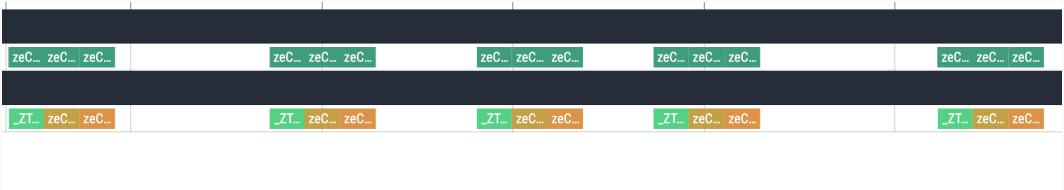
- 3 Independent Kernel To submits (D2H, Kernel, H2D)
- You can imagine as computing current iteration, moving back data of previous iteration, prefetching data for the next iteration

Example: Vibe-coded Bad



Example: Vibe-coded Bad-Problem, solution

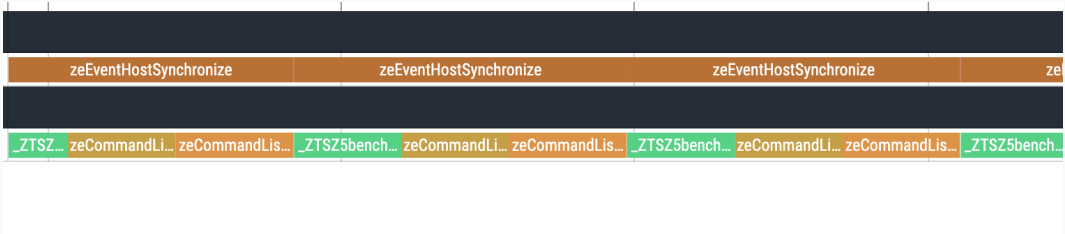
Gap in the timeline... And too much synchronization!



Solution:

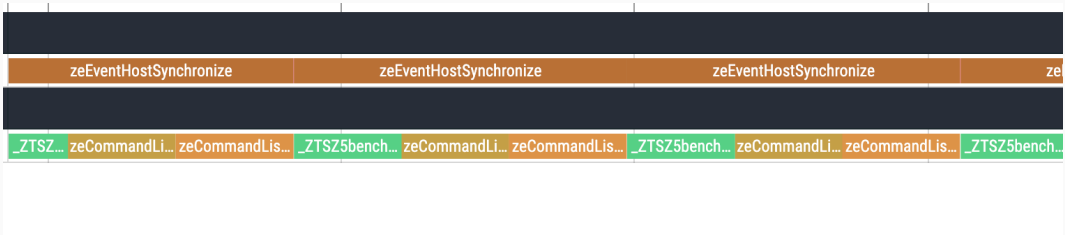
- Use Async API, so with some luck your command will be long enough that you can submit new one in the meantime.
- Stop Synchronize too much

Example: Bad Blocking



Example: Bad Blocking

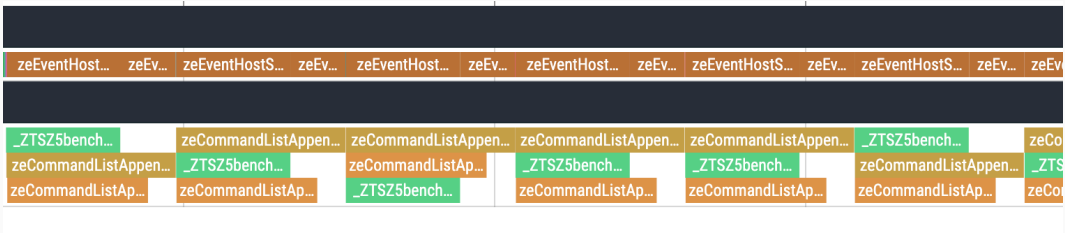
All kernel serialized. Not that Bad gpu utilization but we can do better. (the code is using In-order queue).



Solution: Extract more concurrency from your code! Out-of-order queue, or more multiple in-order queues.

Example: Good

Amazing Concurrency of Kernel, H2D, and D2H (3x buy using a out-of-order queue)



(The picture is more zoomed-in, and maybe misleading; the runtime of the apps is 3 times faster)

Programming Model API / Runtime

Memory Allocation

Types of GPU Memory[38]

- Device Memory: Accessible only on the particular device[35]
- Shared Memory: Accessible by both the host and the device[36]
  - This may impact performance, Different migration strategies
  - Can be migrated via prefetching[37]
- Host memory
  - “Pinned” memory. CPU memory but has been registered by the runtime.
  - May required for some optimizations / performance
- Malloc-ed Memory

[35] Read the documentation to know if it’s accessible by OTHER devices. Context, wink, wink

[36] Nvidia calls it “Managed”

[37] Do not confuse with prefetch of memory inside a kernel

[38] OpenCL has buffer, but lets not go that way...

But future GPUs will be integrated!

- Doesn’t matter,
- NUMA is bad, Locality is good.
- Please don’t use shared everywhere...

Best Case Scenario

- All data fit on the GPU
- Move everything over
- Do a ton of computation
- Move back results

You should aim for this. If you cannot, we will discuss other strategies latter.[39].
Memory transfers are expensive. Don't do it! Or at least try...

[39] Please not that it's the same in CPU. Keep data in cache

Programming Model API / Runtime

Kernel Submission

Kernel: CUDA Runtime / HIP

Magic / Syntactic sugar[40]

```
1 mykernel<<<blocks, threads, shared_mem, stream>>>(args);
```

But that just call HSA / Cuda Driver behind the scenes.

[40] A new syntax just to avoid people being portable...

Behind the seen: OpenCL, CUDA

```
cl_int clEnqueueNDRangeKernel(cl_command_queue command_queue,
                              cl_kernel kernel,
                              cl_uint work_dim,
                              const size_t *global_work_offset,
                              const size_t *global_work_size,
                              const size_t *local_work_size,
                              cl_uint num_events_in_wait_list,
                              const cl_event *event_wait_list,
                              cl_event *event)

CUresult cuLaunchKernel ( CUfunction f, unsigned int  gridDimX, unsigned int
↪  gridDimY, unsigned int  gridDimZ, unsigned int  blockDimX, unsigned int
↪  blockDimY, unsigned int  blockDimZ, unsigned int  sharedMemBytes, CUstream
↪  hStream, void** kernelParams, void** extra )
```

Behind the seen: Level zero, HSA

```
1
2  // Similar in L0, Count versus Size, and by kernel
3  zeKernelSetGroupSize(hKernel, groupSizeX, 1, 1);
4  ze_group_count_t groupCount = { numGroupsX, 1, 1 };
5  zeCommandListAppendLaunchKernel(hCommandList, hKernel, &groupCount, NULL, 0,
   ↪  NULL);
6
7  // HSA you get a packet from queue and then signaling, but still same idea
8  typedef struct hsa_kernel_dispatch_packet_s { uint16_t header ;
9  uint16_t setup;
10 uint16_t workgroup_size_x ; uint16_t workgroup_size_y ; uint16_t workgroup_size_z;
   ↪  uint16_t reserved0;
11 uint32_t grid_size_x ;
12 uint32_t grid_size_y ;
13 uint32_t grid_size_z;
14 uint32_t private_segment_size; uint32_t group_segment_size;
15
```

Programming Model API / Runtime

Recap of those section

Summary of Runtime

- HIP, CUDA (runtime, driver), L0 kind of all the same
- Always: “Discrete” device, load kernel, submit command async, synchronize. If not, it’s just abstracted away
- Push your vendor/institution/PI to use a standard (OpenCL).
- DON’T WRITE YOUR OWN PORTABILITY LAYER! (pretty please)[41]

[41] Just use SYCL...

Summary

- You see, all the programming model are same. And lot of Bridge between them!
- Create Kernel, Queue, Execution, Synchronize
- Some are less verbose more high level (HIP/CUDA runtime) but you lose some flexibility[42]
- IMO HIP/CUDA runtime are in a weird intermediate level and should never be used.

[42] And need to deal with some state-machine...

But does this matter? This sounds trivial

Real Time Experience (controversial)

- Experience: The runtime performance is far more important than the kernel performance
- Improving Kernel performance will give you a few percent; doing too much data-transfer will slowdown your code 100x.

Conclusion

- GPUs do not have the same ISA as CPU, so two compilation phases
- GPU are fast because they are simple $\rightarrow$ they are just a big SIMD 10k+ threads CPU
- Lots of good tutorials online for GPU Kernel Programming

Summary

Summary

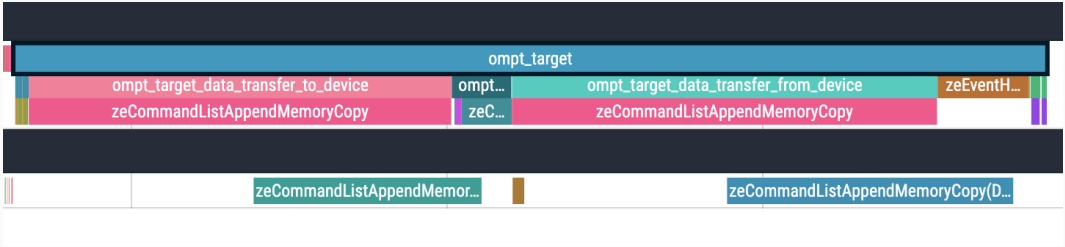
- OpenCL, L0, Cuda Driver -> All The same
- C API. “Double source” (CPU and Accelerator sources).
- Create queue, Kernel Creation, Submission Command, Synchronization Host <-> GPU
- Thinks of GPU are SIMD Machine
- All GPU the same
- Abstraction are powerful.

Example and Q&A

OpenMP: Now tell me what it does

```
1  #pragma omp target parallel for map(to: B[0:N]) map(from: A[0:N])
2  for (int i=0; i < N; i++)
3    A[i] = B[i];
```

Proof


ompt_target
ompt_target_data_transfer_to_device
ompt…
ompt_target_data_transfer_from_device
zeEventH…
zeCommandListAppendMemoryCopy
zeC…
zeCommandListAppendMemoryCopy
zeCommandListAppendMemor…
zeCommandListAppendMemoryCopy(D…

Any questions? If not I will show you some code...