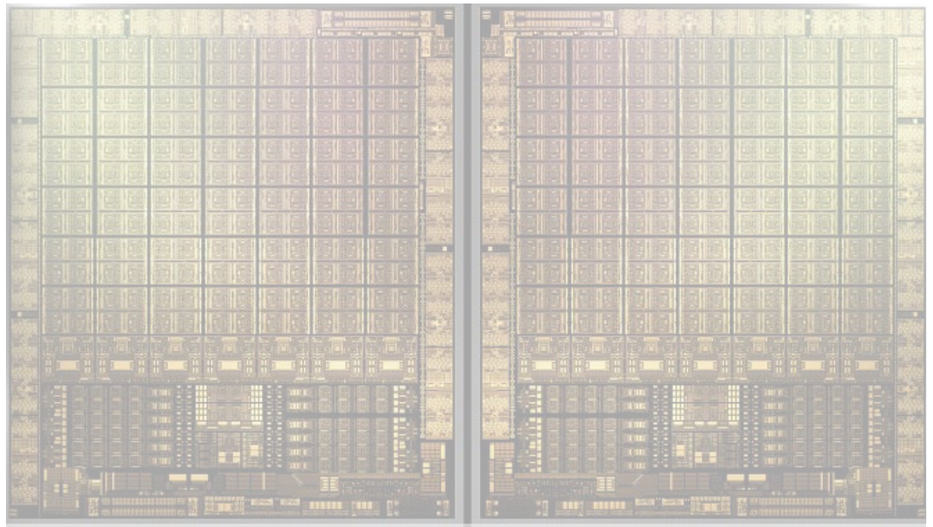
GPU programming review

A basic introduction

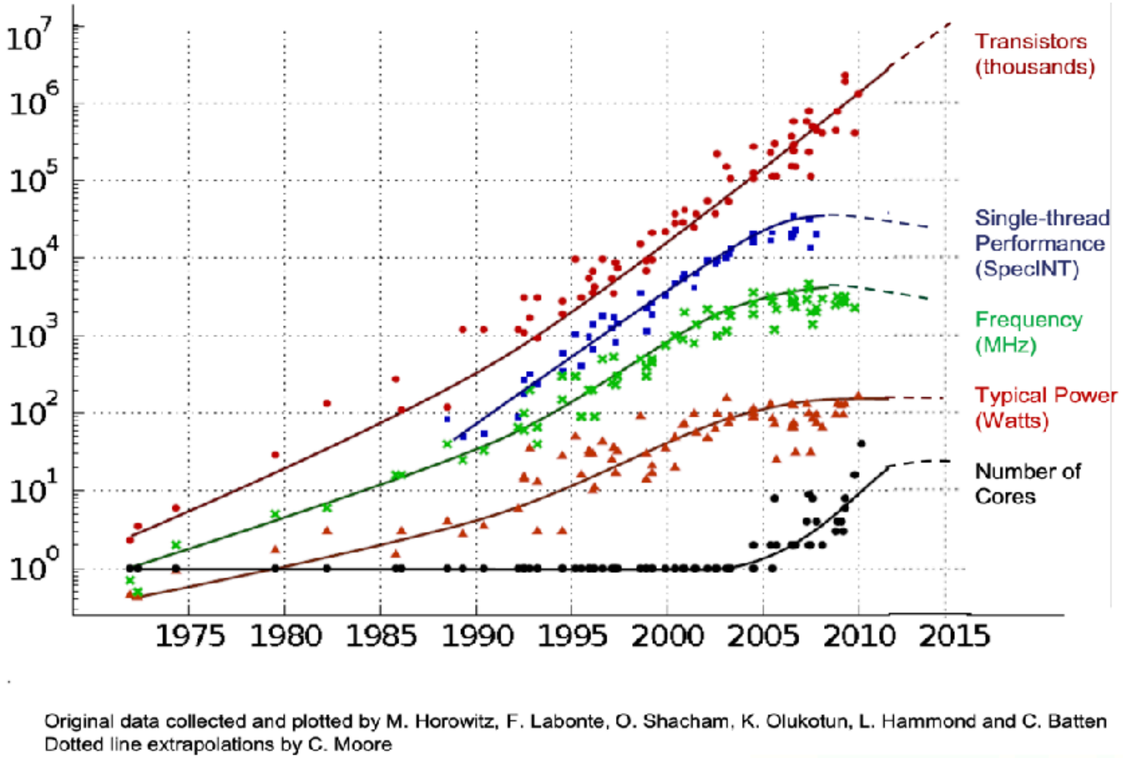


Juan Fernandez

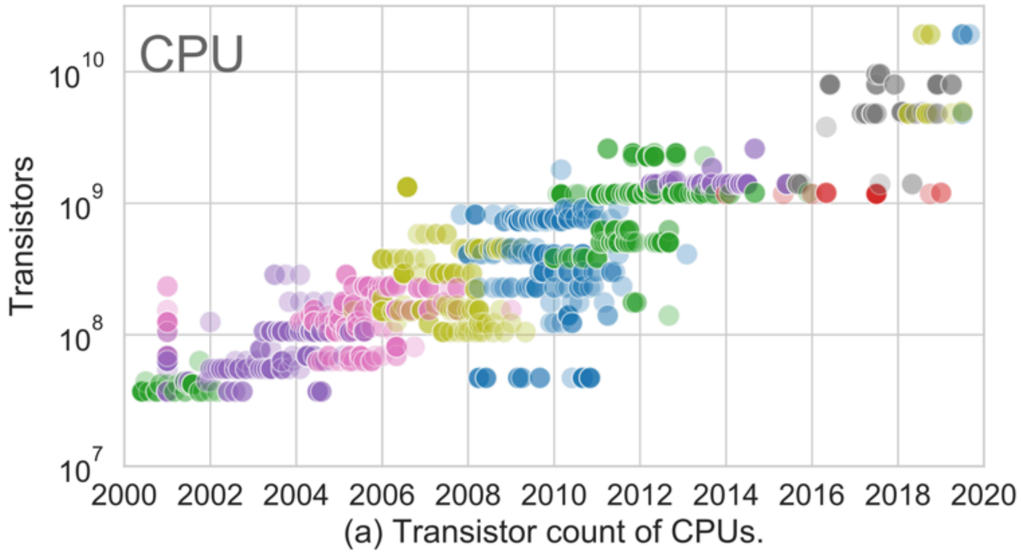
October 31, 2022 @ ETH Zürich

Why use GPUs for general purpose computing?

Let's look at the historical trends of hardware technology



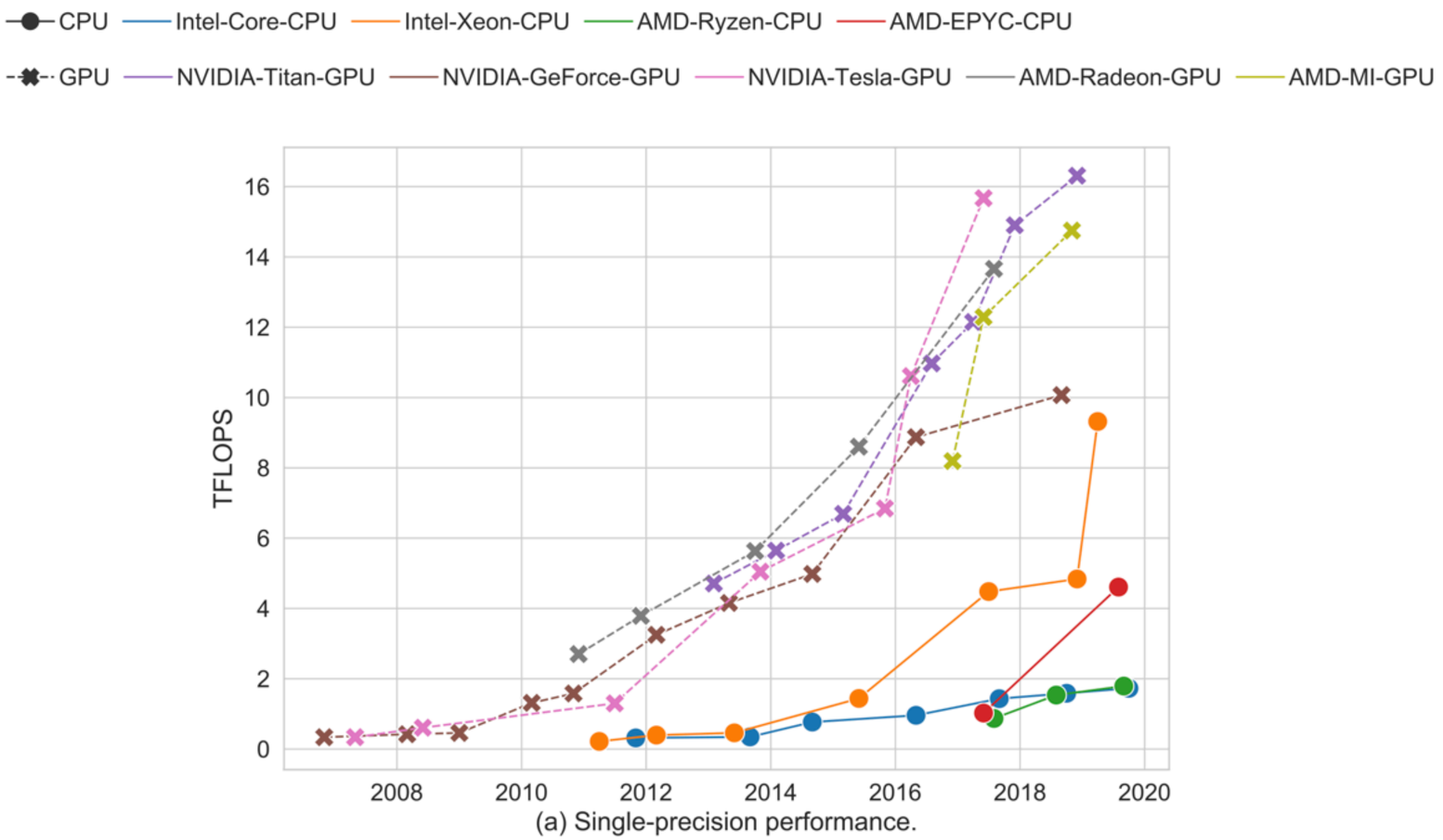
Source: http://www.sci.utah.edu/~mb/Teaching/Week3/SC12_Harrod.pdf



Source: https://arxiv.org/pdf/1911.11313v1.pdf

Why use GPUs for general purpose computing?

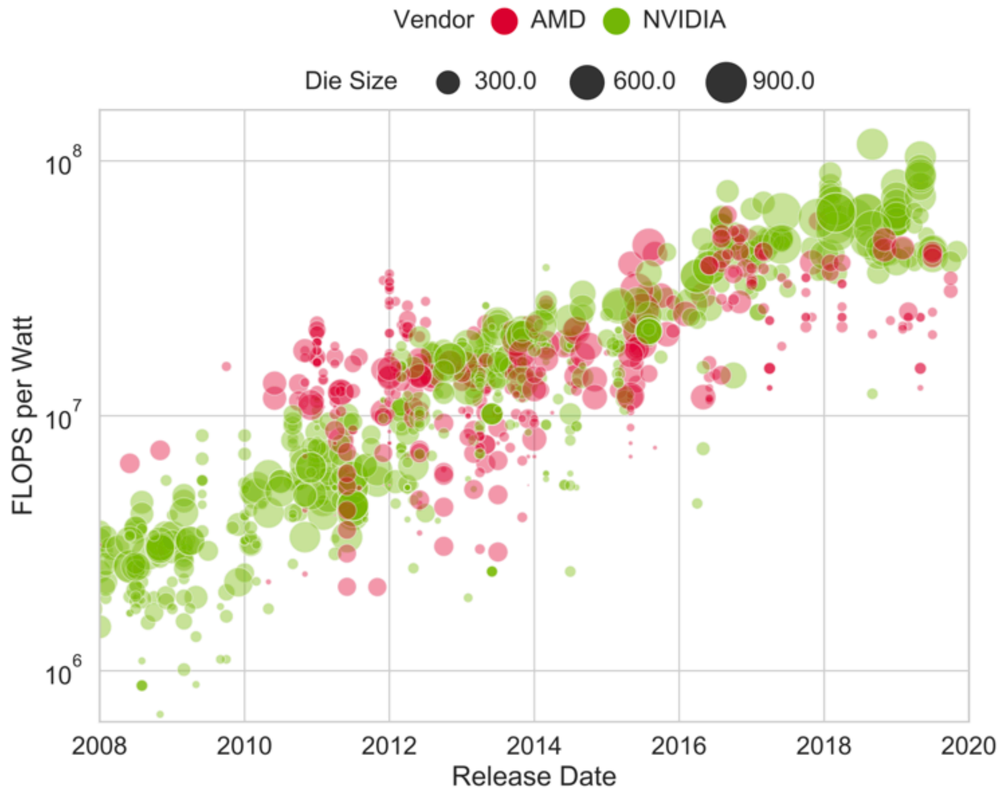
Let’s look at the historical trends of hardware technology



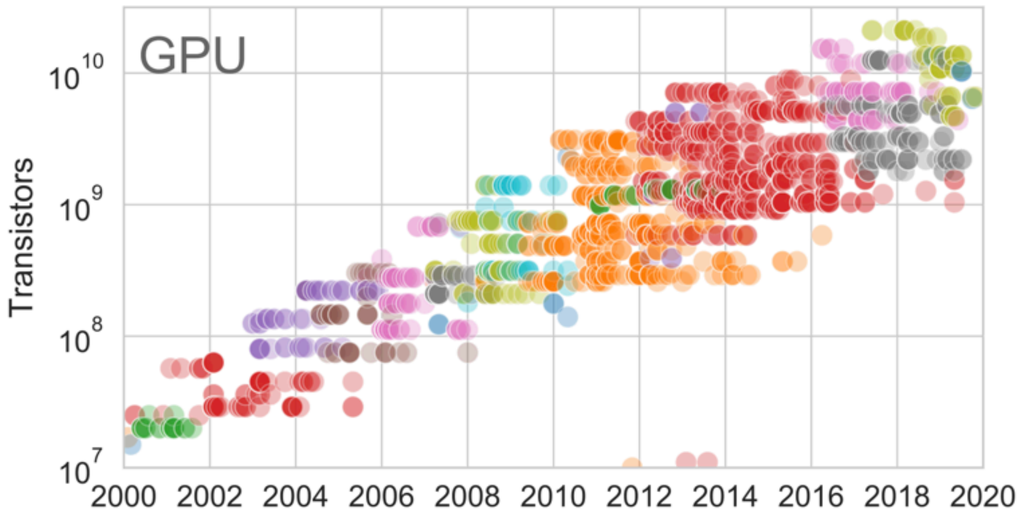
Source: https://arxiv.org/pdf/1911.11313v1.pdf

Why use GPUs for general purpose computing?

Let’s look at the historical trends of hardware technology



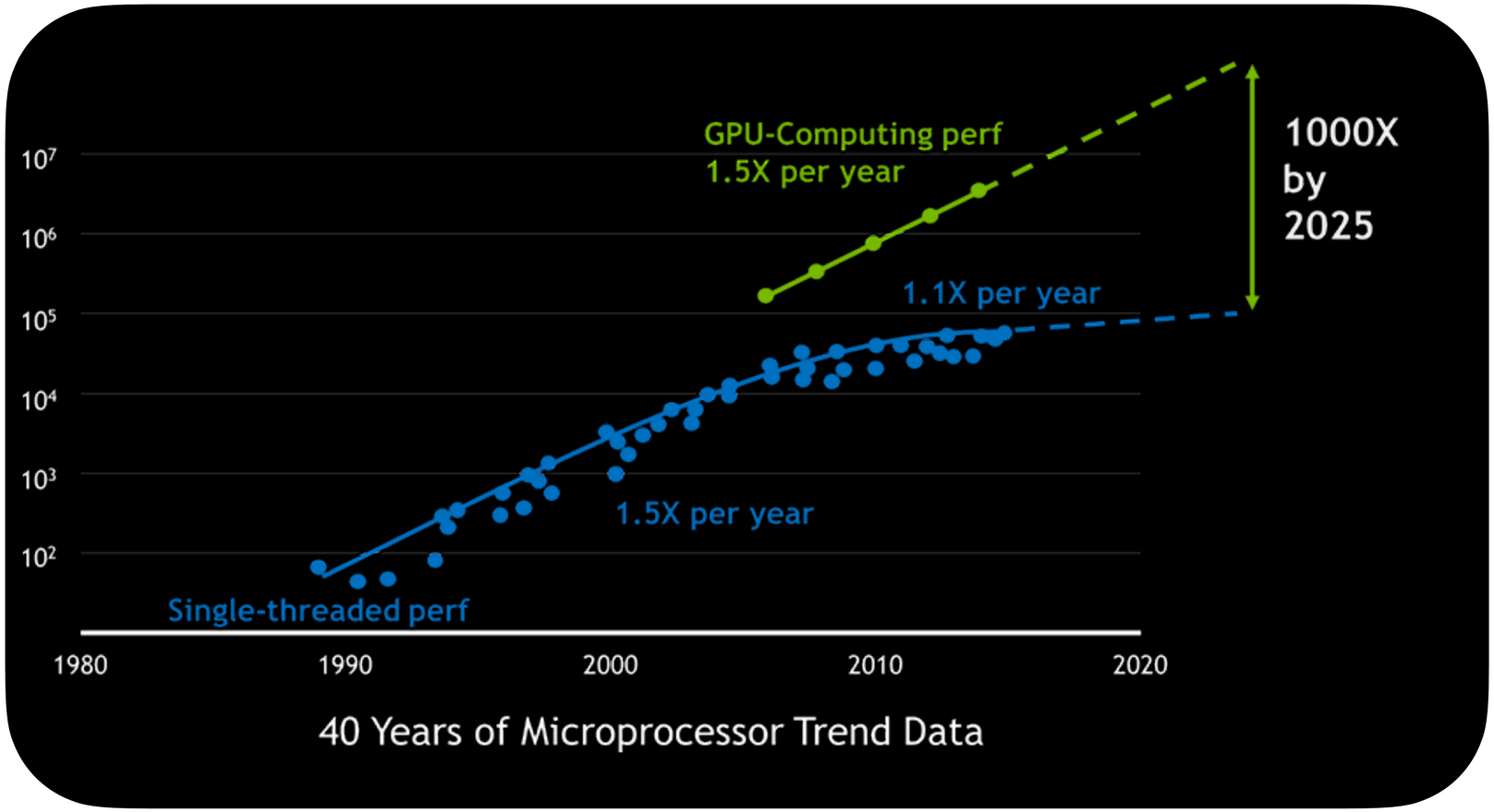
a) FLOPS per Watt historical trend for two major GPU manufacturers



b) Transistor count of GPUs

Source: https://arxiv.org/pdf/1911.11313v1.pdf

NVIDIA's (and everybody's) wish:



Source: https://www.nvidia.com/en-gb/about-nvidia/ai-computing/

How to achieve this acceleration?

- Libraries
  - cuBLAS, cuFFT, CUDA Math Library, cuPy, etc.
- Directive-based programming model
  - OpenMP, OpenACC, etc.
- Languages*
  - CUDA, OpenCL, ROCm, oneAPI, SYCL, etc.

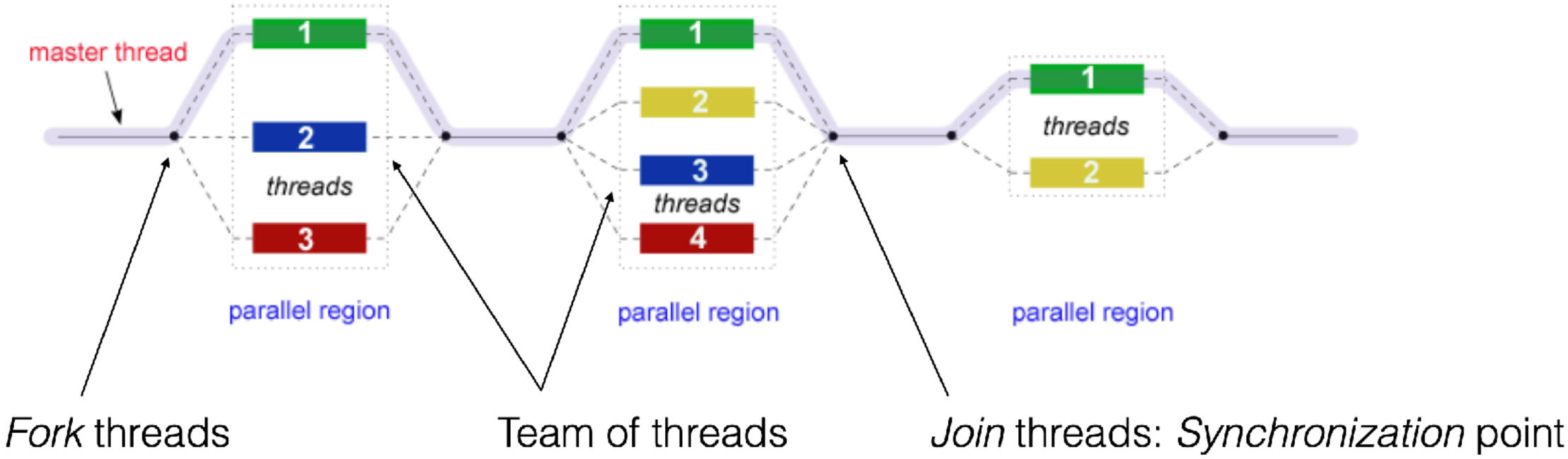
How to achieve this acceleration?

- Libraries
  - cuBLAS, cuFFT, CUDA Math Library, cuPy, etc.
- Directive-based programming model
  - OpenMP, OpenACC, etc.
- Languages*
  - CUDA, OpenCL, ROCm, oneAPI, SYCL, etc.

How to achieve this acceleration?

Application Programming Interfaces

OpenMP
Threads



Source: https://gitlab.ethz.ch/hpcsel_hs21/lecture

OpenMP

Hello World for a CPU

```
1 #include <omp.h>
2 #define NTHREADS 8
3
4 void do_work(const size_t tid)
5 {
6     // more local data
7     // do work
8 }
9
10 int main(int argc, char *argv[])
11 {
12 #pragma omp parallel num_threads(NTHREADS) // spawn threads
13     {
14         // need omp.h header for this
15         const size_t t = omp_get_thread_num();
16         do_work(t);
17     } // join threads (implicit barrier)
18
19     return 0;
20 }
```

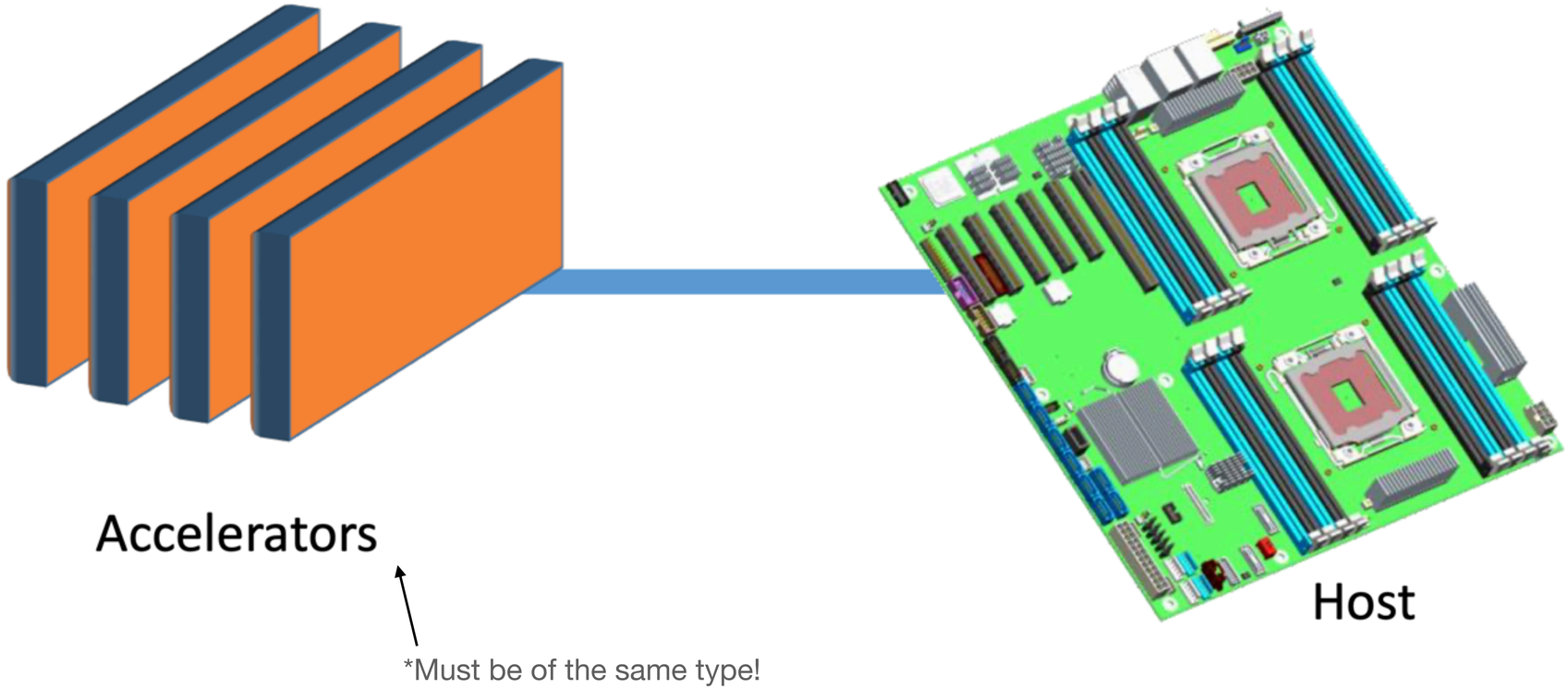
Even less code

Distinction between sequential code and parallel regions is very clear

Implicit barrier at end of structured block (synchronization point)

Source: https://gitlab.ethz.ch/hpcsel_hs21/lecture

OpenMP
Device Model



Source: https://www.openmp.org/wp-content/uploads/2021-10-20-Webinar-OpenMP-Offload-Programming-Introduction.pdf

OpenMP

Sample code to accelerate with a GPU

```
void saxpy() {
    float a, x[SZ], y[SZ];
    // left out initialization
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
#pragma omp parallel for firstprivate(a)
    for (int i = 0; i < SZ; i++) {
        y[i] = a * x[i] + y[i];
    }
    te = omp_get_wtime();
    t = te - tb;
    printf("Time of kernel: %lf\n", t);
}
```

Timing code (not needed, just to have a bit more code to show ☺)

This is the code we want to execute on a target device (i.e., GPU)

Timing code (not needed, just to have a bit more code to show ☺)

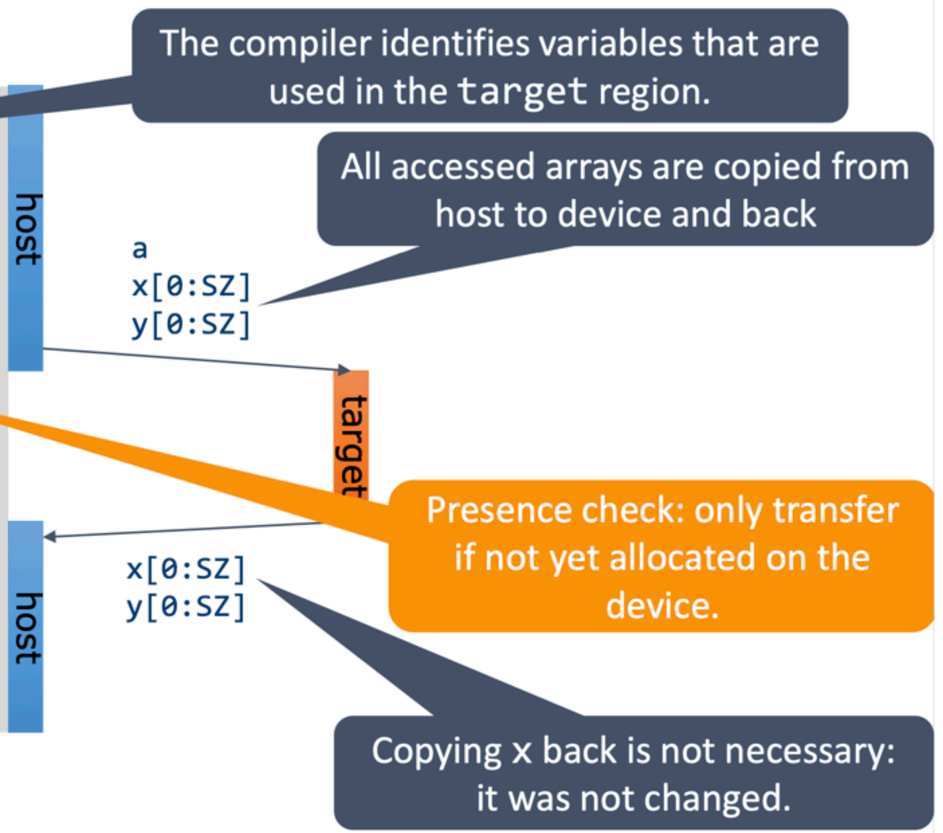
Don’t do this at home!
Use a BLAS library for this!

Source: https://www.openmp.org/wp-content/uploads/2021-10-20-Webinar-OpenMP-Offload-Programming-Introduction.pdf

OpenMP

Target device = GPU

```
void saxpy() {
    float a, x[SZ], y[SZ];
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
#pragma omp target "map(tofrom:y[0:SZ])"
    for (int i = 0; i < SZ; i++) {
        y[i] = a * x[i] + y[i];
    }
    te = omp_get_wtime();
    t = te - tb;
    printf("Time of kernel: %lf\n", t);
}
```



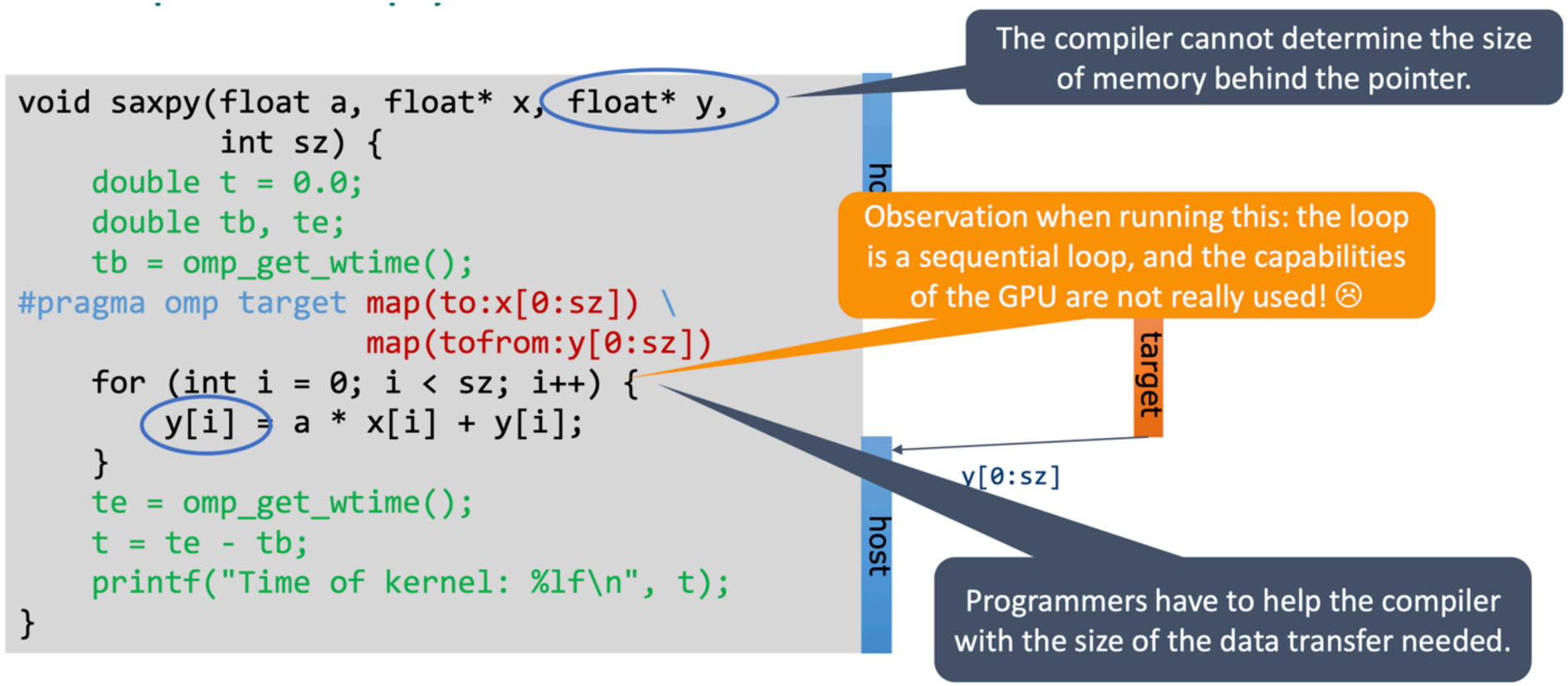
clang/LLVM: `clang -fopenmp -fopenmp-targets=<target triple>`
GNU: `gcc -fopenmp`
AMD ROCm: `clang -fopenmp -offload-arch=gfx908`
NVIDIA: `nvcc –mp=gpu -gpu=cc80`

Source: https://www.openmp.org/wp-content/uploads/2021-10-20-Webinar-OpenMP-Offload-Programming-Introduction.pdf

OpenMP

Another example (naive mistake)

```
void saxpy(float a, float* x, float* y,
           int sz) {
    double t = 0.0;
    double tb, te;
    tb = omp_get_wtime();
#pragma omp target map(to:x[0:sz]) \
                   map(tofrom:y[0:sz])
    for (int i = 0; i < sz; i++) {
        y[i] = a * x[i] + y[i];
    }
    te = omp_get_wtime();
    t = te - tb;
    printf("Time of kernel: %lf\n", t);
}
```



Source: https://www.openmp.org/wp-content/uploads/2021-10-20-Webinar-OpenMP-Offload-Programming-Introduction.pdf

OpenMP

Multi-level parallel devices

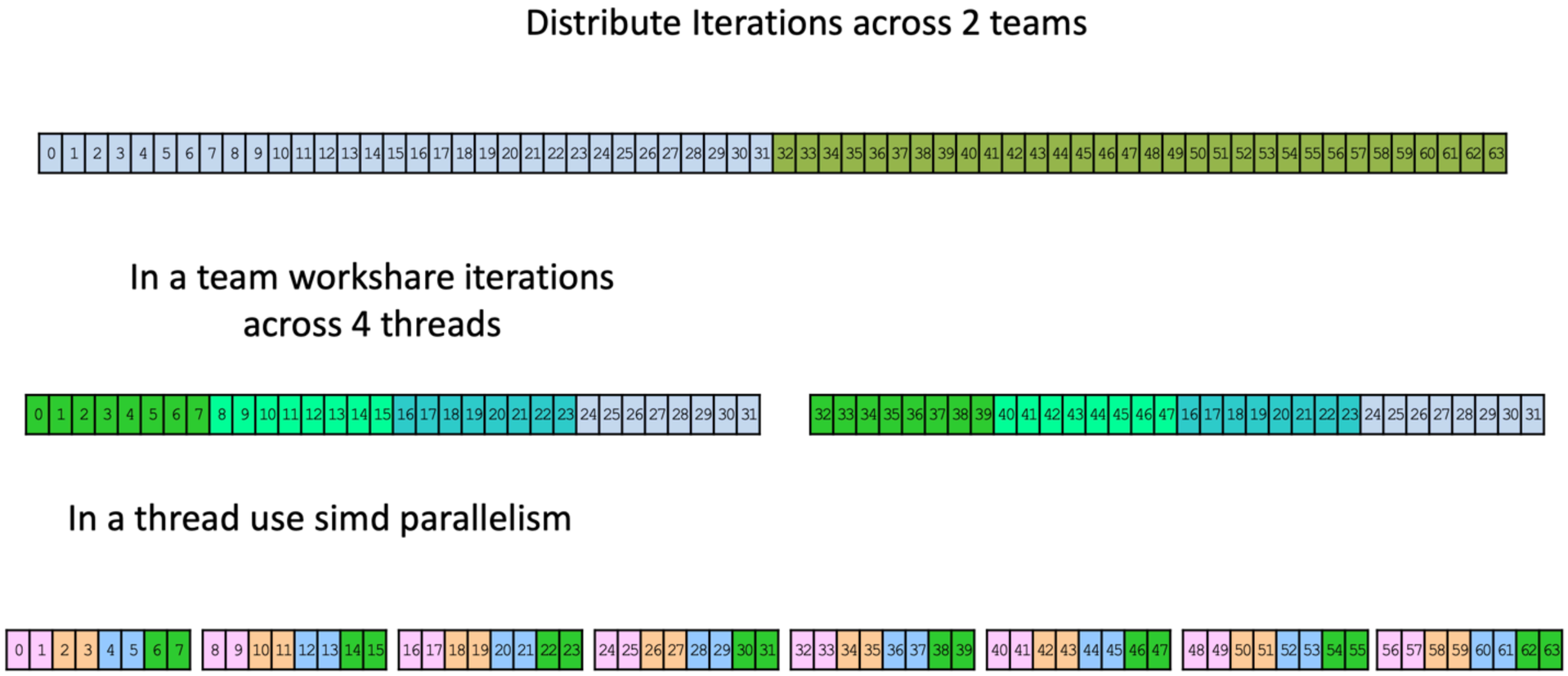
- Teams
  - Threads
    - SIMD units

```
void saxpy(float a, float* x, float* y, int sz) {
    #pragma omp target teams map(to:x[0:sz]) map(tofrom:y[0:sz]) num_teams(nteams)
    {
        int bs = n / omp_get_num_teams();   // n assumed to be multiple of #teams
        #pragma omp distribute
        for (int i = 0; i < sz; i += bs) {
            #pragma omp parallel for simd firstprivate(i,bs)
            for (int ii = i; ii < i + bs; ii++) {
                y[ii] = a * x[ii] + y[ii];
}   }   }   }
```

Source: https://www.openmp.org/wp-content/uploads/2021-10-20-Webinar-OpenMP-Offload-Programming-Introduction.pdf

OpenMP

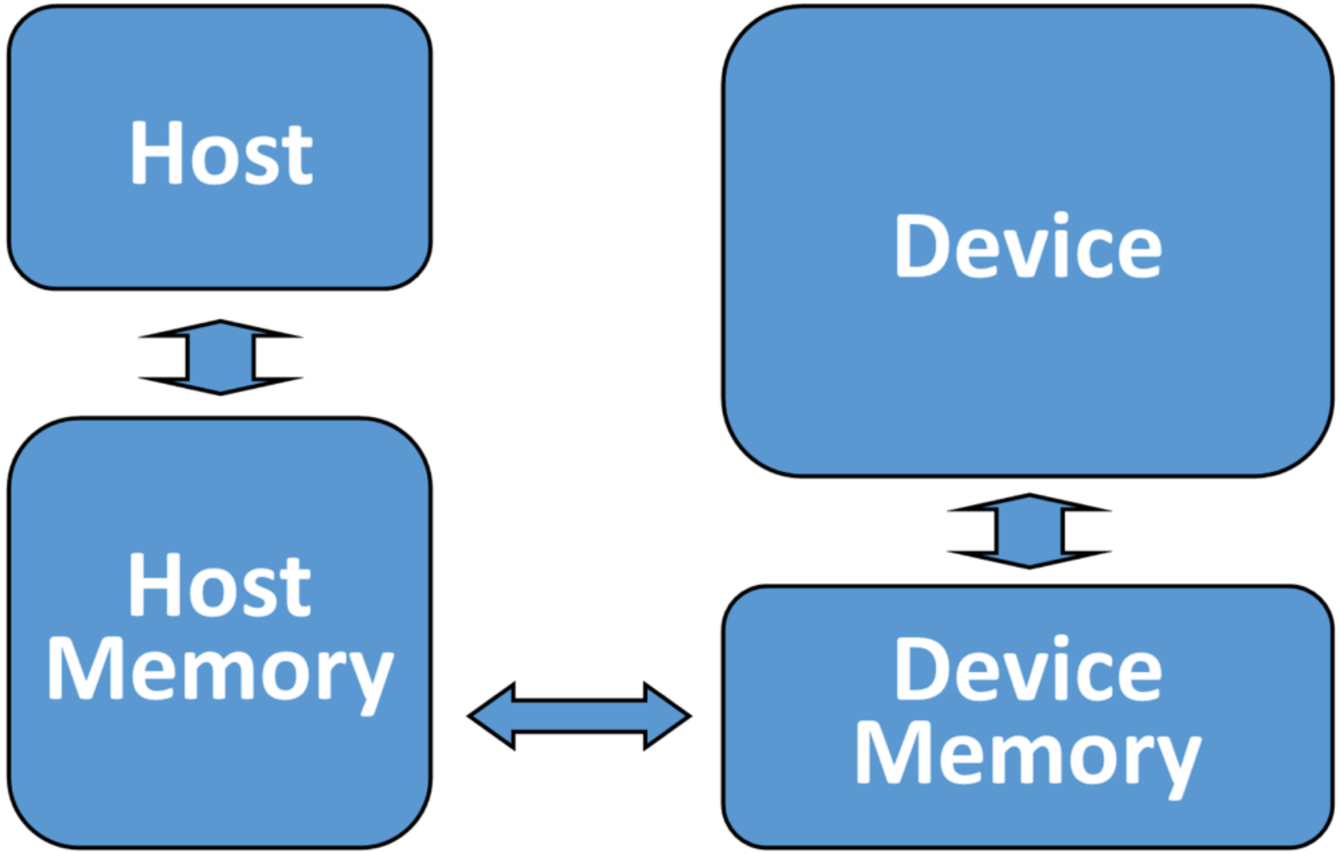
Multi-level parallel devices



Source: https://openmpcon.org/wp-content/uploads/openmpcon2017/Tutorial2-Advanced_OpenMP.pdf

OpenACC

Device Model



Source: https://www.openacc.org/sites/default/files/inline-files/openacc-guide.pdf

OpenACC

#pragma acc <kernels> and <parallel>

`kernels`

Compiler detects parallelization opportunities of a loop to create a parallel kernel.

`parallel`

Equivalent concept to OpenMP's `parallel` construct.

Source: https://www.openacc.org/sites/default/files/inline-files/openacc-guide.pdf

OpenACC

Vector addition example

kernels here

function call here

```
#include <stdio.h>
#include <stdlib.h>
void vecaddgpu( float *restrict r, float *a, float *b, int n ){
    #pragma acc kernels loop copyin(a[0:n],b[0:n]) copyout(r[0:n])
    for( int i = 0; i < n; ++i ) r[i] = a[i] + b[i];
}

int main( int argc, char* argv[] ){
    int n; /* vector length */
    float * a; /* input vector 1 */
    float * b; /* input vector 2 */
    float * r; /* output vector */
    float * e; /* expected output values */
    int i, errs;
    if( argc > 1 ) n = atoi( argv[1] );
    else n = 100000; /* default vector length */
    if( n <= 0 ) n = 100000;
    a = (float*)malloc( n*sizeof(float) );
    b = (float*)malloc( n*sizeof(float) );
    r = (float*)malloc( n*sizeof(float) );
    e = (float*)malloc( n*sizeof(float) );
    for( i = 0; i < n; ++i ){
         a[i] = (float)(i+1);
         b[i] = (float)(1000*i);
    }
    /* compute on the GPU */
    vecaddgpu( r, a, b, n );
    /* compute on the host to compare */
    for( i = 0; i < n; ++i ) e[i] = a[i] + b[i];
    /* compare results */
    errs = 0;
    for( i = 0; i < n; ++i ){
     if( r[i] != e[i] ){
     ++errs;
       }
}
printf( "%d errors found\n", errs );
return errs;
}
```

OpenACC

Vector addition example

kernels here

function call here

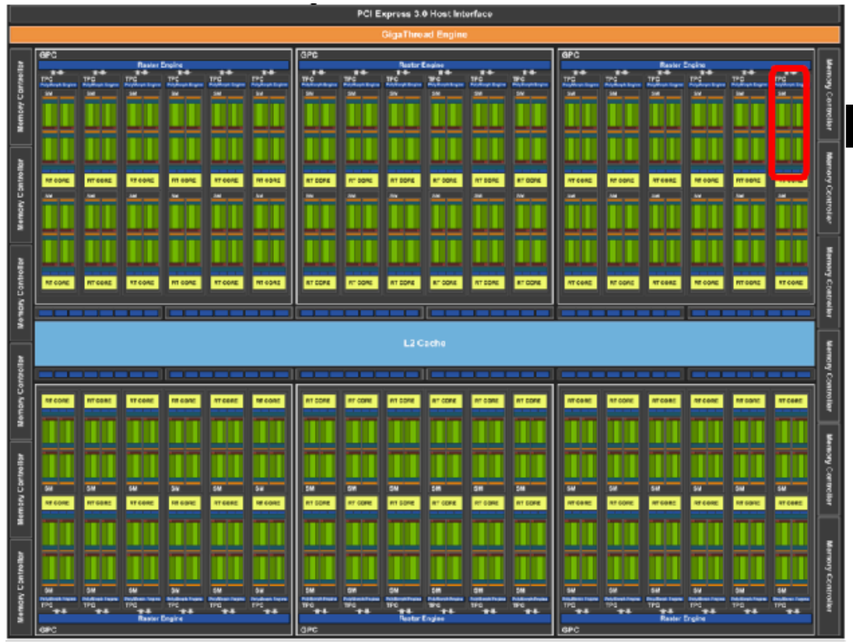
Only one* line needed! 😊

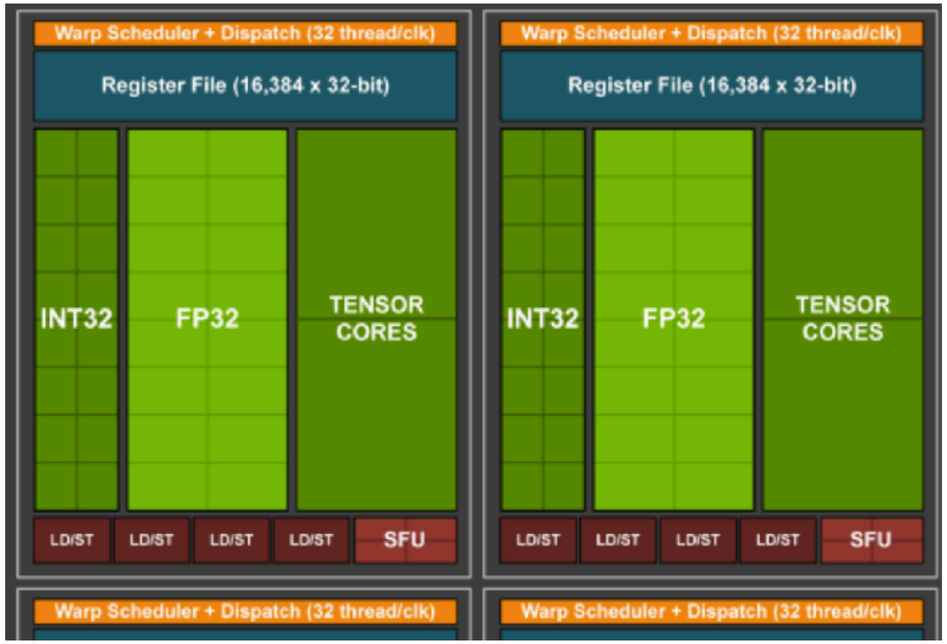
```
#include <stdio.h>
#include <stdlib.h>
void vecaddgpu( float *restrict r, float *a, float *b, int n ){
    #pragma acc kernels loop copyin(a[0:n],b[0:n]) copyout(r[0:n])
    for( int i = 0; i < n; ++i ) r[i] = a[i] + b[i];
}

int main( int argc, char* argv[] ){
    int n; /* vector length */
    float * a; /* input vector 1 */
    float * b; /* input vector 2 */
    float * r; /* output vector */
    float * e; /* expected output values */
    int i, errs;
    if( argc > 1 ) n = atoi( argv[1] );
    else n = 100000; /* default vector length */
    if( n <= 0 ) n = 100000;
    a = (float*)malloc( n*sizeof(float) );
    b = (float*)malloc( n*sizeof(float) );
    r = (float*)malloc( n*sizeof(float) );
    e = (float*)malloc( n*sizeof(float) );
    for( i = 0; i < n; ++i ){
        a[i] = (float)(i+1);
        b[i] = (float)(1000*i);
    }
    /* compute on the GPU */
    vecaddgpu( r, a, b, n );
    /* compute on the host to compare */
    for( i = 0; i < n; ++i ) e[i] = a[i] + b[i];
    /* compare results */
    errs = 0;
    for( i = 0; i < n; ++i ){
     if( r[i] != e[i] ){
     ++errs;
       }
}
printf( "%d errors found\n", errs );
return errs;
}
```

CUDA

Streaming Multiprocessor (SM)

- Group of elements that run one or more **warps**



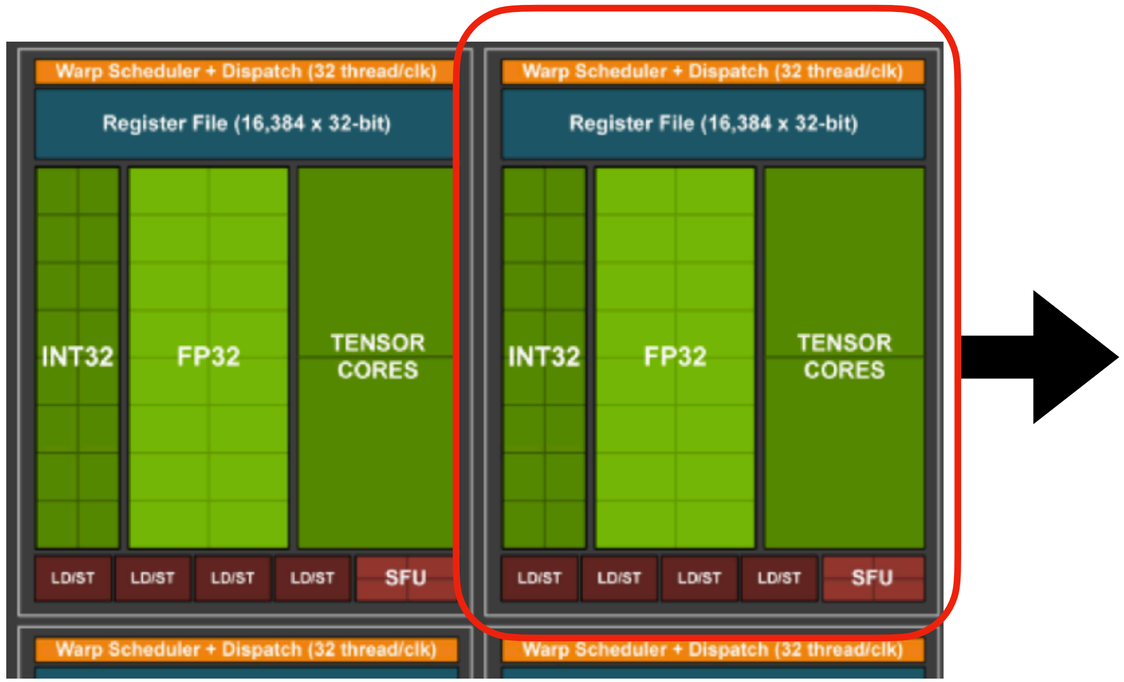


Source: https://gitlab.ethz.ch/hpcse-public-repos/hpcseii-spring-2022-lecture

Source: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

CUDA

Warp scheduler



Is basically a team of threads:

- Scheduled together
- SIMT (different data)
- Independent from other teams in the SM
- Share cache

Source: https://gitlab.ethz.ch/hpcse-public-repos/hpcseii-spring-2022-lecture

Source: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

CUDA

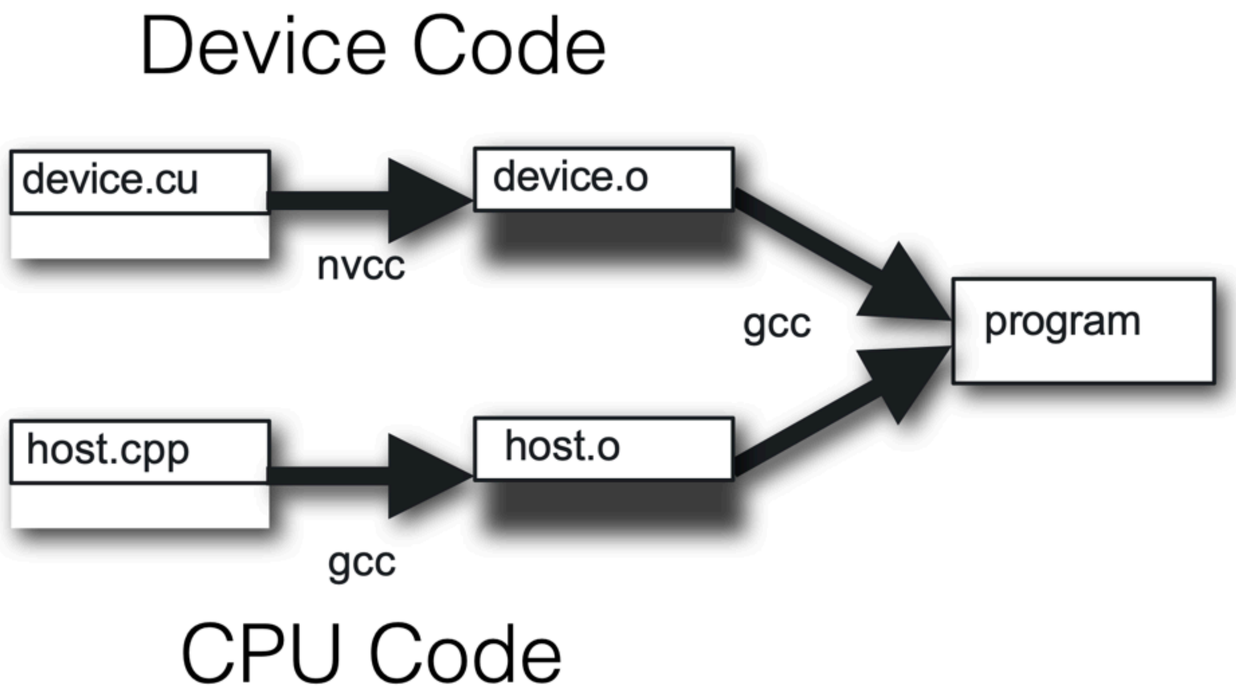
Hello World

```
__global__ void mykernel(void)
{
  printf("Hello World, I'm GPU!\n");
}

int main(void)
{
  mykernel<<<1,1>>>();
  printf("Hello World, I'm CPU!\n");
  return 0;
}
```

Source: https://gitlab.ethz.ch/hpcse-public-repos/hpcseii-spring-2022-lecture

CUDA

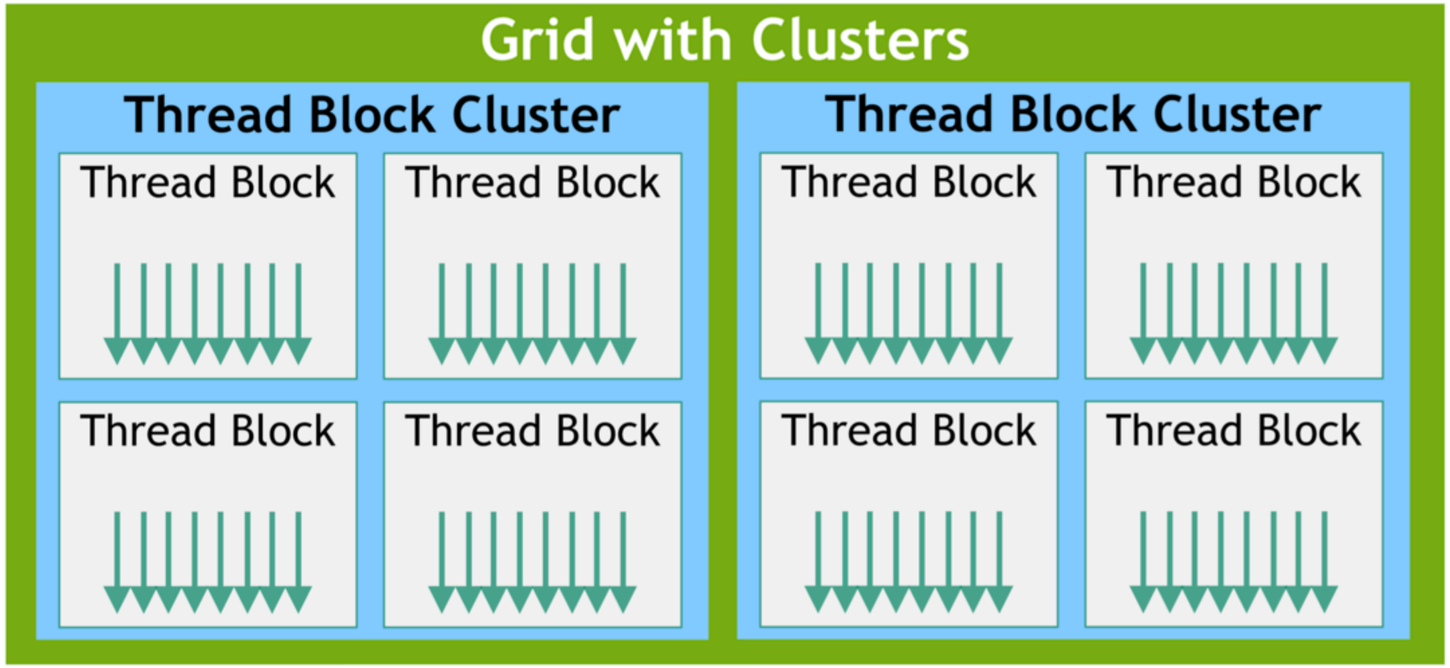
Compilation



Source: https://gitlab.ethz.ch/hpcse-public-repos/hpcseii-spring-2022-lecture

CUDA

Thread hierarchy



*Cluster level is optional

Source: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

CUDA

Vector addition example

```
#define N 1024
int main(void)
{
   int *a, *b, *c; // host copies of a, b, c
   int *d_a, *d_b, *d_c;  // device copies of a, b, c
   int size = N * sizeof(int);

   // Alloc space for device copies of a, b, c
   cudaMalloc((void **)&d_a, size);
   cudaMalloc((void **)&d_b, size);
   cudaMalloc((void **)&d_c, size);

// Alloc space for host copies of a, b, c
// and setup input values
   a = (int *)malloc(size); random_ints(a, N);
   b = (int *)malloc(size); random_ints(b, N);
   c = (int *)malloc(size);
```

```
// Copy inputs to device
cudaMemcpy(d_a, &a, size,  cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &b, size,  cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<N,1>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(&c, d_c, size,  cudaMemcpyDeviceToHost);

// Cleanup
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);


return 0;
}
```

Source: https://gitlab.ethz.ch/hpcse-public-repos/hpcseii-spring-2022-lecture

CUDA

Vector addition example

```
#define N 1024
int main(void)
{
   int *a, *b, *c; // host copies of a, b, c
   int *d_a, *d_b, *d_c;  // device copies of a, b, c
   int size = N * sizeof(int);

   // Alloc space for device copies of a, b, c
   cudaMalloc((void **)&d_a, size);
   cudaMalloc((void **)&d_b, size);
   cudaMalloc((void **)&d_c, size);

// Alloc space for host copies of a, b, c
// and setup input values
   a = (int *)malloc(size); random_ints(a, N);
   b = (int *)malloc(size); random_ints(b, N);
   c = (int *)malloc(size);
```

```
__global__ void add(int *a, int *b, int *c)
{
 int myElement =  blockIdx.x * blockDim.x + threadIdx.x;
 c[myElement] = a[myElement] + b[myElement];
}
```

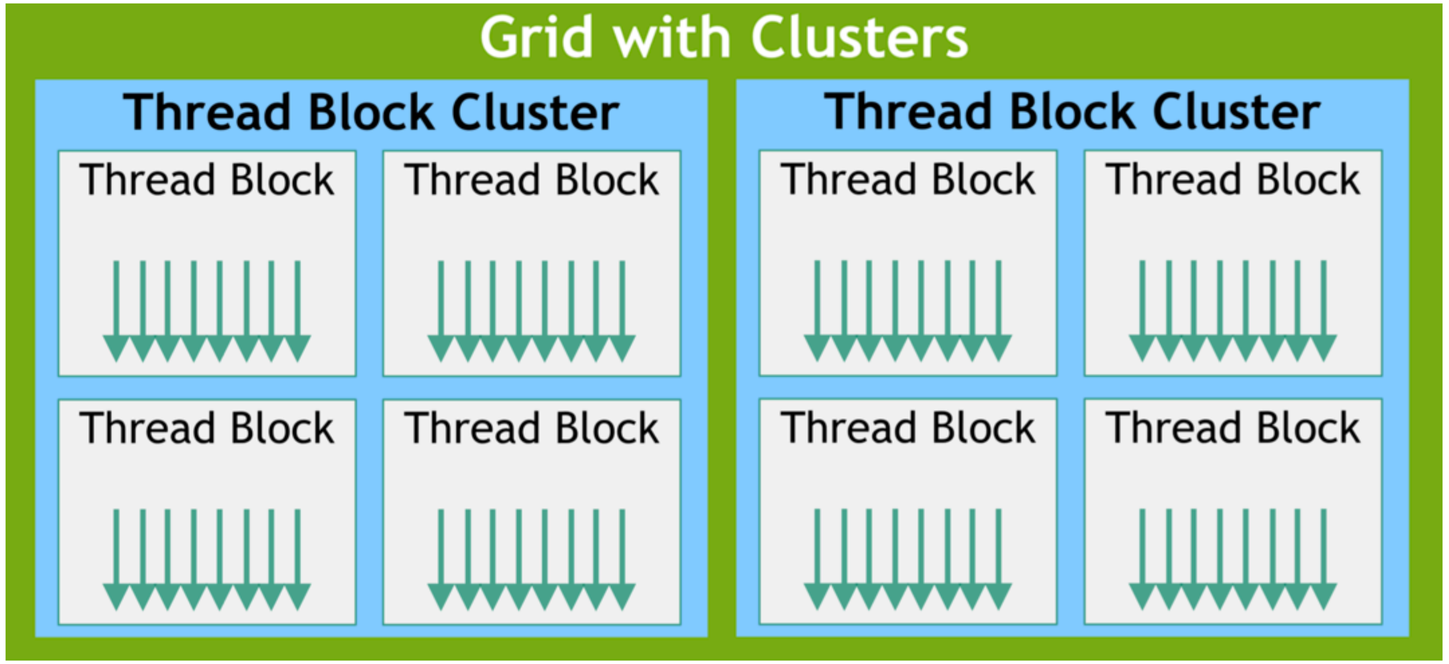
```
// Copy inputs to device
cudaMemcpy(d_a, &a, size,  cudaMemcpyHostToDevice);
cudaMemcpy(d_b, &b, size,  cudaMemcpyHostToDevice);

// Launch add() kernel on GPU
add<<<N,1>>>(d_a, d_b, d_c);

// Copy result back to host
cudaMemcpy(&c, d_c, size,  cudaMemcpyDeviceToHost);

// Cleanup
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_c);


return 0;
}
```

Source: https://gitlab.ethz.ch/hpcse-public-repos/hpcseii-spring-2022-lecture

CUDA

GPU shared memory

- Allocated manually
- Shared among threads **in a block**



CUDA

GPU shared memory example 1

```
N = 4096;
dim3 blockDim(32,32) // 512 Threads
dim3 gridDim(N/32, N/32) // 128x128 Blocks = 16k Blocks
size_t shmSize = 2*32*32*sizeof(double);
my2dProblem<<<gridDim, blockDim, shmSize>>>(grid1, grid2, N);
```

```
__global__ void my2DProblem(double* grid1, double* grid2, size_t N)
{
  const size_t m     = blockIdx.x * blockDim.x + threadIdx.x;
  const size_t t     = threadIdx.x;

  extern __shared__ double s[];
  s[2*t + 0] = grid1[m];
  s[2*t + 1] = grid2[m];
  __syncthreads(); // Important: synchronize to make sure every thread saved their values before accessing

  double myCalc = 0.0;
  for (size_t i = 0; i < blockDim.x; i++) myCalc += sqrt(s[2*i] + s[2*I+1]);
}
```

Source: https://gitlab.ethz.ch/hpcse-public-repos/hpcseii-spring-2022-lecture

CUDA

GPU shared memory example 2

```
__global__ void kernel(double *a, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    double x = idx < N ? a[N] : 0.0;

    __shared__ double shared[1024];
    shared[threadIdx.x] = x;
    __syncthreads();

    // Now every thread of a block has the value `x`
    // of every other thread in a block.
}
```

Source: https://gitlab.ethz.ch/hpcse-public-repos/hpcseii-spring-2022-lecture

SYCL

Overview

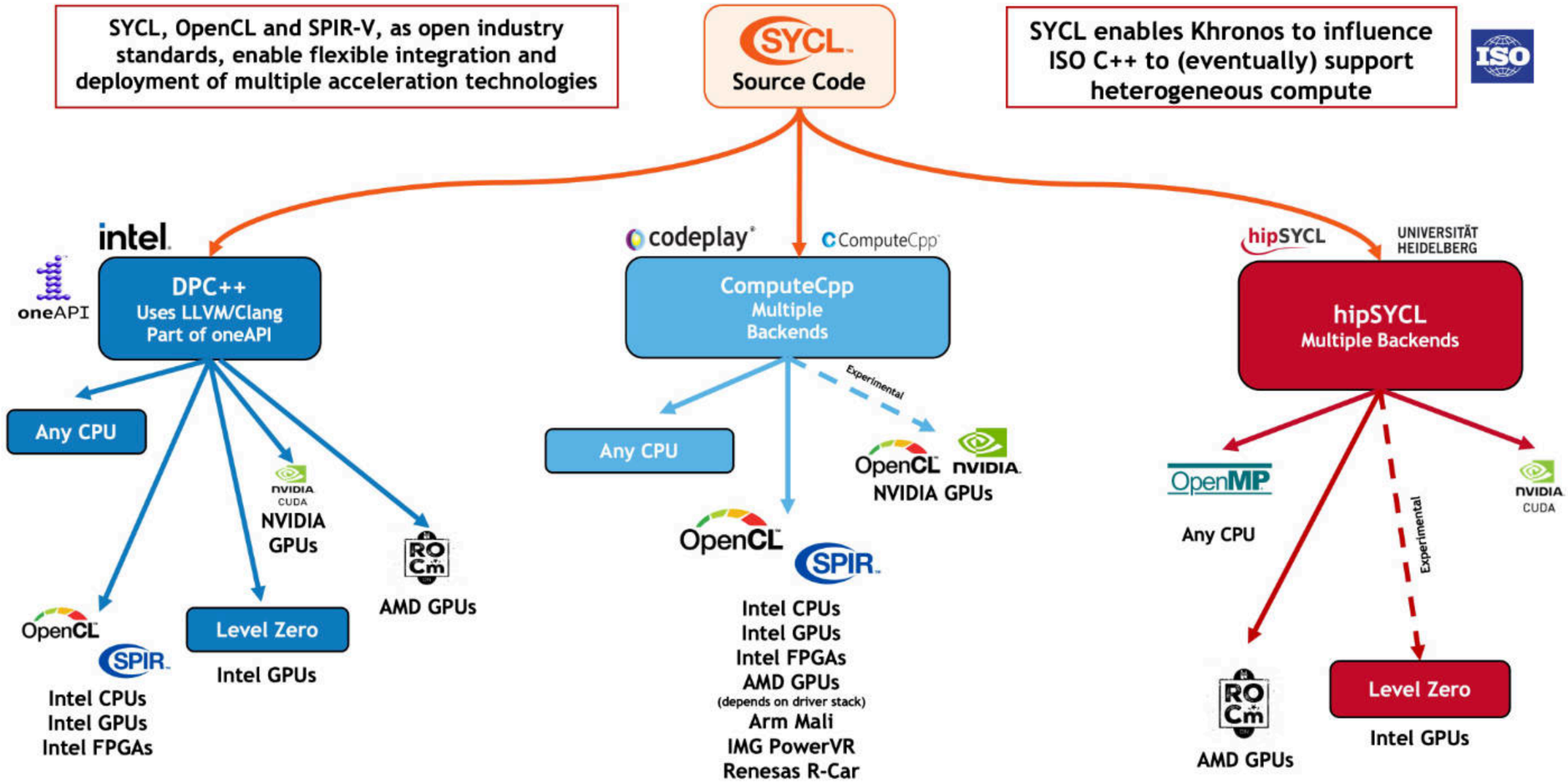
- Supports various devices (CPUs, GPUs, FPGAs)
- Vendor independent
- Std C++ codebase

- Heterogeneous programming support
- More extendibility and flexibility with simpler codes
- No separate source files

Source: https://www.codeproject.com/Articles/5324827/Comparing-Programming-models-SYCL-and-CUDA

SYCL

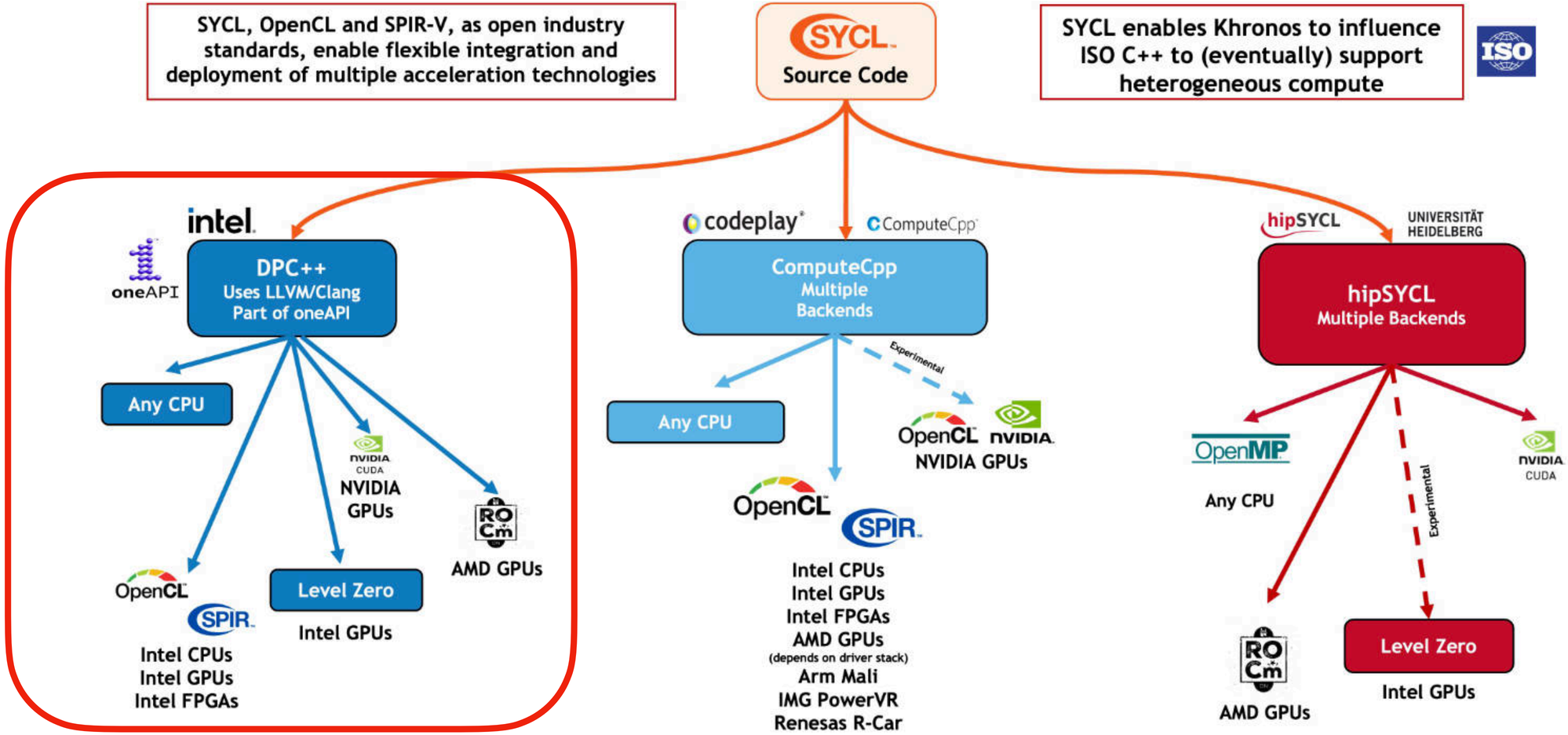
Implementations



Source: https://www.khronos.org/sycl/

SYCL

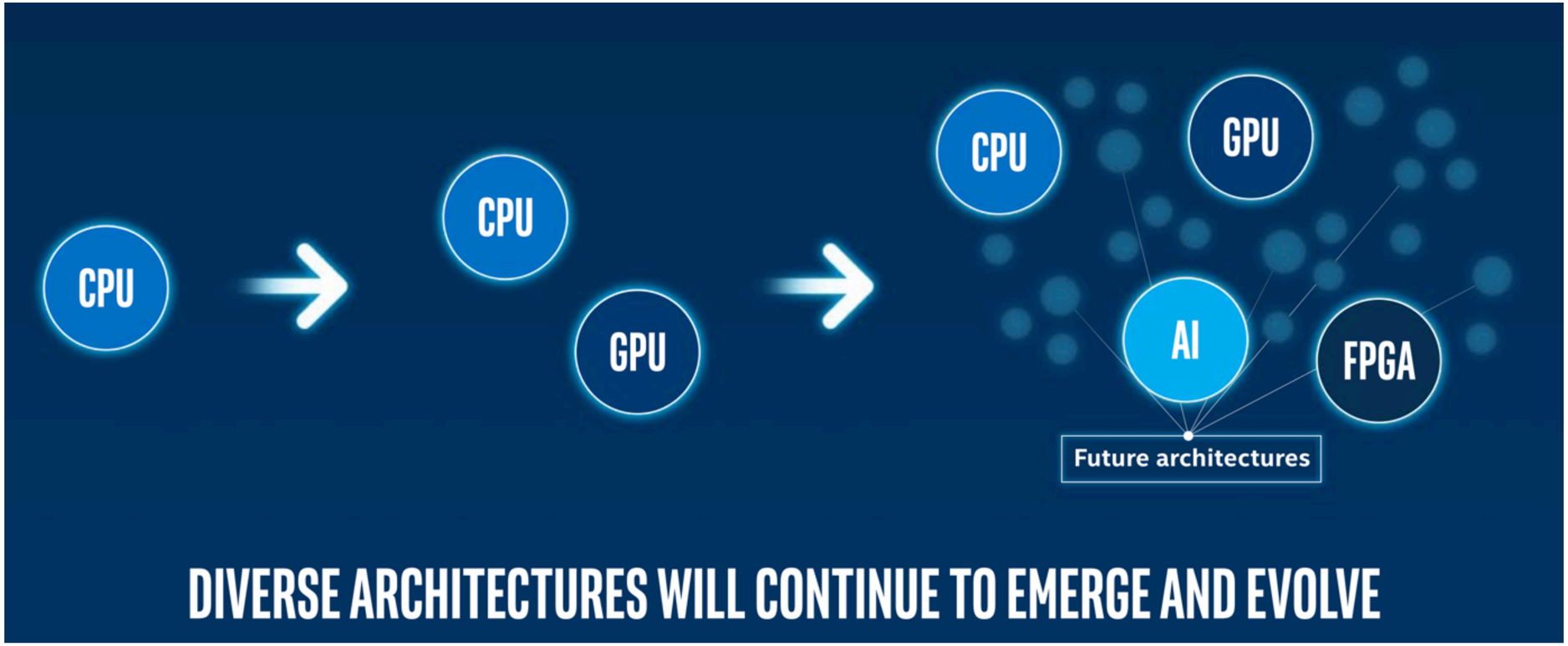
Implementations



Source: https://www.khronos.org/sycl/

oneAPI (DPC++ by Intel, open source in LLVM)

Intel's argument



Source: https://newsroom.intel.com/wp-content/uploads/sites/11/2019/11/intel-oneapi-info.pdf

SYCL

NVIDIA implementation:

Compilation with DPC++ compiler for NVIDIA gpu:

```
clang++ -fsycl -fsycl-targets=nvptx-64-nvidia-cuda-sycldevice sample.cpp -o sample

SYCL_BE=PI_CUDA ./sample
```

SYCL selector code for NVIDIA gpu:

```
class CUDASelector : public cl::sycl::device_selector {
   public:
      int operator()(const cl::sycl::device &Device) const override {
         using namespace cl::sycl::info;

         const std::string DeviceName = Device.get_info<device::name>();
         const std::string DeviceVendor = Device.get_info<device::vendor>();

         if (Device.is_gpu() && (DeviceName.find("NVIDIA") != std::string::npos))
           { return 1; };
         return -1;
      }
};
```

Source: https://codeplay.com/portal/news/2020/02/03/codeplay-contribution-to-dpcpp-brings-sycl-support-for-nvidia-gpus.html

SYCL

Device selector Hello World

```
#include <CL/sycl.hpp>
int main() {
  sycl::device d;

  try {
    d = sycl::device(sycl::gpu_selector());
  } catch (sycl::exception const &e) {
    std::cout << "Cannot select a GPU\n" << e.what() << "\n";
    std::cout << "Using a CPU device\n";
    d = sycl::device(sycl::cpu_selector());
  }

  std::cout << "Using " << d.get_info<sycl::info::device::name>();
}
```

Source: https://sycl.readthedocs.io/en/latest/iface/device-selector.html

SYCL

Vector addition example

```
#include <CL/sycl.hpp>
using namespace sycl;

const int SIZE = 10;

void show_platforms() {
  auto platforms = platform::get_platforms();

  for (auto &platform : platforms) {
    std::cout << "Platform: "
         << platform.get_info<info::platform::name>()
         << std::endl;

    auto devices = platform.get_devices();
    for (auto &device : devices ) {
      std::cout << "  Device: "
     << device.get_info<info::device::name>()
     << std::endl;
    }
  }
}
```

```
void vec_add(int *a, int *b, int *c) {
  range<1> a_size{SIZE};

  buffer<int> a_buf(a, a_size);
  buffer<int> b_buf(b, a_size);
  buffer<int> c_buf(c, a_size);

  queue q;

  q.submit([&](handler &h) {
      auto c_res = c_buf.get_access<access::mode::write>(h);
      auto a_in = a_buf.get_access<access::mode::read>(h);
      auto b_in = b_buf.get_access<access::mode::read>(h);
int main() {
  int a[SIZE], b[SIZE], c[SIZE];
          [=](id<1> idx) {
  for (int i = 0; i < SIZE; ++i) { + b_in[idx];
    a[i] = i;
    b[i] = i;
    c[i] = i;
  }

  show_platforms();
  vec_add(a, b, c);

  for (int i = 0; i < SIZE; i++) std::cout << c[i] << std::endl;
  return 0;
}
```

Source: https://docs.oneapi.io/versions/latest/model/sample-program.html

Thanks for your attention

(Q) ? A :

oneAPI

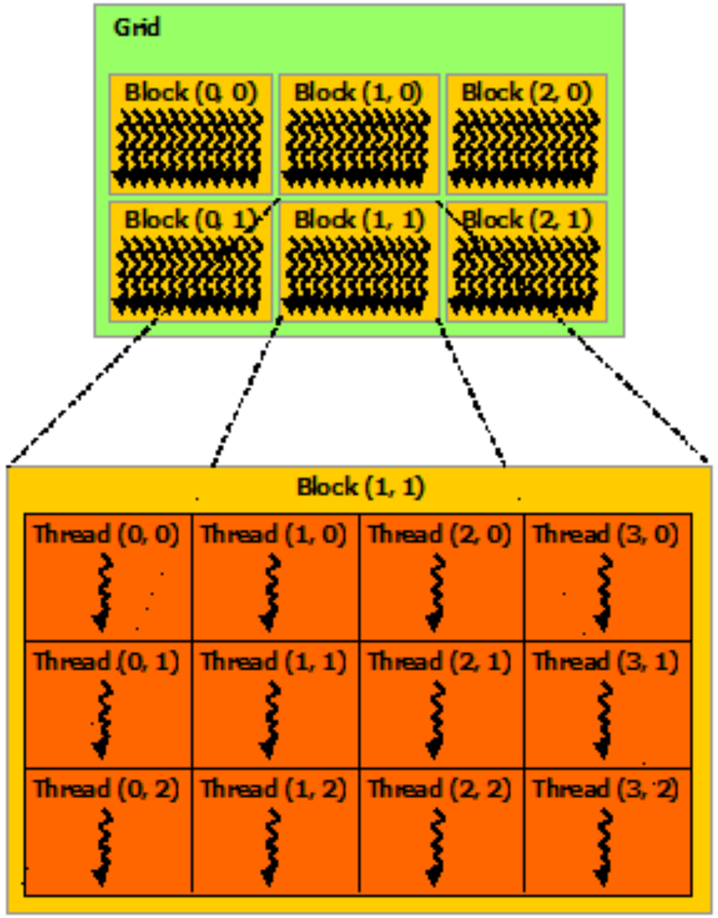
oneDAL (Data Analytics Library) example

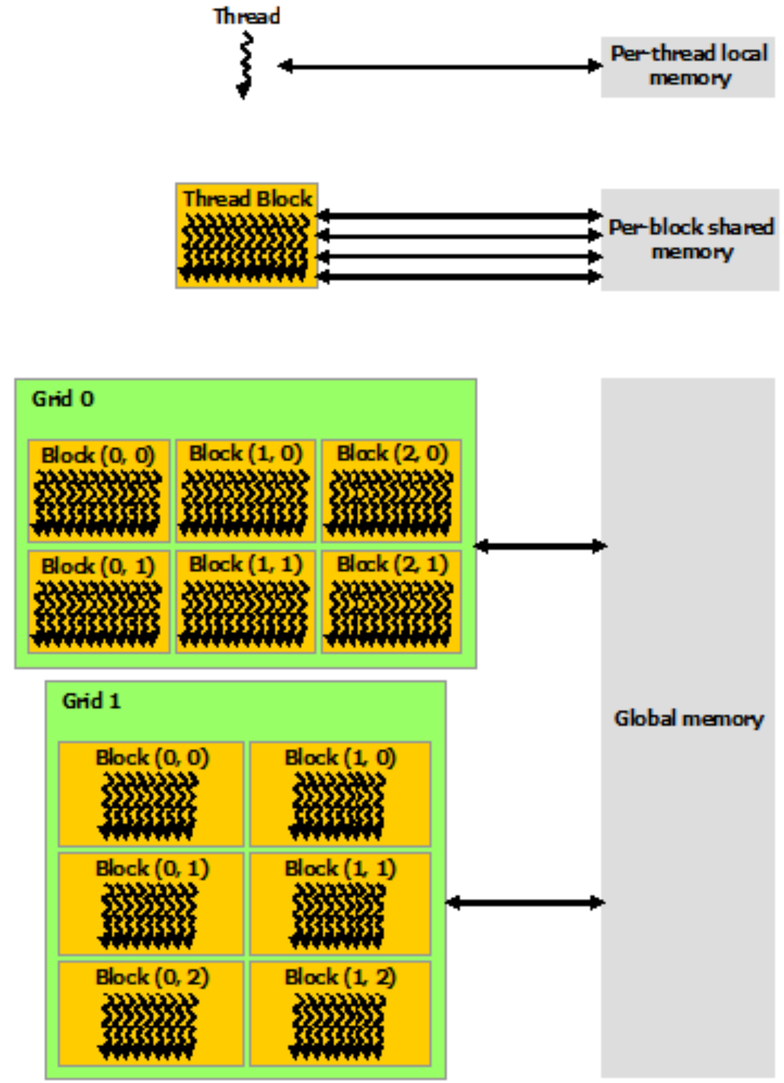
```
#include "oneapi/dal.hpp"
#include <cassert>
#include <iostream>

const auto queue = sycl::queue{ sycl::gpu_selector{} };

using namespace oneapi;

const auto data = dal::read<dal::table>(queue, dal::csv::data_source{"data.csv"});

const auto pca_desc = dal::pca::descriptor<float>
   .set_component_count(3)
   .set_deterministic(true);
const dal::pca::train_result train_res = dal::train(queue, pca_desc, data);

const dal::table eigenvectors = train_res.get_eigenvectors();
const auto acc = dal::row_accessor<const float>{eigenvectors};
for (std::int64_t i = 0; i < eigenvectors.row_count(); i++) {
   /* Get i-th row from the table, the eigenvector stores pointer to USM */
   const dal::array<float> eigenvector = acc.pull(queue, {i, i + 1});
   assert(eigenvector.get_count() == eigenvectors.get_column_count());

   std::cout << i << "-th eigenvector: ";
   for (std::int64_t j = 0; j < eigenvector.get_count(); j++) {
      std::cout << eigenvector[j] << " ";
   }
   std::cout << std::endl;
}

const dal::pca::model model = train_res.get_model();
const dal::table data_transformed =
   dal::infer(queue, pca_desc, data).get_transformed_data();

assert(data_transformed.column_count() == 3);
```

Source: https://oneapi-src.github.io/oneDAL/quick-start.html

Interlude

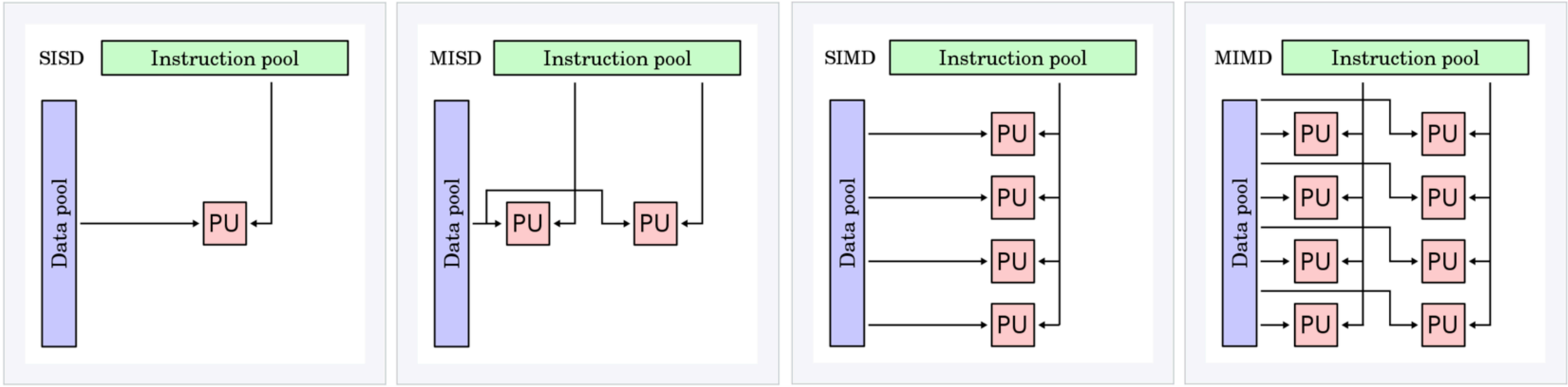
Most common thread hierarchy





Interlude

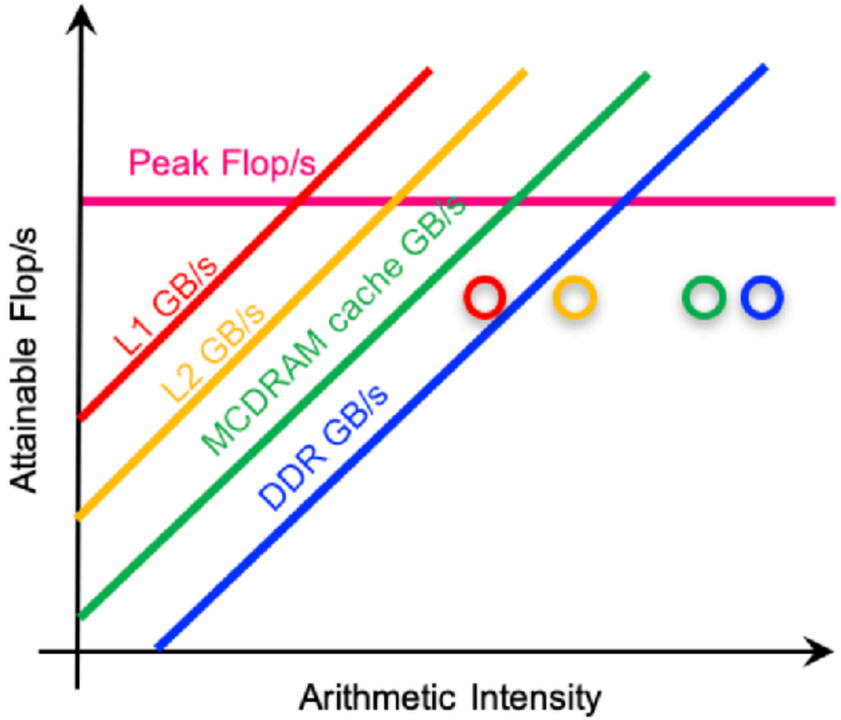
Flynn’s taxonomy

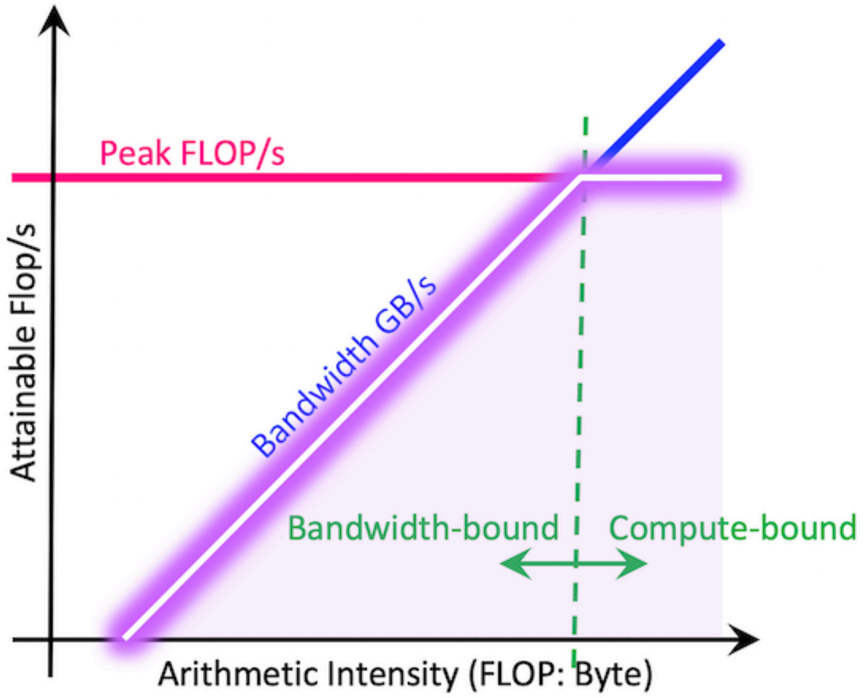


Source: Wikipedia

Interlude

Roofline model

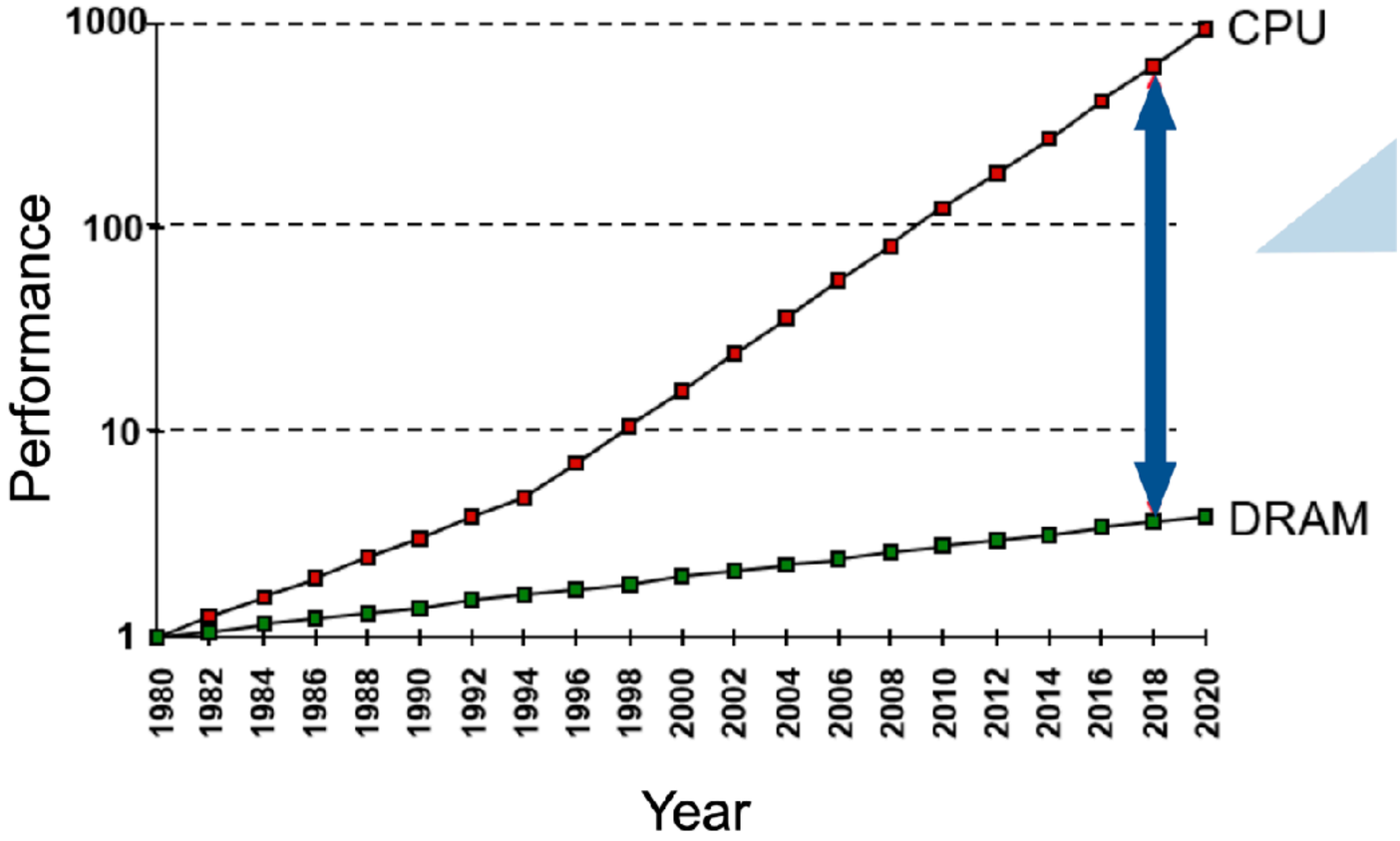




Source: https://docs.nersc.gov/tools/performance/roofline/

Interlude

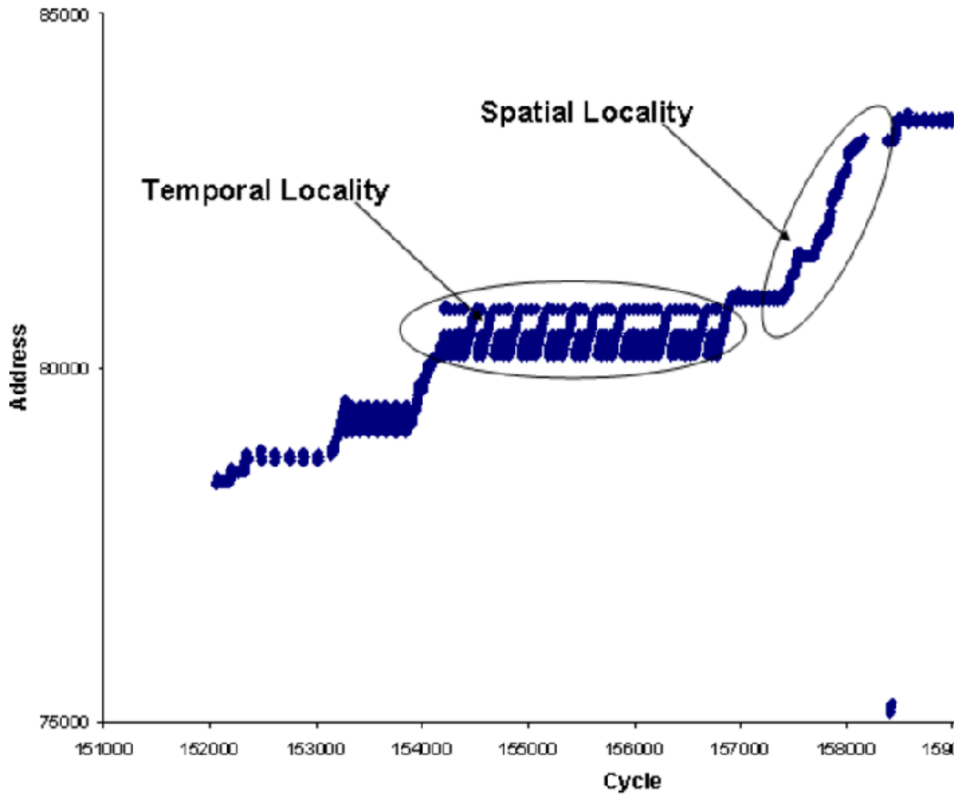
Performance memory gap

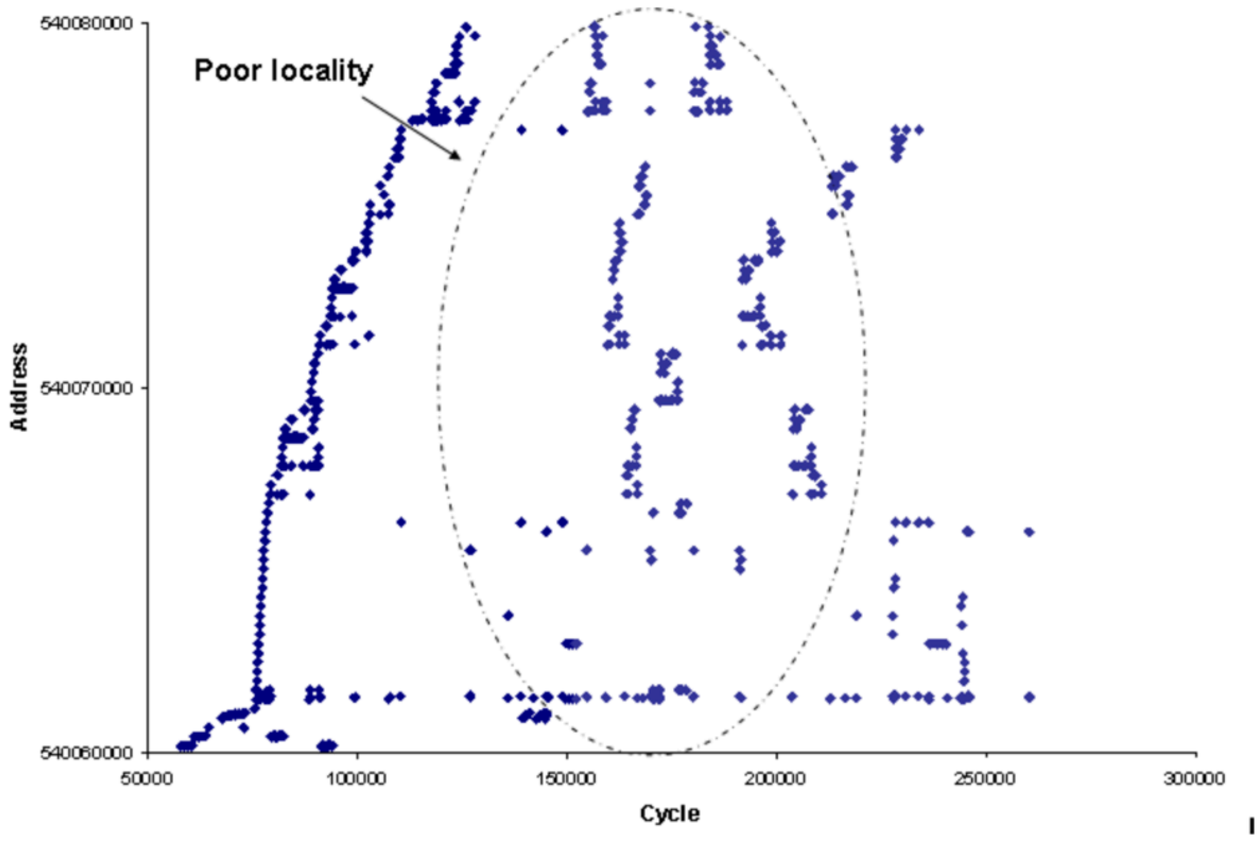


Source: https://db.in.tum.de/teaching/ss21/dataprocessingonmodernhardware/MH_2.pdf?lang=de

Interlude

Cache locality





Source: https://gitlab.ethz.ch/hpcsel_hs21/lecture