Lecture 20: Introduction to GPU Computing

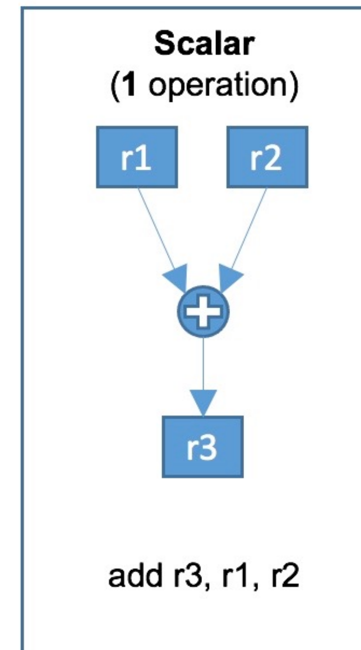
Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

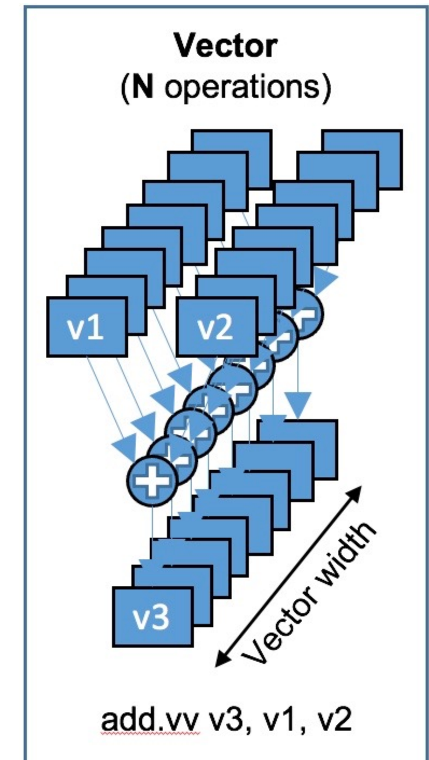
 © Vivek Kumar

Last Lecture (Recap)

- SIMD vector extensions
  - Special registers at each core that support instructions to operate upon vectors values

```
#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);
  }
}
```



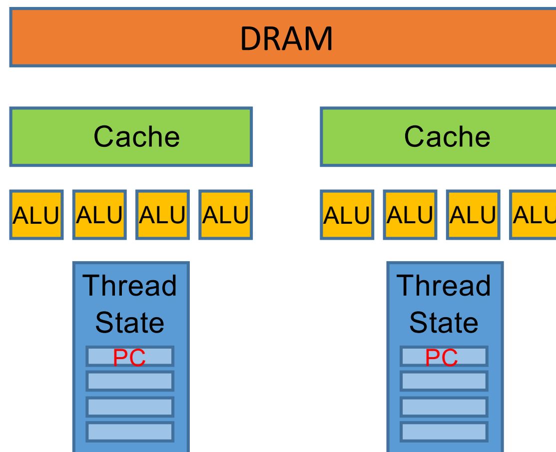


Today's Class

- GPU architecture
- GPU programming

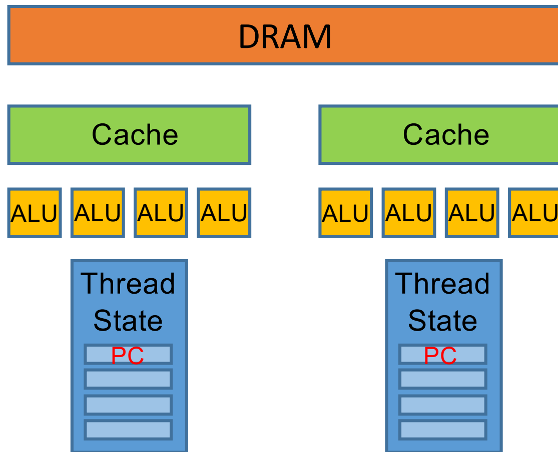
This lecture will give you a high-level overview of GPU architecture and a platform-neutral high-level library-based programming model for writing GPU programs that can compile with standard C++ compilers

Multicore CPUs with SIMD Support



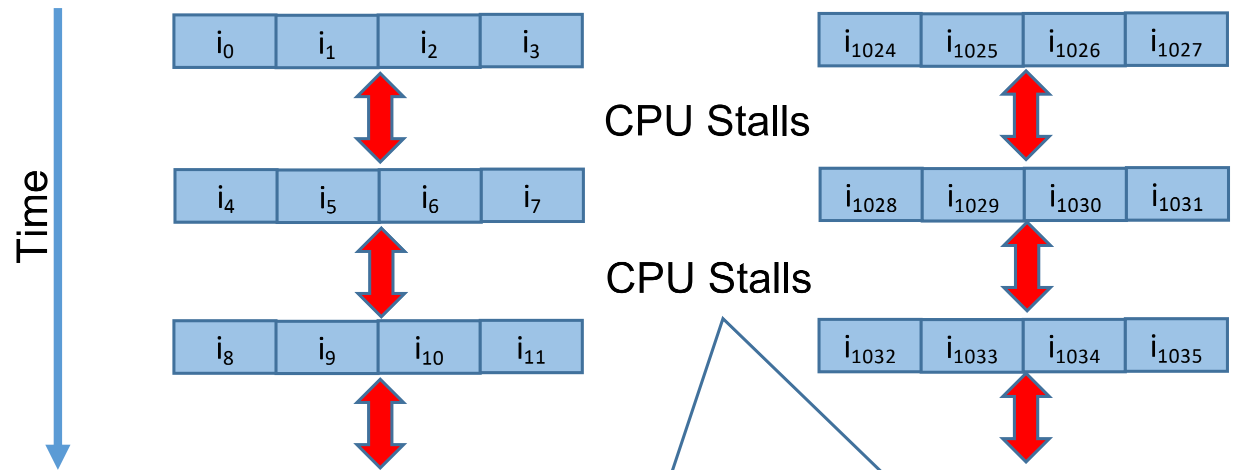
- Multicore processors are latency oriented!
  - How?
- Modern multicore processors have sophisticated cores to support general purpose computing
  - High core frequency for low latency operations
  - Large cache and prefetcher unit for improving memory access latency
    - Dynamically decide future 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




```
for (int i=0; i<1024; i+=4) {
    Vec4_Ai = func (Vec4_Bi, Vec4_Ci);
}
```
Thread-1

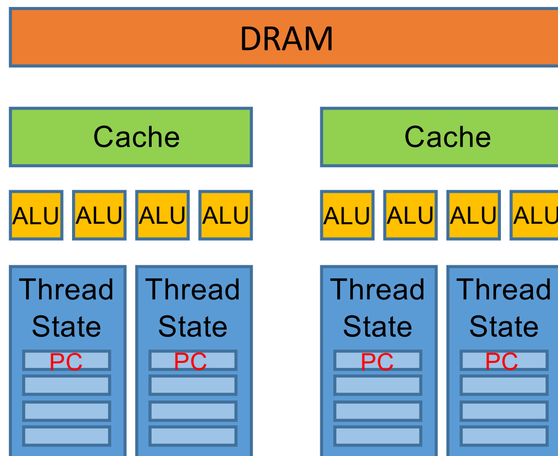
```
for (int i=1024; i<2048; i+=4) {
    Vec4_Ai = func (Vec4_Bi, Vec4_Ci);
}
```
Thread-2



How to reduce these stalls?

Instructions being executed require very few cycles than the cycles required for fetching memory, thereby leading to CPU stalls

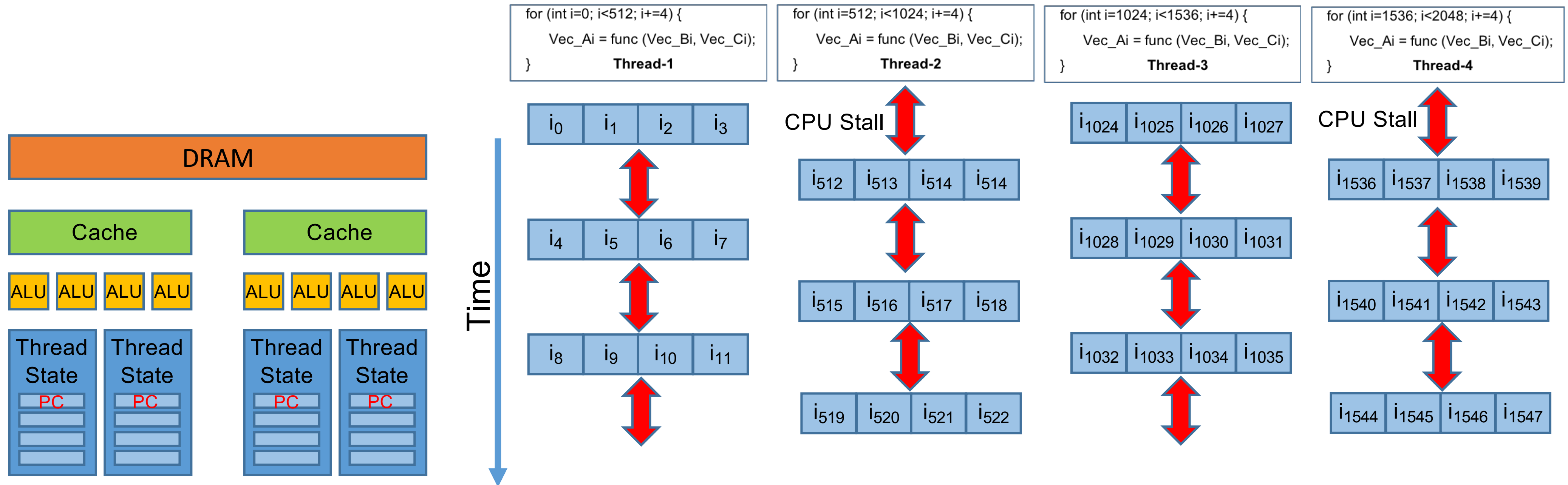
Using SMT for Hiding Stalls



- **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)

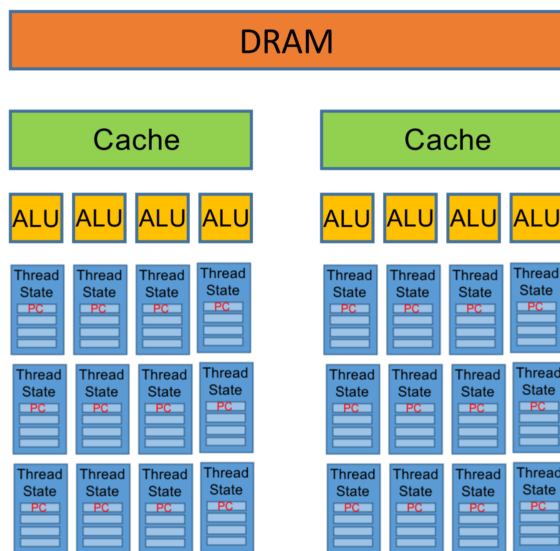
 © Vivek Kumar

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?



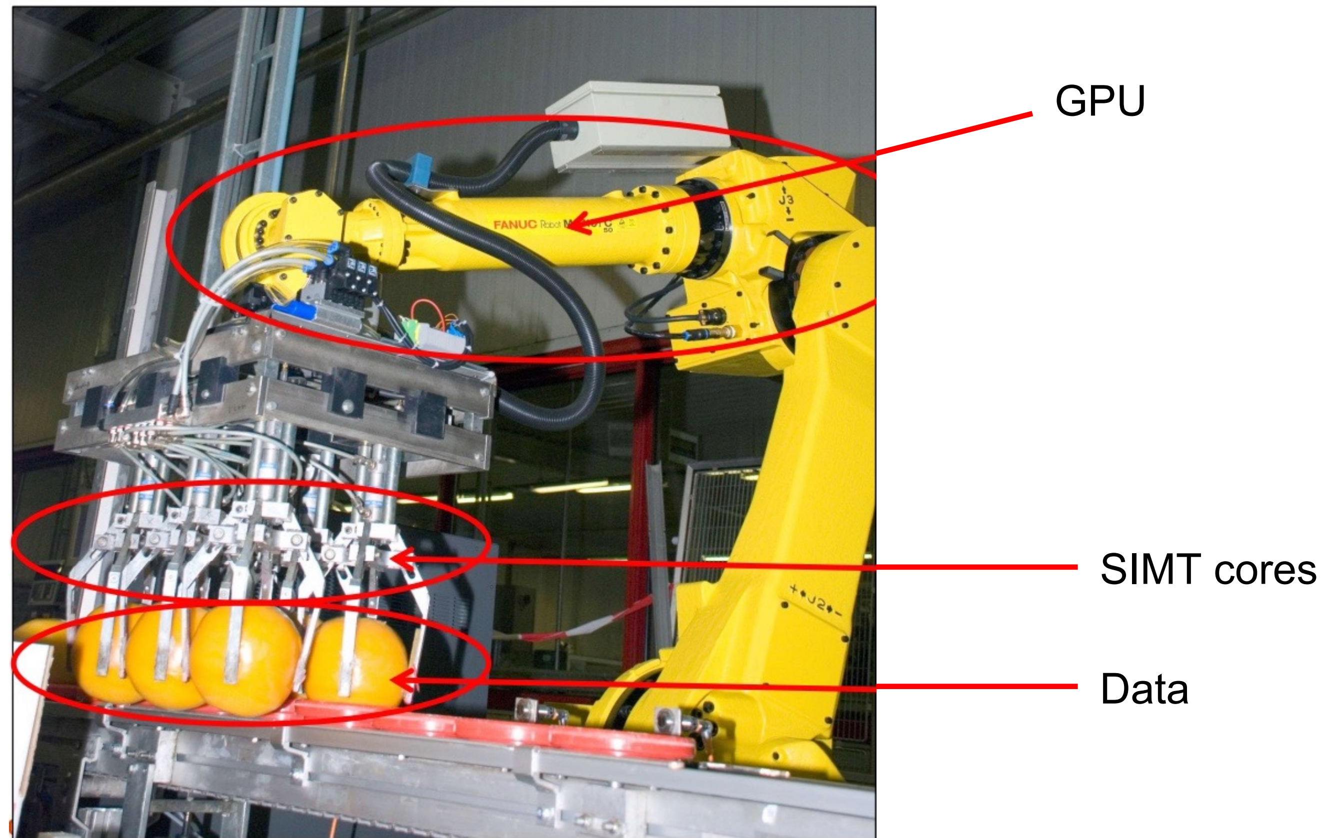
- Increase the number of hardware threads supported on each core
  - CPU stalls are significantly reduced
  - Improves the performance as the hardware schedule the threads instead of the OS
- Improve the memory bandwidth
  - As large chunks of memory addresses are being fetched from DRAM due to large 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 MHz level
  - 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

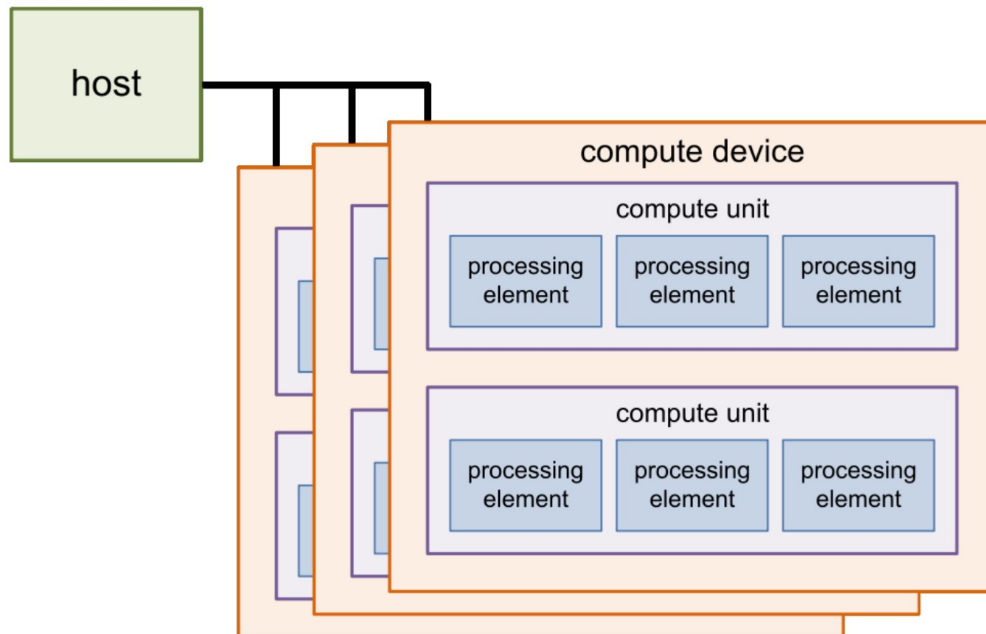
Today's Class

- GPU architecture
- GPU programming

GPU Programming Model

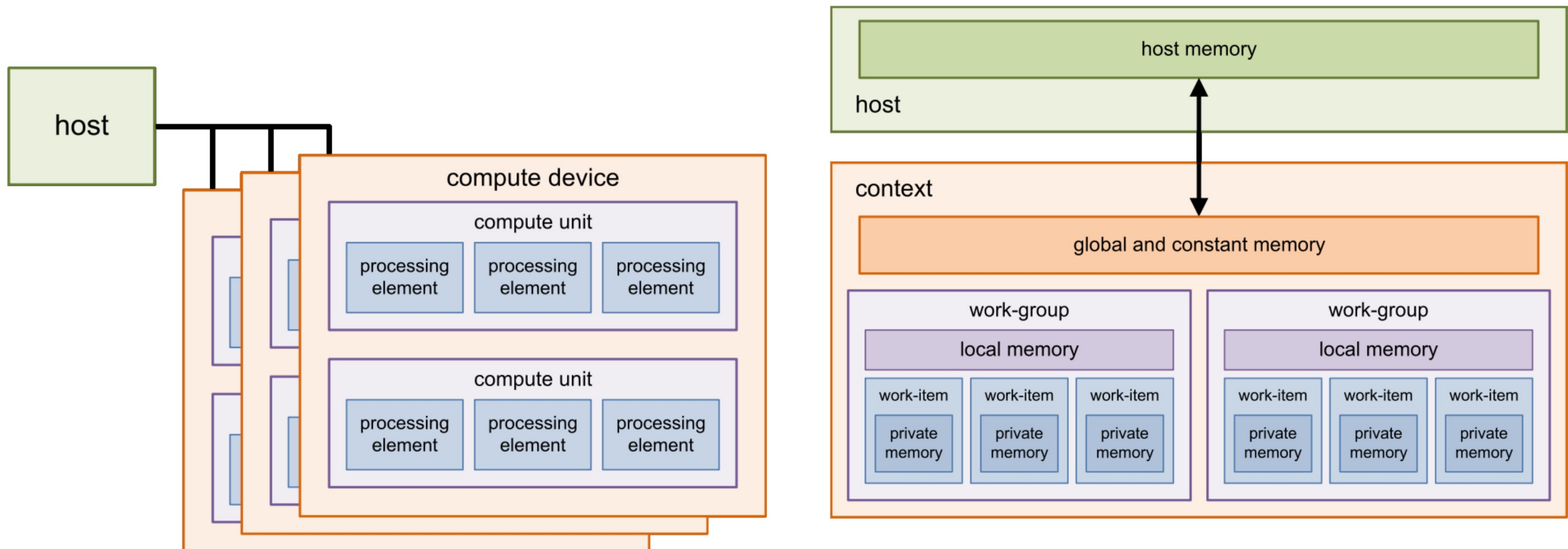
- Vendor supported programming model
    - CUDA on NVIDIA GPUs
    - oneAPI on Intel GPUs
  - Provides high performance
  - Cannot compile with standard compilers (lacks portability)
- OpenCL is vendor neutral
  - Does not require any special compiler or compiler extensions
    - Works with standard 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
  - Multicore processor, GPU, FPGA, etc
- Each compute device is composed of one or more compute units
  - Cores in multicore processor
  - CUDA-core in a GPU
- Each compute unit is divided into one or more processing elements
  - SIMD / SIMT execution

OpenCL Memory Model



Programming using OpenCL

- **Low productivity!**
- The complete OpenCL program 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 vector addition in OpenCL (LOC = 171): https://hpc.mediawiki.hull.ac.uk/Programming/OpenCL

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
- 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

```
#include <boost/compute.hpp>
#include <iostream>
#include <vector>
namespace compute = boost::compute;
compute::command_queue setup(compute::context &context,
                                              compute::device &device) {
   // Approach-1 (will fail if GPU is not detected)
   for(const auto &d : compute::system::devices()) {
      if(d.type() == CL_DEVICE_TYPE_GPU) {
         device = d;
         break;
      }
   }
   // OR Approach-2 (can run even on CPU if GPU not found)
   device = compute::system::default_device();

   context = compute::context(device);
   compute::command_queue queue(context, device);
   return queue;
}
```

1. **Setup (choose where to run)**
   a) Get available **platforms** (e.g., CPU, GPU vendors)
   b) Select a **platform**
   c) Get available **devices** (GPU/CPU)
   d) Select a **device**
2. **Create environment**
   a) Create a context (ties everything together)
   b) Create a command queue (to send work to device)

Vector Addition using boost.compute

```
#define SIZE 1024
int main() {
   // Step 1 & 2 Setup and create environment
   compute::device device;
   compute::context context;
   compute::command_queue queue = setup(context, device);
   // Step 3a. Manage memory (allocate host data)
   std::vector<float> A(SIZE), B(SIZE), C(SIZE); // Initialize
   // Step 3b. Create device vectors and copy host data to device
   compute::vector<float> dA(A.begin(), A.end(), queue);
   compute::vector<float> dB(B.begin(), B.end(), queue);
  // Step 3c. Create output vector at device
   compute::vector<float> dC(SIZE, context);



}
```

1. **Setup (choose where to run)**
   a) Get available **platforms** (e.g., CPU, GPU vendors)
   b) Select a **platform**
   c) Get available **devices** (GPU/CPU)
   d) Select a **device**
2. **Create environment**
   a) Create a context (ties everything together)
   b) Create a command queue (to send work to device)
3. **Manage memory**
   a) Create input memory objects on device
   b) Copy input data to device (via command queue)
   c) Create output memory object on device

Vector Addition using boost.compute

```
#define SIZE 1024
int main() {
    // Step 1 & 2 Setup and create environment
    compute::device device;
    compute::context context;
    compute::command_queue queue = setup(context, device);
    // Step 3a. Manage memory (allocate host data)
    std::vector<float> A(SIZE), B(SIZE), C(SIZE); // Initialize
    // Step 3b. Create device vectors and copy host data to device
    compute::vector<float> dA(A.begin(), A.end(), queue);
    compute::vector<float> dB(B.begin(), B.end(), queue);
   // Step 3c. Create output vector at device
    compute::vector<float> dC(SIZE, context);
   // Step 4. Prepare program (create OpenCL kernel)
    BOOST_COMPUTE_FUNCTION(float, add, (float x, float y), {
        return x + y;
    });
    // Step 5. Execute kernel (queue at device)
    compute::transform(dA.begin(), dA.end(), dB.begin(), dC.begin(), add, queue);
    // Step 6. Get result (copy output vector from device to host vector)
    compute::copy(dC.begin(), dC.end(), C.begin(), queue);
    return 0;
}
```

4. **Prepare program**
   a) Provide kernel code
   b) Compile program
   c) Create kernel (contained in the program)
5. **Execute kernel**
   a) Set kernel arguments
   b) Put kernel into command queue (for execution)
6. **Get results**
   a) Copy output data from device to host
   b) Clean up by releasing all resources (buffers, kernel, program, queue, context)

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

- Power management in multicore processors