GPU Programming I

Andrey Alekseenko

KTH Royal Institute of Technology & SciLifeLab

About this course

- Goals:
  - Understand why and when to use GPUs
  - Become comfortable with key concepts in GPU programming
  - Acquire an overview of different software frameworks
  - Learn the basics of SYCL, enabling you to quickly become a productive GPU programmer
- Prerequisites:
  - Knowledge of C++

- This course is based on ENCCS online course “GPU Programming: why, when and how?” (CC-BY 4.0)

Programming experience?

- C
- C++
- Fortran
- Python
- Julia
- ???

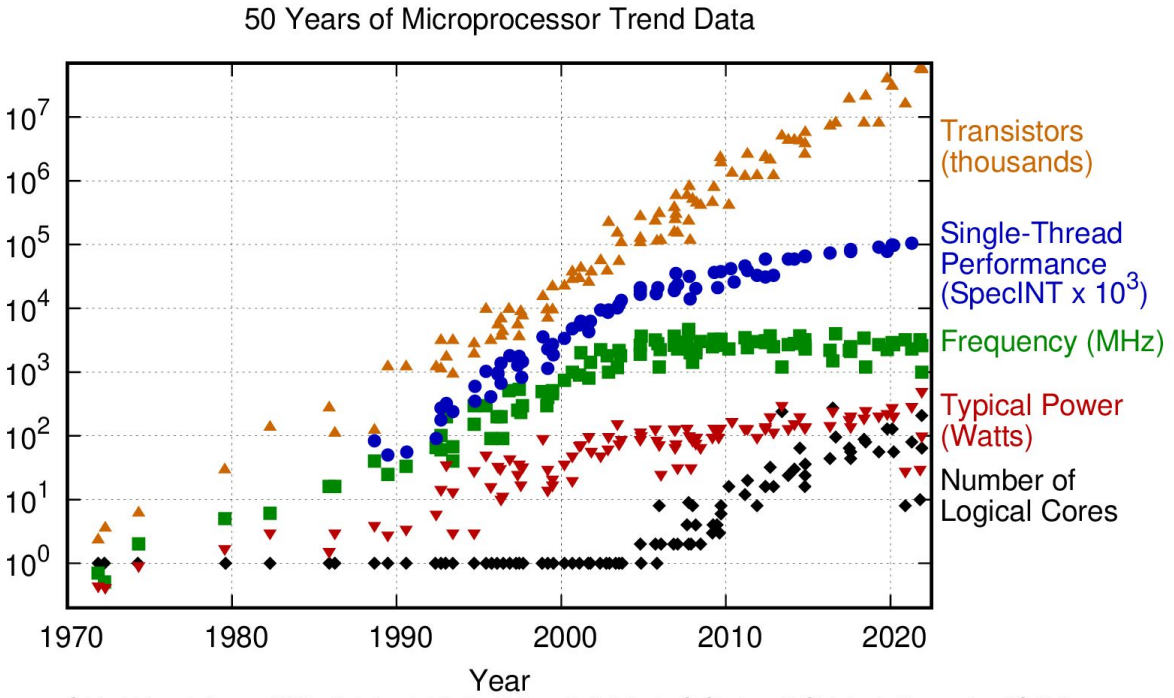
- POSIX Threads and the link
- OpenMP
- MPI
- GPU
- FPGA
- ???

Your goals?

Why GPUs?

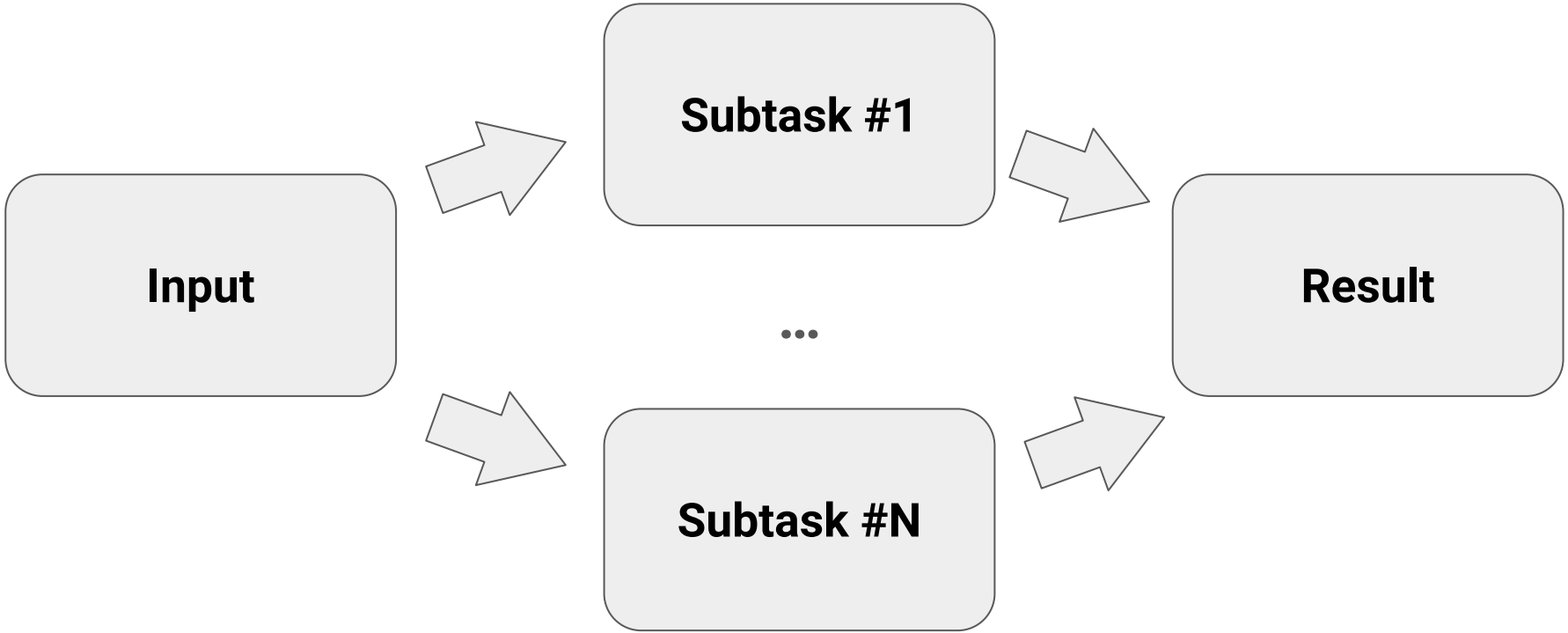
Moore’s law

- “Number of transistors in an integrated circuit doubles about every two years”
- Before ~2005: increasing single-core performance
- After ~2005: increasing the number of cores



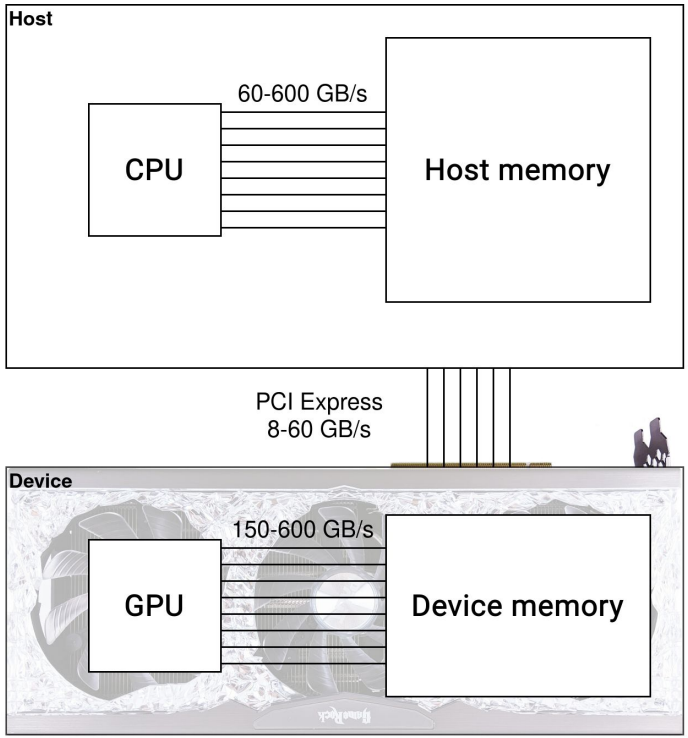
https://github.com/karlrupp/microprocessor-trend-data

Computing in parallel



Graphics processing units

- GPU: Most common type of *accelerator*
- Specialized hardware
  - Highly parallel
  - Not self-sufficient
- Separate memory
- Different programming paradigm

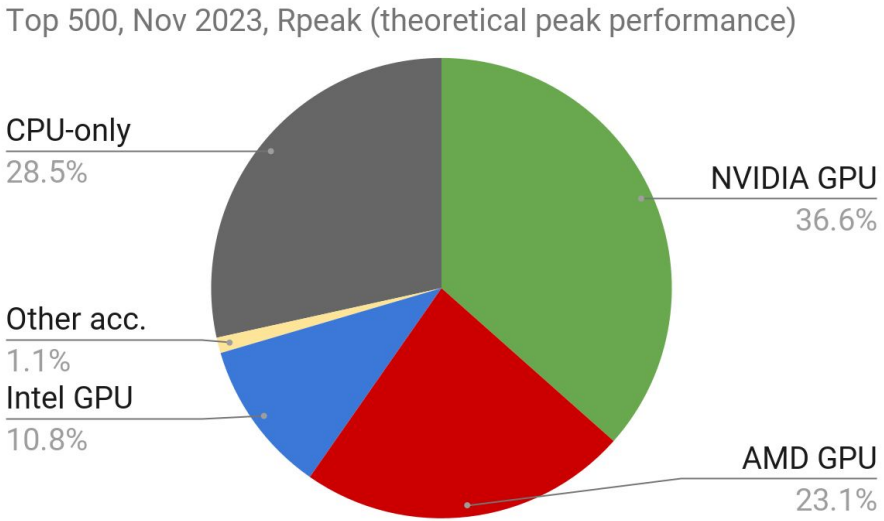


Note: the diagram is applicable to other accelerators; but also different architectures exist

Figure adapted from the Carpentry GPU Programming lesson; GPU photo by @zelebb on Unsplash

A look at the Top-500 list

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,699,904	1,194.00	1,679.82	22,703
2	**Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel DOE/SC/Argonne National Laboratory United States	4,742,808	585.34	1,059.33	24,687
3	**Eagle** - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR, Microsoft Microsoft Azure United States	1,123,200	561.20	846.84	
4	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
5	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,752,704	379.70	531.51	7,107



https://top500.org/statistics/list/

Why GPUs?

- Speed
  - GPU computing can significantly accelerate many types of scientific workloads.
- Energy efficiency
  - Compared to CPUs, GPUs can perform more calculations per watt of power.
- Cost efficiency
  - GPUs can be more cost-effective than traditional CPU-based systems for certain workloads.
- You can’t escape it
  - More and more compute clusters rely of GPUs for most of their performance.

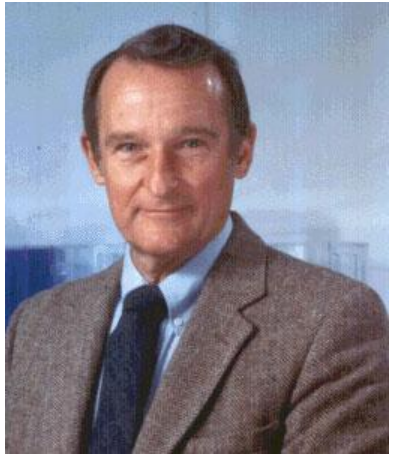
Why **not** GPUs?

- Only for certain workloads
  - Not all workloads can be efficiently parallelized and accelerated on GPUs.
- Steep learning curve
  - This course should help! :)

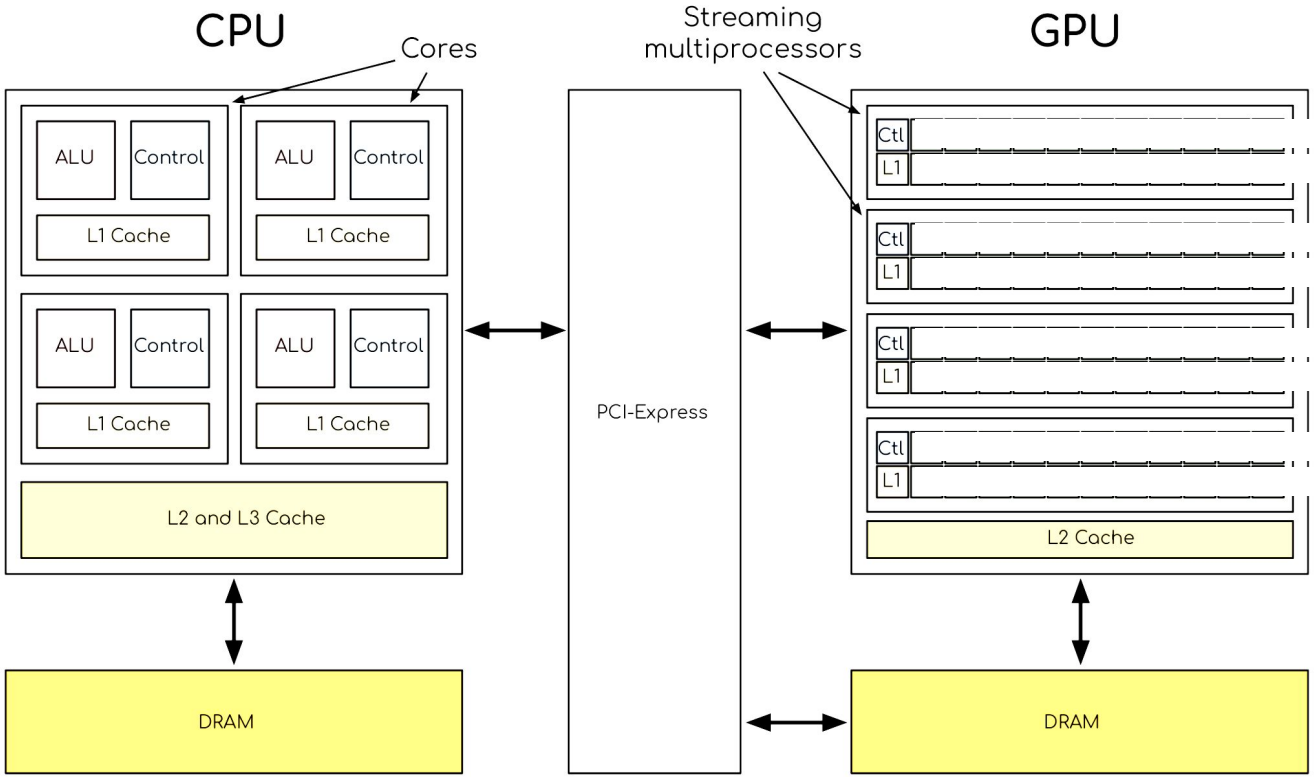
CPU vs GPU

If you were plowing a field, which would you rather use:
two strong oxen or 1024 chickens?

Seymour Cray

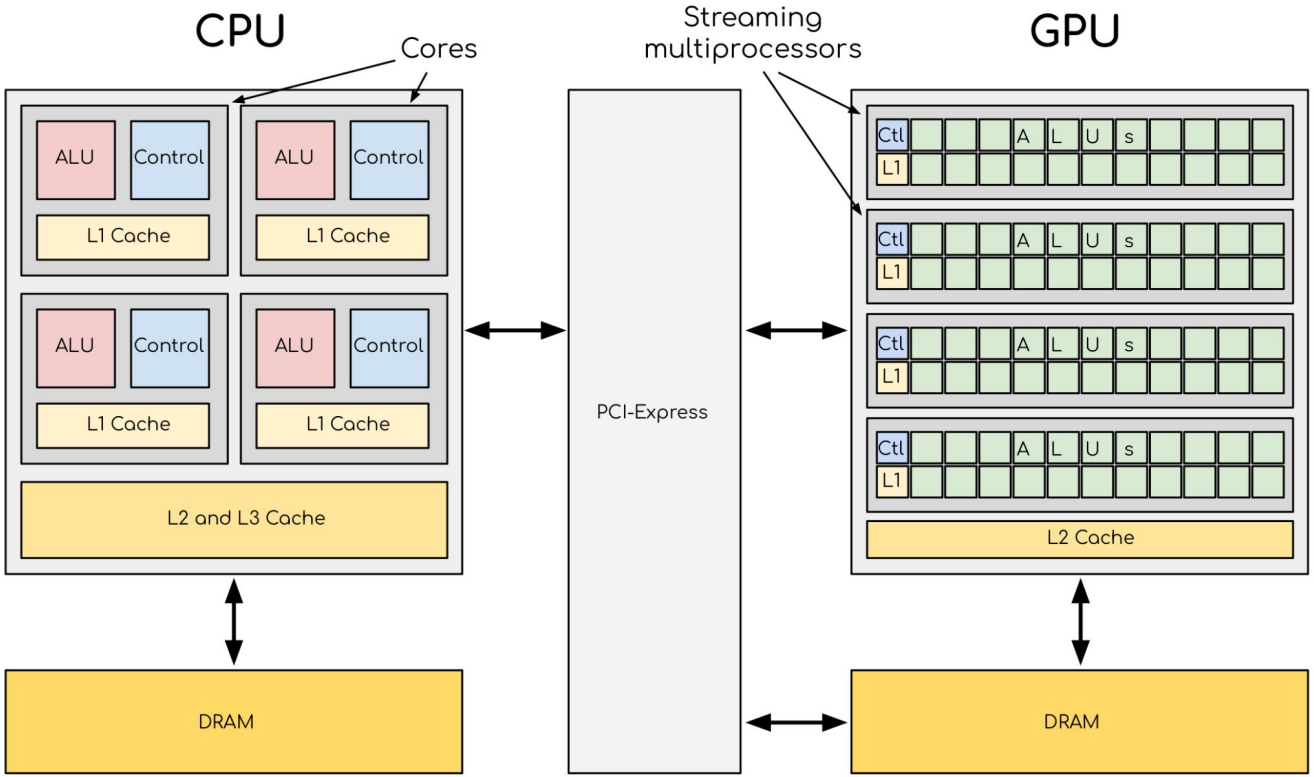


Overview of GPU hardware



Based on https://enccs.github.io/gpu-programming/2-gpu-ecosystem/#overview-of-gpu-hardware

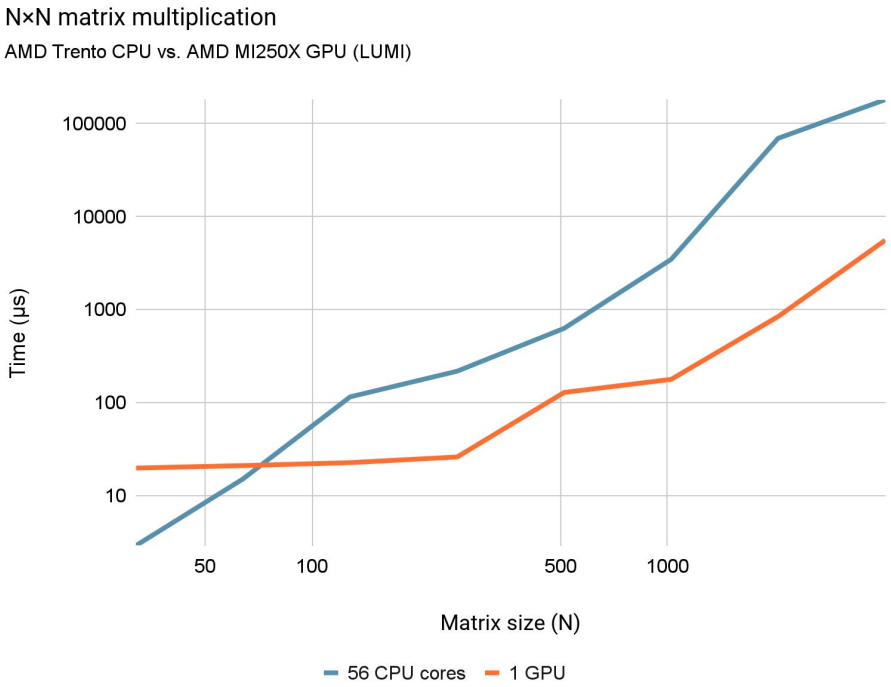
Overview of GPU hardware



CPU vs GPU

CPU	GPU
General purpose	Highly specialized for parallelism
Good for serial processing	Good for parallel processing
Great for task parallelism	Great for data parallelism
Large area dedicated to cache and control	Thousands of floating-point execution units
Low latency per thread	High-throughput

Bandwidth vs. latency




```julia
using AMDGPU
using BenchmarkTools

N = 5:12
for n in N
  A = rand(2^n, 2^n);
  A_d = ROCArray(A);
  @btime $A * $A;
  @btime begin
    $A_d * $A_d;
    AMDGPU.synchronize()
  end
end
```

What are GPUs good for

- Large-scale matrix and vector operations
- Fourier transforms
- Monte Carlo simulations
- Molecular dynamics simulations
- Computational fluid dynamics
- Convolutional neural networks
- Big data analytics: Clustering, classification, regression, etc.
- Graphics rendering

What are GPUs not good for

- Sequential tasks
- Fine-grained branching
- Low arithmetic intensity
- Small data sets
- Limited parallelism
- Memory-bound problems

GPU ecosystem

- Vendor software stacks
  - NVIDIA: CUDA
  - AMD: ROCm/HIP
  - Intel: oneAPI
  - Include low-level (C++) compilers, libraries (BLAS, FFT, etc), profilers
- Portable frameworks
  - OpenCL, SYCL, Kokkos, alpaka, OpenMP, C++ StdPar, …: based on C/C++, Fortran
- High-level frameworks
  - PyCUDA, Numba, AMDGPU.jl, …
- Task-specific frameworks
  - PyTorch, …

GPU systems available to Swedish researchers

- NAISS:
  - LUMI (AMD GPUs, CSC)
  - Dardel (AMD GPUs, KTH)
  - Alvis (NVIDIA GPUs, Chalmers)
  - Tetralith (NVIDIA GPUs, Linköping)
  - Your laptop :)
- EuroHPC:
  - If you need a lot of compute

Quiz time!

Which of the following computational tasks is likely to gain the **least** performance benefit from a GPU?

A. Training a large, deep neural network.
B. Performing a Monte Carlo simulation with a many independent trials.
C. Executing an algorithm with heavy use of recursion and frequent branching.
D. Processing a large image with a convolutional filter.

Quiz time!

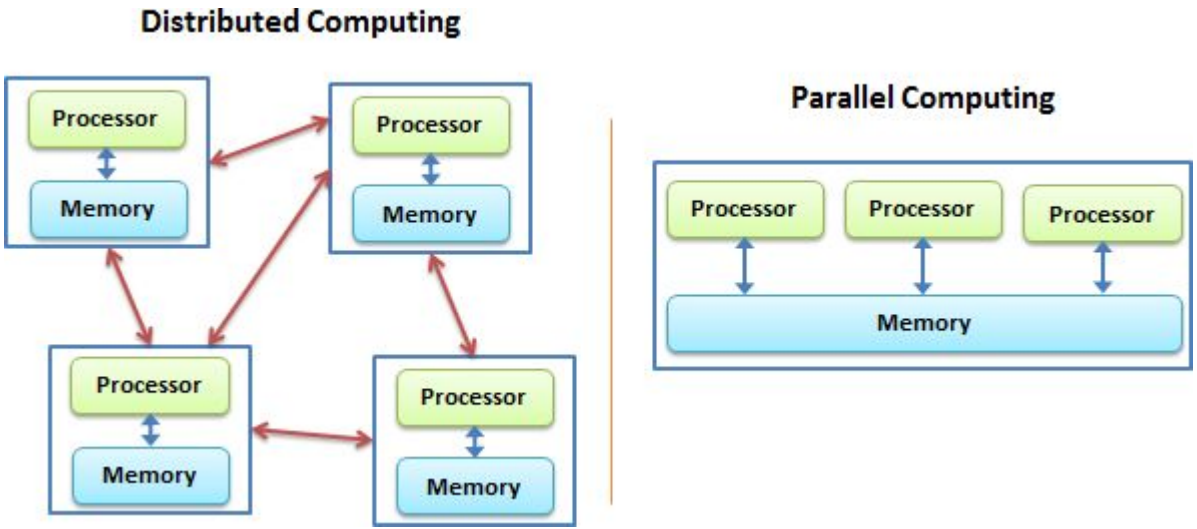
Which of the following computational tasks is likely to gain the **least** performance benefit from a GPU?

A. Training a large, deep neural network.
B. Performing a Monte Carlo simulation with a many independent trials.
C. Executing an algorithm with heavy use of recursion and frequent branching.
D. Processing a large image with a convolutional filter.

GPU Programming Concepts

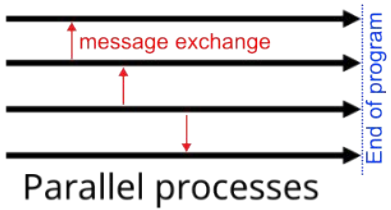
Different types of parallelism

Distributed- vs. Shared-Memory Architecture

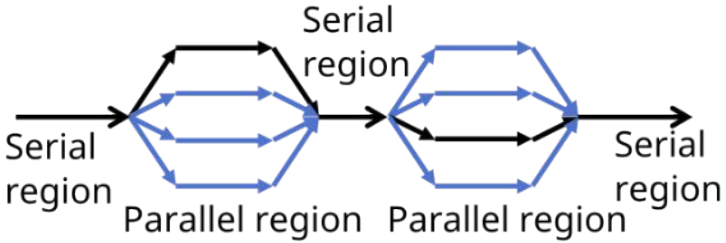


Different types of parallelism

Processes and threads



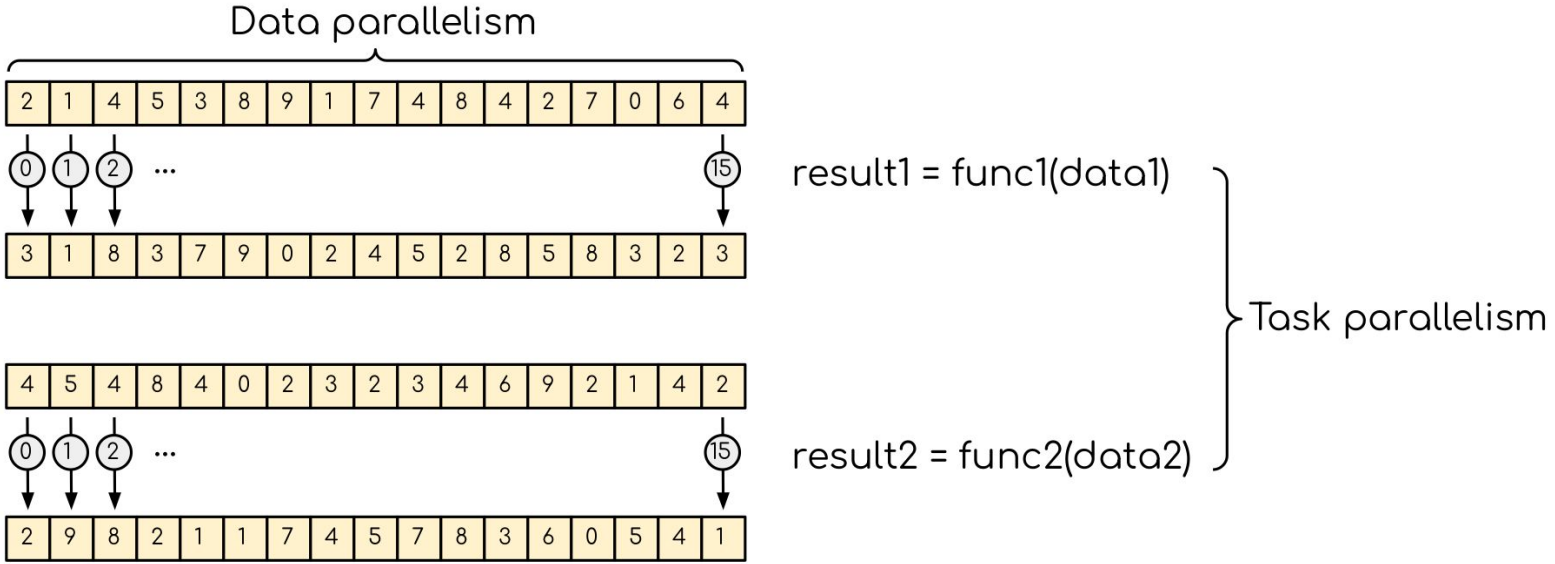
Distributed memory parallelism



Shared memory parallelism

Different types of parallelism

Data- vs. task parallelism



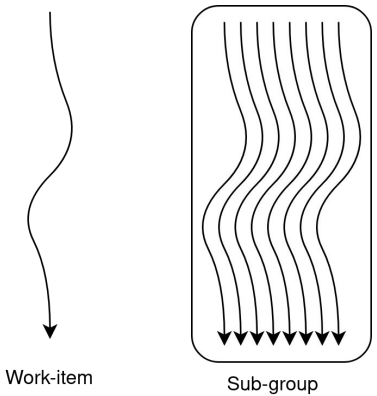
https://enccs.github.io/gpu-programming/4-gpu-concepts/#exposing-parallelism

GPU Concepts

- A **kernel** is executed by tens of thousands of **work-items (threads**)
- **Work-items** run on GPU cores in parallel
- Each **work-item** has an **index** and **private variables**
- Each GPU core executes multiple **work-items** to overlap latencies

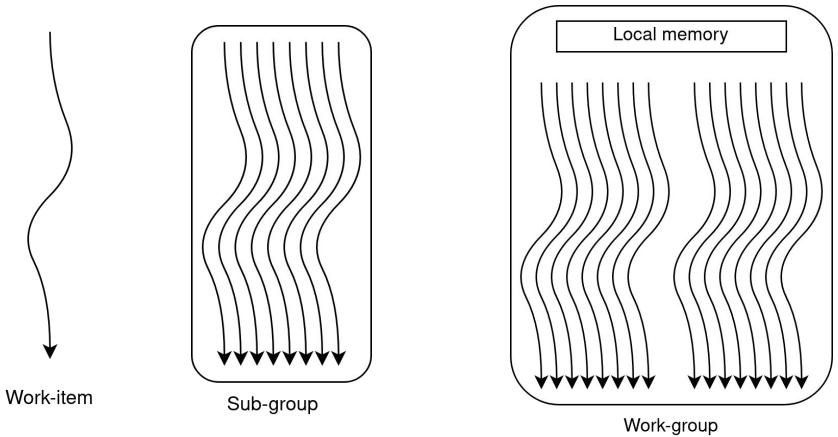
GPU Concepts

- Several “adjacent” **work-items** constitute **sub-group (warp)**
- Size of **sub-group** is a hardware property
- **Work-items** in a **sub-group** are scheduled together and can exchange data
- Adjacent memory accesses are **coalesced**



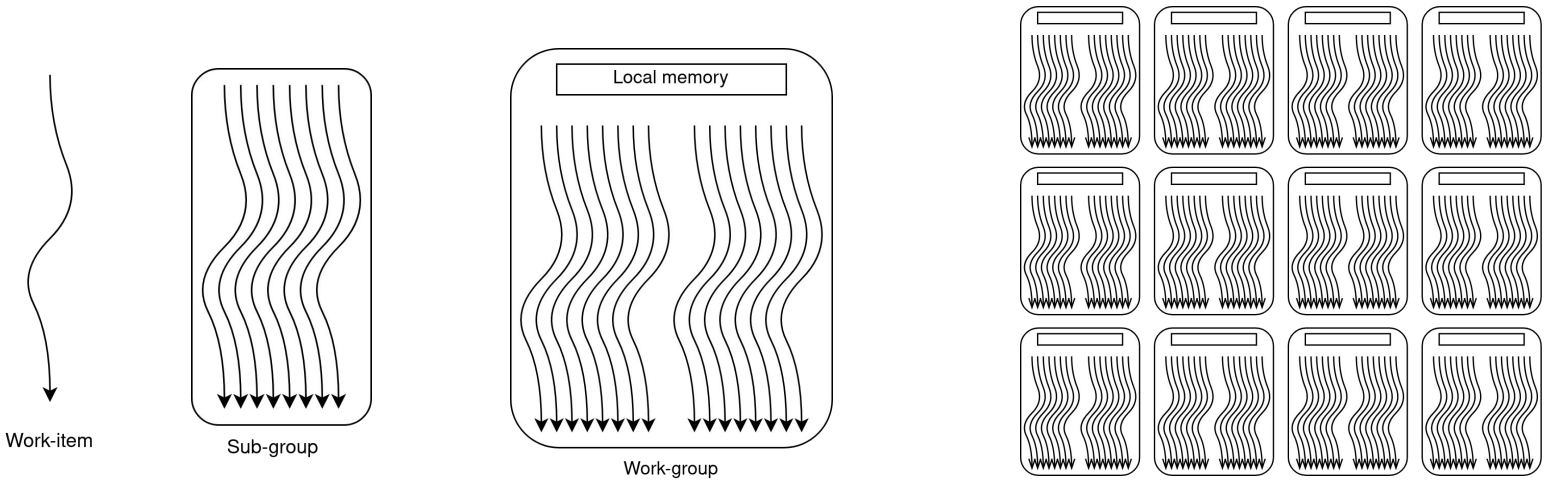
GPU Concepts

- Adjacent **work-items** can be grouped into a **work-group (block)**
- **Work-group** size is determined by the programmer (but limited by hardware)
- **Work-items** in a **work-group** have access to fast **local (shared) memory**
- **Work-items** in a **work-group** can synchronize using **barriers**



GPU Concepts

- All **work-groups** running a **kernel** constitute an **NDRange (grid)**
- **Work-items** in different **work-groups** cannot synchronize
- **Work-items** run the same **kernel**, have different **index**



Dictionary

SYCL	CUDA	HIP
work-item	thread	
sub-group	warp	wavefront
work-group	block	
NDRange	grid	

Quiz time!

What is a kernel in the context of GPU execution?

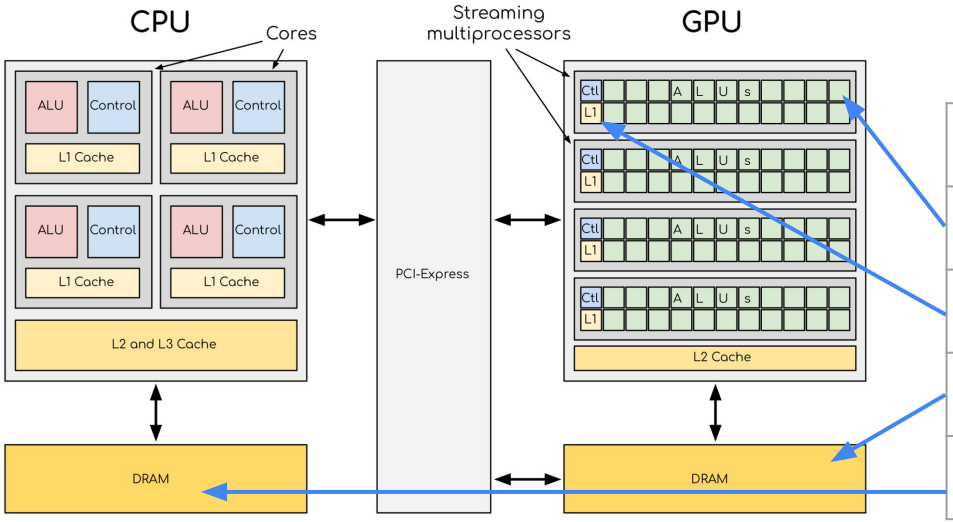
A. A specific section of the CPU used for memory operations.
B. A specific section of the GPU used for memory operations.
C. A type of thread that operates on the GPU.
D. A function that is executed simultaneously by thousands of threads on a GPU.

Quiz time!

What is a kernel in the context of GPU execution?

A. A specific section of the CPU used for memory operations.
B. A specific section of the GPU used for memory operations.
C. A type of thread that operates on the GPU.
D. A function that is executed simultaneously by thousands of threads on a GPU.

GPU memory hierarchy



Memory	Scope	Speed
private	work-item	fast
local	work-group	fast
device	device	slow
host	device and host	very slow

GPU memory hierarchy

SYCL	CUDA	HIP	Scope	Speed
private	registers	registers	work-item	fast
spill	local	scratch	work-item	slow
local	shared	local data store	work-group	fast
device	global	global	device	slow
shared	migrateable	migrateable	device and host	slow
host	host	host	device and host	very slow

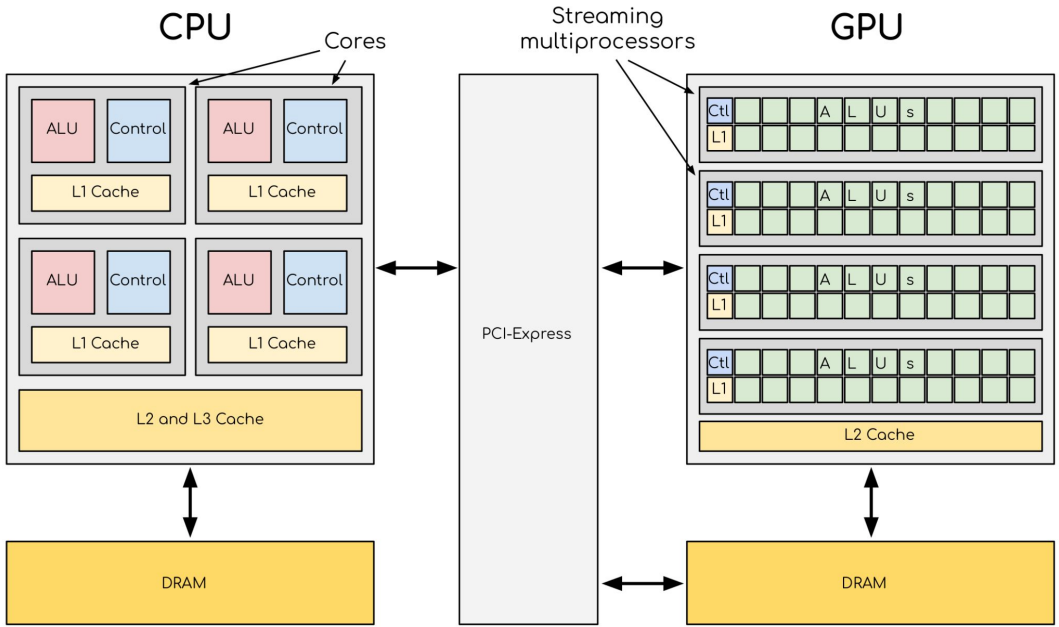
GPU memory hierarchy

SYCL	CUDA	HIP	Scope	Speed
private	registers	registers	work-item	fast
spill	local	scratch	work-item	slow
local	shared	local data store	work-group	fast
device	global	global	device	slow
shared	migrateable	migrateable	device and host	slow
host	host	host	device and host	very slow

Also image/texture and constant…

GPU programming cornerstones

- Get data location right
- Get memory types right
- Minimize divergence
- Oversubscribe
- Know your hardware



GPU Programming models

GPU Programming models

- C++/Fortran:
  - Standard-based solutions
  - Directive-based programming
  - Non-portable kernel-based models
  - Portable kernel-based models
- Higher-level languages
- Exotic languages
- Domain-specific languages

This classification is neither definitive nor exhaustive.

Standard-based models

- Use C++/Fortran to offload algorithms from the standard library to the GPU.
- Despite being "standard", the support for them is extremely limited.
- Limited flexibility and performance.

- Supported on NVIDIA GPUs via NVIDIA HPC SDK (`nvc++`, `nvfortran`).
- Experimental support for all GPUs via AdaptiveCpp.
- Experimental support for AMD GPUs via ROCm StdPar (C++ only).

C++ Standard Parallelism

- Introduced in C++17
- Automatic parallelization of algorithms from C++ standard library

```
#include <algorithm>
#include <execution>
// ...
std::transform(std::execution::par_unseq, A, A + n, B, C,
               [](float x, float y) { return x + y; });
```

Fortran Standard Parallelism

```
subroutine vectoradd(A, B, C, n)
  real :: A(:), B(:), C(:)
  integer :: n, i
  do concurrent (i = 1: n)
    C(i) = A(i) + B(i)
  enddo
end subroutine vectoradd
```

Directive-based programming

- Annotate existing C++/Fortran *serial* code with *hints* to indicate which loops and regions to execute on the GPU.
- Minimal changes to the source code.
- Limited performance and expressiveness.
- OpenACC is more *descriptive*.
  - Developed in 2010 specifically to target accelerators.
  - Supports NVIDIA GPUs with NVIDIA HPC SDK, NVIDIA and AMD GPUs with GCC 12.
- OpenMP is more *prescriptive*.
  - Developed in 1997 for multicore CPUs, accelerator support added later.
  - Supports most GPUs with GCC and Clang.

Directive-based programming: OpenMP

```
// OpenMP in C++
#pragma omp target teams distribute parallel for simd
for (i = 0; i < n; i++) {
  C[i] = A[i] + B[i];
}


! OpenMP in Fortran
!$omp target teams distribute parallel do simd
do i = 1, n
  C(i) = A(i) + B(i)
end do
!$omp end target
```

Directive-based programming: OpenACC

```cpp
// OpenACC in C++
#pragma acc parallel loop
for (i = 0; i < n; i++) {
  C[i] = A[i] + B[i];
}
```

```fortran
! OpenACC in Fortran
!$acc parallel loop
do i = 1, n
  C(i) = A(i) + B(i)
end do
!$acc end parallel loop
```

Kernel-based programming

- Requires *deep knowledge* of GPU architecture.
- The code to execute on the GPU is outlined into a separate function (kernel).
- Low-level code (conceptually), more control, higher performance (or not).

Non-portable models

- CUDA
- HIP

Portable models

- OpenCL
- SYCL
- Kokkos

Kernel-based programming workflow

1. Select/initialize the device
2. Allocate memory on the device
3. Copy input data from host to device
4. Launch kernel on the device
5. Copy output from device to host
6. Free allocated device memory

Don’t forget to check for errors!

Non-portable models

- CUDA
  - First mainstream GPU computing framework, widely used and supported.
  - Developed by NVIDIA, works only on NVIDIA GPUs.
  - Extensive ecosystem, libraries, tutorial.
  - Stable, feature-rich, well-supported.
- HIP
  - Developed by AMD, works on AMD and NVIDIA* GPUs.
  - Very similar to CUDA, automatic conversion tools exist.
  - Less mature, no official support on most consumer GPUs.

Non-portable models: CUDA

```
__global__ void vector_add(float *A, float *B, float *C, int n) {
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  if (tid < n) {
    C[tid] = A[tid] + B[tid];
  }
}

// Allocate GPU memory
cudaMalloc((void**)&Ad, n * sizeof(float)); // ...
// Copy the data from the CPU to the GPU
cudaMemcpy(Ad, Ah, sizeof(float) * n, cudaMemcpyHostToDevice); // ...
// Define grid dimensions: how many threads to launch
dim3 blockDim{256, 1, 1};
dim3 gridDim{(n/256)*256 + 1, 1, 1};
vector_add<<<gridDim, blockDim>>>(Ad, Bd, Cd, n);
```

Non-portable models: HIP

```
__global__ void vector_add(float *A, float *B, float *C, int n) {
  int tid = threadIdx.x + blockIdx.x * blockDim.x;
  if (tid < n) {
    C[tid] = A[tid] + B[tid];
  }
}

// Allocate GPU memory
hipMalloc((void**)&Ad, n * sizeof(float)); // ...
// Copy the data from the CPU to the GPU
hipMemcpy(Ad, Ah, sizeof(float) * n, hipMemcpyHostToDevice); // ...
// Define grid dimensions: how many threads to launch
dim3 blockDim{256, 1, 1};
dim3 gridDim{(n/256)*256 + 1, 1, 1};
vector_add<<<gridDim, blockDim>>>(Ad, Bd, Cd, n);
```

Portable models: OpenCL

- Cross-platform open standard for accelerator programming, based on C.
- Supports CPUs, GPUs, FPGAs, embedded devices.
- Supported by all major vendors, but to a varying degree. No latest features.
- Good performance requires vendor extensions, defeating portability aspect.
- Separate-source model, no compiler support required
    - just headers and a runtime library.

Portable models: OpenCL

Kernel:

```
static const char* kernel_source = R"(
__kernel void vector_add(
          __global float *A, __global float *B, __global float *C, int n) {
  int tid = get_global_id(0);
  if (tid < n) {
    C[tid] = A[tid] + B[tid];
  }
}
)";
```

Portable models: OpenCL

Launch code:

```
// Boilerplate...
cl_platform_id platform;
clGetPlatformIDs(1, &platform, NULL);
cl_device_id device;
clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);
cl_context context = clCreateContext(NULL, 1, &device, NULL, NULL, NULL);
cl_command_queue queue = clCreateCommandQueue(context, device, 0, NULL);
cl_program program = clCreateProgramWithSource(context, 1, &kernel_source, NULL, NULL);
clBuildProgram(program, 1, &device, NULL, NULL, NULL);
cl_kernel kernel = clCreateKernel(program, "vector_add", NULL);
// Allocate the arrays on GPU
cl_mem Ad = clCreateBuffer(context, CL_MEM_READ_ONLY, n * sizeof(float), NULL, NULL); // ...
// Copy the data from CPU to GPU
clEnqueueWriteBuffer(queue, Ad, CL_TRUE, 0, n * sizeof(float), Ah, 0, NULL, NULL); // ..
// Set arguments and launch the kernel
clSetKernelArg(kernel, 0, sizeof(cl_mem), &Ad); // ...
cl_int n_as_cl_int = n;
clSetKernelArg(kernel, 3, sizeof(cl_int), &n_as_cl_int);
size_t globalSize = n;
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &globalSize, NULL, 0, NULL, NULL);
```

Portable models: SYCL

- Cross-platform open standard for accelerator programming
- Single-source model based on C++17
  - requires compiler support and C++ knowledge
- Two major implementations: Intel oneAPI DPC++ and AdaptiveCpp
- Supported officially only by Intel, works via CUDA/HIP for NVIDIA/AMD
- Interoperability with native features (at the expense of portability)

Portable models: SYCL

```
// Boilerplate
sycl::queue queue{{sycl::property::queue::in_order()}};
// Allocate GPU memory
float* Ad = sycl::malloc_device<float>(n, queue); // ...
// Copy the data from CPU to GPU
queue.copy<float>(Ah, Ad, n); // ...
// Submit a kernel into a queue; cgh is a helper object
queue.submit([&](sycl::handler &cgh) {
  cgh.parallel_for<class Kernel>(sycl::range<1>{n}, [=](sycl::id<1> i) {
    Cd[i] = Ad[i] + Bd[i];
  });
});
```

Portable models: SYCL

```
// Boilerplate
sycl::queue queue{{sycl::property::queue::in_order()}};
// Allocate GPU memory
float* Ad = sycl::malloc_device<float>(n, queue); // ...
// Copy the data from CPU to GPU
queue.copy<float>(Ah, Ad, n); // ...
// Submit a kernel into a queue; cgh is a helper object
queue.submit([&](sycl::handler &cgh) {
  cgh.parallel_for<class Kernel>(sycl::range<1>{n}, [=](sycl::id<1> i) {
    Cd[i] = Ad[i] + Bd[i];
  });
});
```

Portable models: SYCL

- Open-source programming model for heterogeneous parallel computing
- Mainly developed at Sandia National Laboratories.
- Single source model, similar to SYCL (but earlier).
- No official vendor support, implemented on top of other frameworks.
- Interoperability with native features (at the expense of portability).

High-level languages: Julia

Julia has first-class support for GPU programming through the following packages that target GPUs from all three major vendors:

- CUDA.jl for NVIDIA GPUs,
- AMDGPU.jl for AMD GPUs,
- oneAPI.jl for Intel GPUs,
- Metal.jl for Apple M-series GPUs.

```
using AMDGPU

A_d = ROCArray(A); # …
C_d = $A_d + $B_d;
AMDGPU.synchronize()
```

High-level languages: Julia

```
using CUDA
function vector_add!(A, B, C)
  i = workitemIdx().x + (workgroupIdx().x - 1) * workgroupDim().x
  if i <= length(A)
    @inbounds C[i] = A[i] + B[i]
  end
  return
end

Ad = CuArray(A); # ...

nthreads = 256
numblocks = cld(length(Ad), nthreads)

@cuda threads=nthreads blocks=numblocks vector_add!(A, B, C)
```

High-level languages: Python

- CuPy: NVIDIA-only.
- cuDF: NVIDIA-only.
- PyCUDA: ... you guessed it.
- Numba: NVIDIA and deprecated AMD support.

High-level languages: Python (Numba)

```
import math
import numba
import numpy as np

@numba.vectorize([numba.float32(numba.float32, numba.float32)], target='cuda')
def vector_add_gpu(a, b):
    return a + b

a, b = np.array(..., dtype=np.float32)
c = vector_add_gpu(a, b)
```

Exotic GPU-specific languages

- Chapel: developed by Cray/HPE for parallel scientific programming, early AMD/NVIDIA support.

```
on here.gpus[0] {
  var A, B, C : [0..#n] real(32); // Allocate
  A = Ah; // Copy from CPU to GPU, ...
  forall i in 0..#n { C[i] = A[i] + B[i]; };
}
```

- Futhark: functional data-parallel language for GPU programming with OpenCL and CUDA backends.

```
def vecadd A B =
  map (\\(x,y) -> x + y) (zip A B)
```

Opinionated summary

- Standard-based: if you already use STL algorithms and have supported compiler
- Directive-based: if you have serial code and no time
- CUDA or HIP: if you are targeting specific vendor and want best performance
- OpenCL: if you want portability on any hardware
- Kokkos or SYCL: if you want performance and portability on modern GPUs
- Higher-level languages: if you hate C, C++, and Fortran
- Exotic languages: if you are feeling adventurous
- Libraries: when it’s already done