Lecture 20: Introduction to GPU Computing

Vivek Kumar

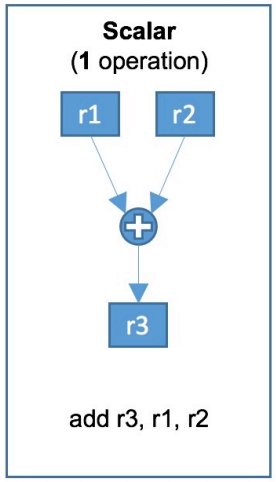
Computer Science and Engineering

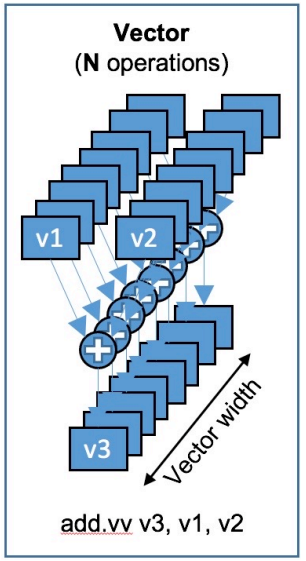
IIIT Delhi

vivekk@iiitd.ac.in

Last Lecture (Recap)

- SIMD vector extensions
  - Special registers at each core that support instructions to operate upon vectors values
- Limitations
  - Loop size should be countable at runtime
  - Loop iterations should not have different control flow
  - Loop iterations should be independent
  - Loop should only use basic math functions
  - Only a single arithmetic type operation
  - Should not have non-contiguous memory accesses
  - Unsupported data-dependencies
    - Read-After-Write: A[i] = A[i-1] + 1
    - Write-After-Write: A[i%2] = B[i] + C[i]
  - Supported data-dependencies
    - Write-After-Write: A[i-1] = A[i] + 1
    - Read-After-Read: A[i] = B[i%2] + C[i]





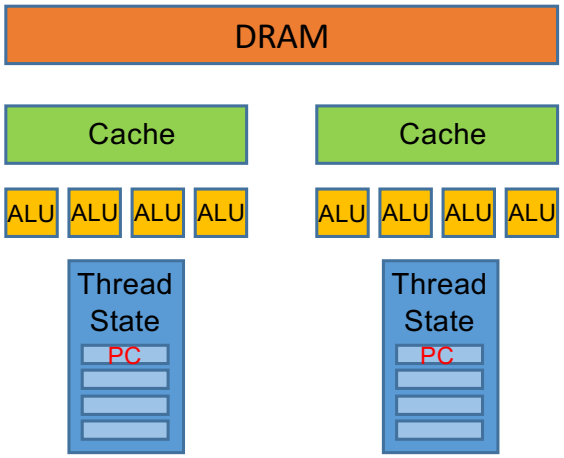

```
#include “vectorclass.h”
int A[1024], B[1024], C[1024];
void sum() {
  Vec8i Av;
  for (int i=0; i<1024; i+=8) {
    Vec8i Bv = Vec8i().load(B+i);
    Vec8i Cv = Vec8i().load(C+i);
    Av = Bv + Cv;
    Av.store(A+i);
  }
}
```

Today's Class

- GPU architecture
- GPU programming

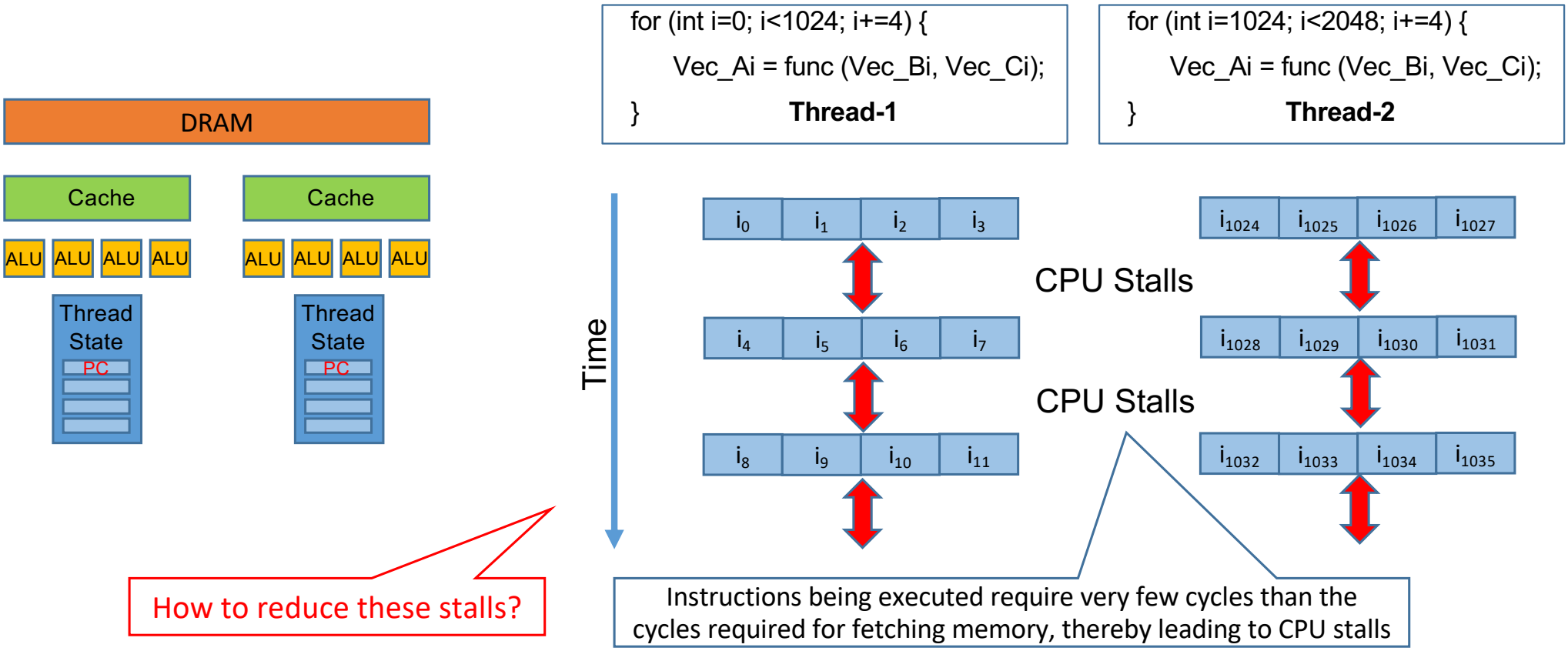
This lecture will give you a high-level overview of GPU architecture and a platform-neutral high-level library-based programming model for writing GPU programs that can compile with standard C++ compilers

Multicore CPUs with SIMD Support

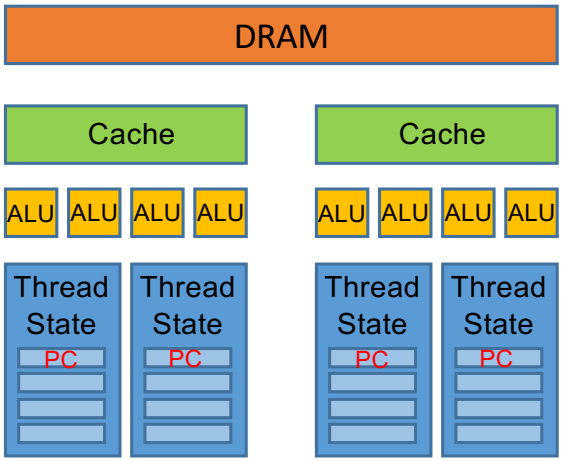


- Multicore processors are latency oriented!
  - How?
- Modern multicore processors have sophisticated cores to support general purpose computing
  - High core frequency for low latency operations
  - Large cache and prefetcher unit for improving memory access latency
    - Dynamically decide future memory accesses based on current access pattern to reduce CPU stalls
  - Superscalar capabilities allowing it to use Instruction Level Parallelism (ILP)
- They also support data parallel execution
  - Each physical core has bunch of ALUs and wide vector registers for SIMD operations

CPU Stalls in SIMD Execution

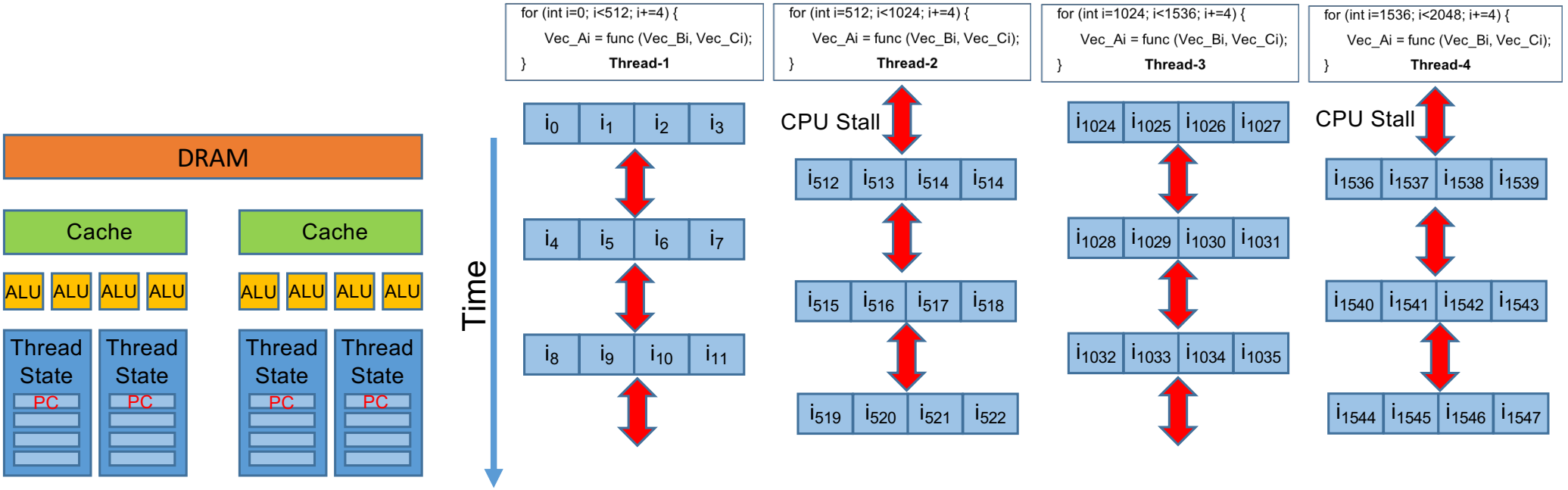


Using SMT for Hiding Stalls



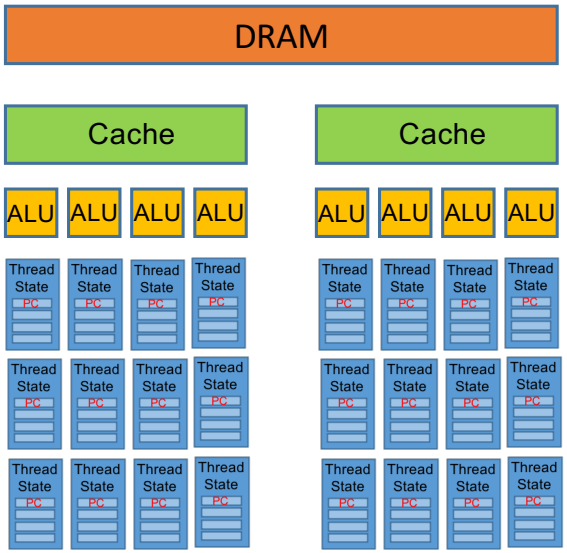
- **Two-way** SMT at each multicore (Simultaneous Multithreading)
  - Each SMT core has its own PC register, thereby allowing each core to simultaneously execute a completely different execution stream
  - Each SMT core has its own set of vector registers
  - Each SMT core pair share ALUs
  - Each SMT core pair can execute different set of SIMD operations (as they don't share PC register)

CPU Stalls in SIMD Execution



- Using SMT for hiding stalls
  - Thread-1 on Core-1 and Thread-3 on Core-2 completes the first iteration, and then stalls for memory fetch
  - Thread-2 on Core-1 and Thread-4 on Core-2 memory fetch has completed, hence they start their first iteration while Thread-1 and Thread-3 are blocked for memory fetch
  - Key idea here is to increase the number of hardware threads for hiding CPU stalls

How to Further Optimize SIMD Execution?



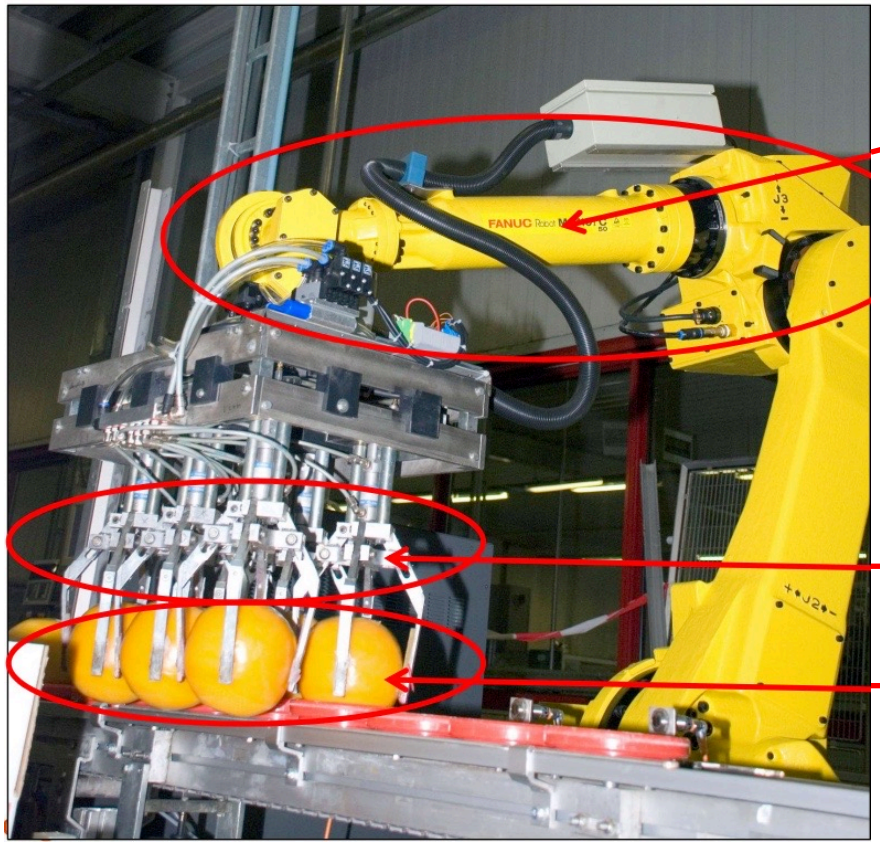
- Increase the number of hardware threads supported on each core
  - CPU stalls are significantly reduced
  - Improves the performance as the hardware schedule the threads instead of the OS
- Improve the memory bandwidth
  - As large chunks of memory addresses are being fetched from DRAM due to large number of threads
- But, won't these enhancements increase the complexity and cost of the multicore processor?

How to Design a Processor for SIMD?

- If we only have to run SIMD applications on a processor, then how to cut down the complexity of the processor?
  - Reduce core frequency and increase the number of cores
  - Support large number of hardware threads at each core
    - Requires a large amount of data, but stalls are hidden due to large number of threads
  - Cores have smaller cache
    - Large number of threads per core would operate on large amount of data, thereby requiring frequent DRAM accesses
  - Increase the number of ALUs per core and the width of SIMD registers
  - Group of threads could share a single PC register
    - Single Instruction Multiple Thread (**SIMT**)
    - Shared instruction cache
  - Support high bandwidth data transfer

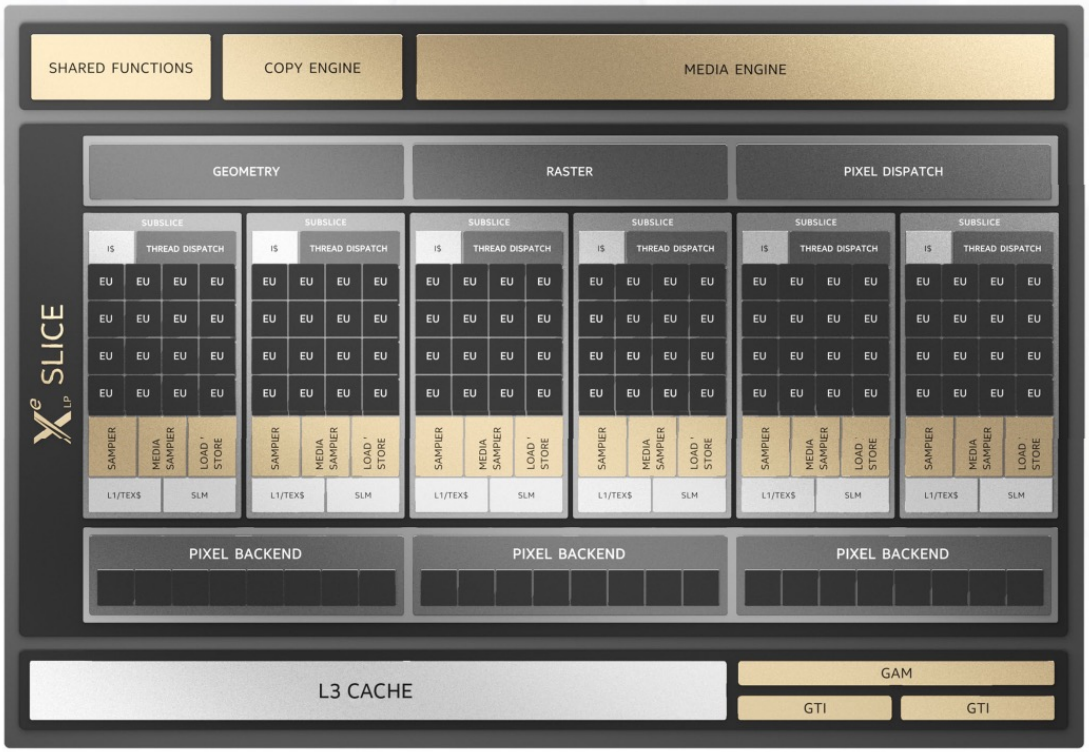
This is the design of a throughput oriented processor or a GPU

Mechanical Equivalent of a GPU



Slide credit: https://web.engr.oregonstate.edu/~mjb/cs575/Handouts/gpu101.1pp.pdf

Intel GPU Architecture



Intel's Iris® Xe single Slice

- Execution Unit (EU) is the smallest building block (same as a core in the CPU)
  - Operates at MHz level instead of GHz
  - Each EU supports 7-way SMT
  - Supports one 8-wide SIMD operation
- Each slice contains 6 subslice
  - 16 EUs at each subslice
  - Total FP32 SIMD operations per slice per cycle are 7x8x16x6 (=5376)
- Intel supports multiple slices in GPU

Source: https://www.intel.com/content/www/us/en/develop/documentation/oneapi-gpu-optimization-guide/top/xe-arch.html

NVIDIA GPU Architecture

Pascal GP100 single SM (Streaming Multiprocessor)

- CUDA-core is the smallest building block (akin to EU in Intel)
  - Operates at 1126 MHz
  - Each CUDA-core can process 32 data elements (FP 32) simultaneously (**warps**). Similar to 32-wide vector operation
    - Warp has a common PC (SIMT)
- Each SM (akin to subslice in Intel) has 32x2 CUDA-cores
  - How many SIMD operations per SM?
    - An SM can operate on 64 warps, i.e., each SM can process 32x64 FP32 data elements simultaneously
- GP100 has 56 SMs per GPU
  - Total FP32 that can be processed simultaneously are 32x64x56

Today's Class

- GPU architecture
- GPU programming

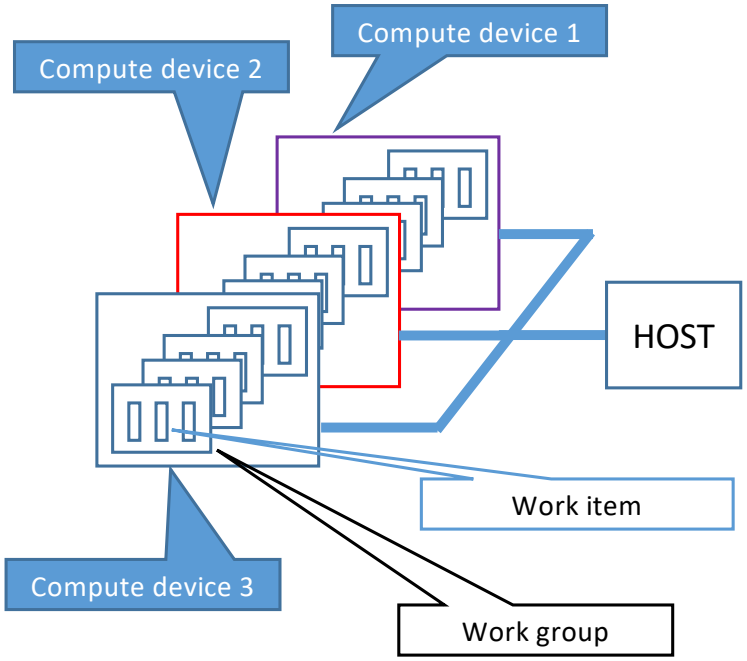
GPU Programming Template

1. Setup inputs on the host CPU
2. Allocate memory on the host CPU
3. Allocate memory on the GPU
4. Copy inputs from the host to GPU
5. Start GPU kernel
6. Copy output from the GPU to host

GPU Programming Model

- Vendor supported programming model
  - CUDA on NVIDIA GPUs
  - oneAPI on Intel GPUs
  - Provides high performance
  - Cannot compile with standard compilers (lacks portability)
- OpenCL is vendor neutral
  - Does not require any special compiler or compiler extensions
    - Works with standard C/C++ compiler
  - Provides direct access to underlying hardware (CPU, GPU, FPGA)
  - High portability
    - Same program can run on multiple device types
      - Although, performance may not be optimal without device specific tuning
  - Requires some serious effort for writing OpenCL programs

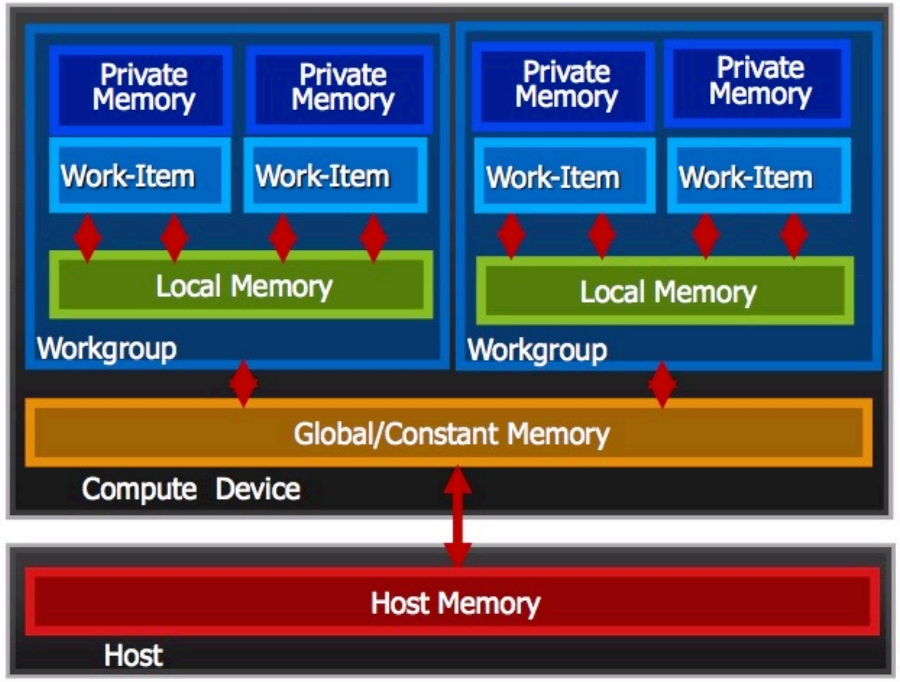
OpenCL Platform Model



- One host is connected to one or more OpenCL compute devices
  - Compute device is a processor (e.g., multicore processor or GPU)
- Each compute device is composed of one or more work groups (a.k.a. compute units)
  - Work group is analogous to a “core” in multicore processor, or SIMD vector register in CPU, or CUDA-core in a GPU
- Each work group is divided into one or more work items (a.k.a. processing elements)
  - Work item is analogous to a thread that execute code as SIMD

OpenCL Memory Model

- **Private Memory**
  - Per work-item
- **Local Memory**
  - Shared within a workgroup
- **Global/Constant Memory**
  - Visible to all workgroups
- **Host Memory**
  - On the CPU

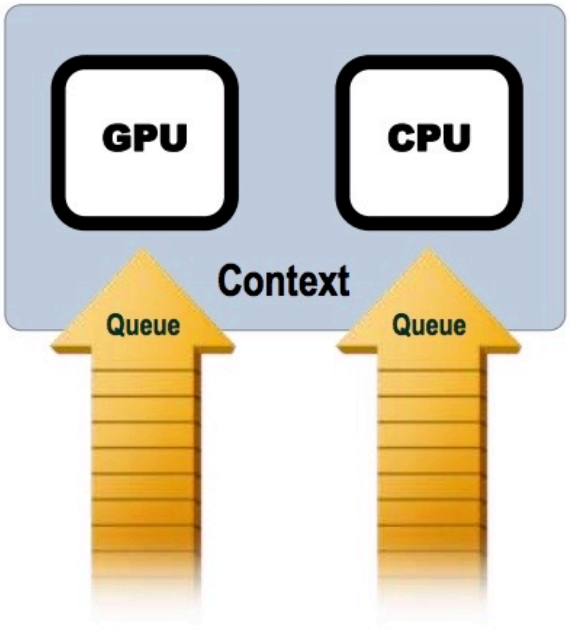


Memory management is Explicit

You must move data from host -> global -> local ... *and* back

OpenCL Execution Model

- **OpenCL application runs on a host which submits work to the compute devices**
  - **Context**: The environment within which work-items executes … includes devices and their memories and command queues
  - **Program**: Collection of kernels and other functions (Analogous to a dynamic library)
  - **Kernel**: the code for a work item. Basically a C function
  - **Work item**: the basic unit of work on an OpenCL device
- **Applications queue kernel execution**
  - Executed in-order or out-of-order

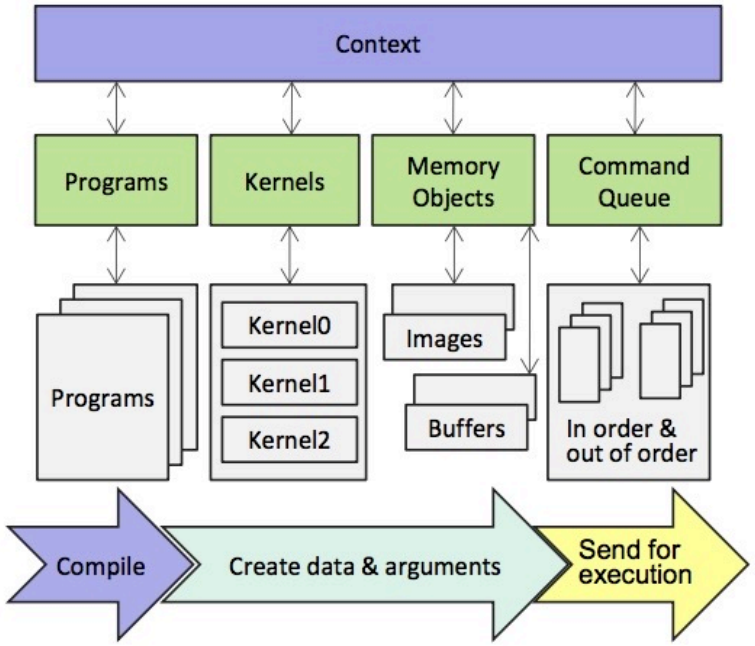


Allows independent kernels to execute simultaneously whenever possible, and thus keeps the GPU fully utilized

Executing OpenCL Programs

1. Query host for OpenCL devices
2. Create a context to associate OpenCL devices
3. Create programs for execution on one or more associated devices
4. Select kernels to execute from the programs
5. Create memory objects accessible from the host and/or the device
6. Copy memory data to the device as needed
7. Provide kernels to command queue for execution
8. Copy results from the device to the host

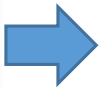
Similar to GPU programming template listed in Slide #13



OpenCL Kernel Example

```
void vector_addition(float* A, float* B,
                     float* C, int size) {
  for (int i=0; i<size; i++) {
    C[i] = A[i] + B[i];
  }
}
```

Using C/C++ in traditional way



```
__kernel void vector_addition(__global float* A, __global float* B,
                              __global float* C) {
  int i = get_global_id(0);
  C[i] = A[i] + B[i];
}
```

Using OpenCL data parallel loop

- Vector addition using OpenCL
  - The complete OpenCL program to compute vector addition could span to several hundred lines of **low-level code** as compared to the few lines of **simple code** in the traditional C/C++ program
    - See: https://www.intel.com/content/www/us/en/support/programmable/support-resources/design-examples/horizontal/vector-addition.html
    - **Low productivity!**

Boost.Compute for GPU Computing

- A header-only C++ library for GPU computing
  - Easy to use GPU programming APIs → High Productivity!
  - Provides a thin C++ wrapper over OpenCL APIs
  - Works with standard C++ compilers
  - Provides several ready-to-use optimized kernel implementations (e.g., binary_search, reduce, sort_by_key, etc.)
- Supports varieties of GPUs (Intel, NVIDIA, and AMD), as well as CPUs
- Caches OpenCL programs
  - Each OpenCL program (kernel) requires compilation and incurs overheads
  - Boost.compute stores frequently used kernels in a global cache
    - Reduces overheads by avoiding multiple compilation for the same kernel
- May not match the performance of natively supported GPU programming model (e.g., CUDA on a NVIDIA GPU) without tuning

Vector Addition using Boost.Compute

```
14 namespace compute = boost::compute;
15
16 int main(int argc, char** argv) {
17   int size = argc>1?atoi(argv[1]):1024*1024;
18   double time_gpu=0;
19   // lookup default compute device
20   auto gpu = compute::system::default_device();
21   // create opencl context for the device
22   auto context = compute::context(gpu);
23   // create command queue for the device
24   compute::command_queue queue(context, gpu);
25   // print device name
26   std::cout << "device = " << gpu.name() << std::endl;
27
28   int* a = new int[size];
29   int* b = new int[size];
30   int* c = new int[size];
31   std::fill(a, a+size, 1);
32   std::fill(b, b+size, 2);
33   // create 'a' vector on the GPU
34   compute::vector<int> vector_a(size, context);
35   // create 'b' vector on the GPU
36   compute::vector<int> vector_b(size, context);
37   // create output 'c' vector on the GPU
38   compute::vector<int> vector_c(size);  // TODO: Why cor
39   // copy data from the host to the device
40   compute::future<void> future_a = compute::copy_async(
41     a, a+size, vector_a.begin(), queue
42   );
43   // copy data from the host to the device
44   compute::future<void> future_b = compute::copy_async(
45     b, b+size, vector_b.begin(), queue
46   );
47   // wait for copy to finish
48   timer::kernel("asynchronous copy", [=]() {
49     future_a.wait();
50     future_b.wait();
51   });
```

Vector Addition using Boost.Compute

```
52   time_gpu+=timer::duration();
53   // Create function defining the body of the for-loop for carrying out vect
   or addition
54   BOOST_COMPUTE_FUNCTION(int, vector_sum, (int x, int y), {
55     return x + y;
56   });
57   // Launch the computation on the GPU using the command queue created above
58   timer::kernel("GPU kernel execution", [&]() {
59     compute::transform(
60       vector_a.begin(),
61       vector_a.end(),
62       vector_b.begin(),
63       vector_c.begin(),
64       vector_sum
65     );
66   });
67   time_gpu+=timer::duration();
68   // transfer results back to the host array 'c'
69   timer::kernel("copy from device", [&]() {
70     compute::copy(vector_c.begin(), vector_c.end(), c);
71   });
72   time_gpu+=timer::duration();
73   std::cout<<"Total GPU time = "<<1000*time_gpu<<"ms"<<std::endl;
74   //verify the computation
75   for(int i=0; i<size; i++) assert(c[i] == 3);
76   std::cout<<"Test Passed at GPU\n";
77   timer::kernel("CPU kernel", [=]() {
78     for(int i=0; i<size; i++) c[i] = a[i] + b[i];
79   });
80   std::cout<<"Speedup of GPU over CPU= "<<timer::duration()/time_gpu<<std::e
   ndl;
81   //cleanup
82   delete [] a;
83   delete [] b;
84   delete [] c;
85   return 0;
86 }
```

Reading Materials

- OpenCL
  - https://sites.google.com/site/csc8820/opencl-basics/opencl-concepts#TOC-Kernel-and-compute-kernel
  - https://www.khronos.org/assets/uploads/developers/library/2012-pan-pacific-road-show-June/OpenCL-Details-Taiwan_June-2012.pdf
- Boost.Compute
  - https://www.boost.org/doc/libs/1_80_0/libs/compute/doc/html/index.html#boost_compute.introduction

Next Lecture

- Heterogeneous parallel programming