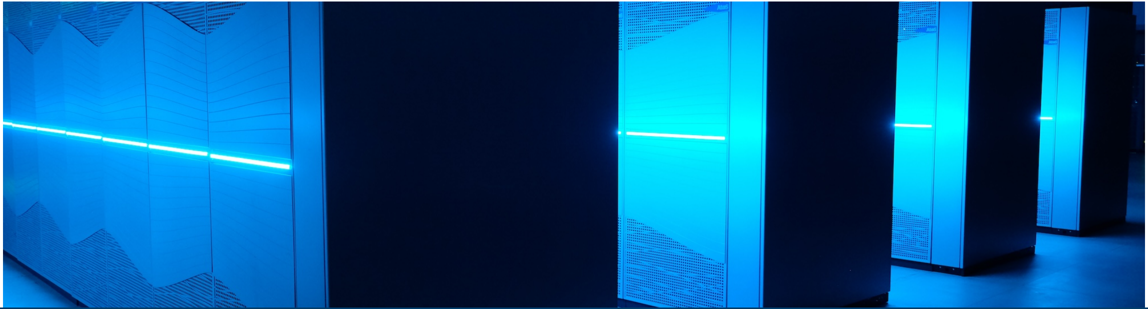


Taming The GPU Beasts & CUDA

natESM Training Workshop

6 November 2024 | Andreas Herten | Forschungszentrum Jülich

Member of the Helmholtz Association

JÜLICH
Forschungszentrum

Outline

Plan

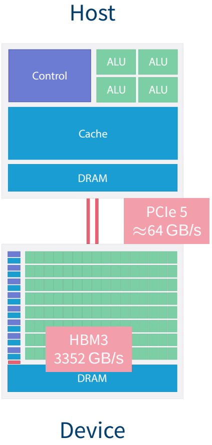
- Concrete → abstract
- Vendor-specific → portable
- Prepared exercises
  1. CUDA
  2. OpenACC
  3. Kokkos
- Usually, C and Fortran
- Running example: **Jacobi**, but sometimes *side-quests*
- Timetable online, but only *guideline*

Platform

GPU Architecture Design

GPU optimized to **hide latency**

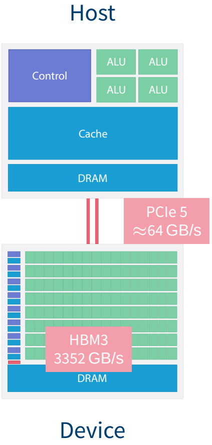
- Memory
  - GPU has small (40 GB), but high-speed memory 1555 GB/s
  - Stage data to GPU memory: via PCIe 4 (32 GB/s) or PCIe 5 (64 GB/s) bus



GPU Architecture Design

GPU optimized to **hide latency**

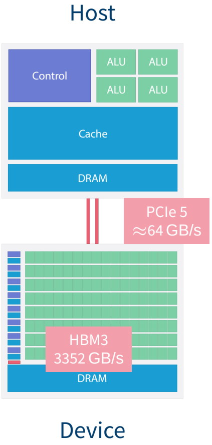
- Memory
  - GPU has small (40 GB), but high-speed memory 1555 GB/s
  - Stage data to GPU memory: via PCIe 4 (32 GB/s) or PCIe 5 (64 GB/s) bus



GPU Architecture Design

GPU optimized to **hide latency**

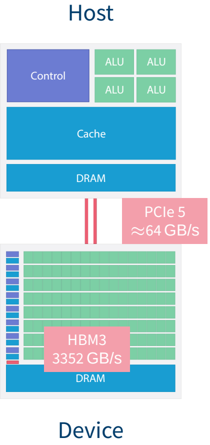
- Memory
  - GPU has small (40 GB), but high-speed memory 1555 GB/s
  - Stage data to GPU memory: via PCIe 4 (32 GB/s) or PCIe 5 (64 GB/s) bus
  - Stage automatically (*Unified Memory*), or manually



GPU Architecture Design

GPU optimized to **hide latency**

- Memory
  - GPU has small (40 GB), but high-speed memory 1555 GB/s
  - Stage data to GPU memory: via PCIe 4 (32 GB/s) or PCIe 5 (64 GB/s) bus
  - Stage automatically (*Unified Memory*), or manually
  - GH200: NVLink C2C 900 GB/s
- Two engines: Overlap compute and copy



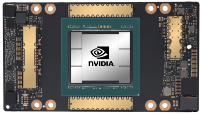
GPU Architecture Design

GPU optimized to **hide latency**

- Memory
  - GPU has small (40 GB), but high-speed memory 1555 GB/s
  - Stage data to GPU memory: via PCIe 4 (32 GB/s) or PCIe 5 (64 GB/s) bus
  - Stage automatically (*Unified Memory*), or manually
  - GH200: NVLink C2C 900 GB/s
- Two engines: Overlap compute and copy

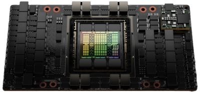
A100

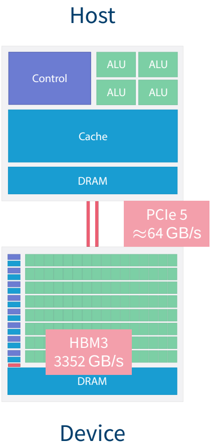
40 GB RAM, 1555 GB/s



H100

80 GB RAM, 3352 GB/s





GPU Architecture Design

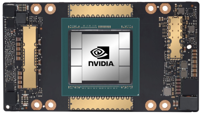
GPU optimized to **hide latency**

- Memory
  - GPU has small (40 GB), but high-speed memory 1555 GB/s
  - Stage data to GPU memory: via PCIe 4 (32 GB/s) or PCIe 5 (64 GB/s) bus
  - Stage automatically (*Unified Memory*), or manually
  - GH200: NVLink C2C 900 GB/s
- Two engines: Overlap compute and copy

- SIMT

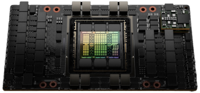
A100

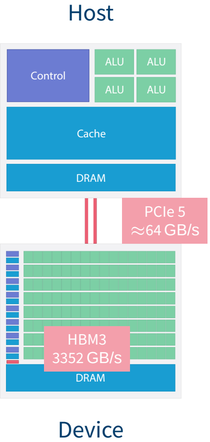
40 GB RAM, 1555 GB/s



H100

80 GB RAM, 3352 GB/s



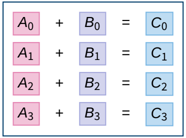


SIMT

SIMT = SIMD $\oplus$ SMT

- CPU:
  - Single Instruction, Multiple Data (SIMD)

Scalar



$A_0 + B_0 = C_0$

$A_1 + B_1 = C_1$

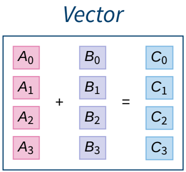
$A_2 + B_2 = C_2$

$A_3 + B_3 = C_3$

SIMT

SIMT = SIMD ⊕ SMT

- CPU:
  - Single Instruction, Multiple Data (SIMD)

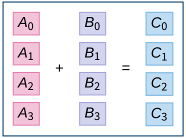


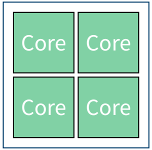
SIMT

SIMT = SIMD $\oplus$ SMT

- CPU:
  - Single Instruction, Multiple Data (SIMD)
  - Simultaneous Multithreading (SMT)

Vector


$$\begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{pmatrix} + \begin{pmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{pmatrix} = \begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \end{pmatrix}$$

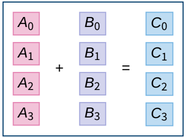


SIMT

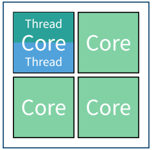
SIMT = SIMD ⊕ SMT

- CPU:
  - Single Instruction, Multiple Data (SIMD)
  - Simultaneous Multithreading (SMT)

Vector



SMT

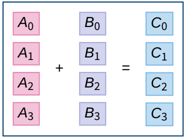


SIMT

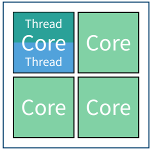
SIMT = SIMD $\oplus$ SMT

- CPU:
    - Single Instruction, Multiple Data (SIMD)
    - Simultaneous Multithreading (SMT)
- GPU: Single Instruction, Multiple Threads (SIMT)

Vector



SMT

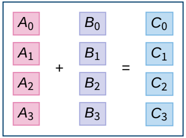


SIMT

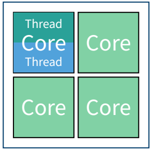
SIMT = SIMD ⊕ SMT

- CPU:
    - Single Instruction, Multiple Data (SIMD)
    - Simultaneous Multithreading (SMT)
- GPU: Single Instruction, Multiple Threads (SIMT)

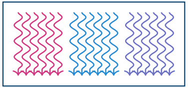
Vector



SMT



SIMT

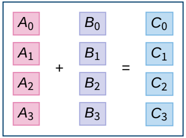


SIMT

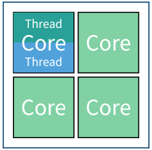
SIMT = SIMD ⊕ SMT

- CPU:
  - Single Instruction, Multiple Data (SIMD)
  - Simultaneous Multithreading (SMT)
- GPU: Single Instruction, Multiple Threads (SIMT)
  - CPU core ≊ GPU multiprocessor (SM)
  - Working unit: set of threads (32, a *warp*)
  - Fast switching of threads (large register file)
  - Branching 

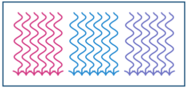
Vector



SMT

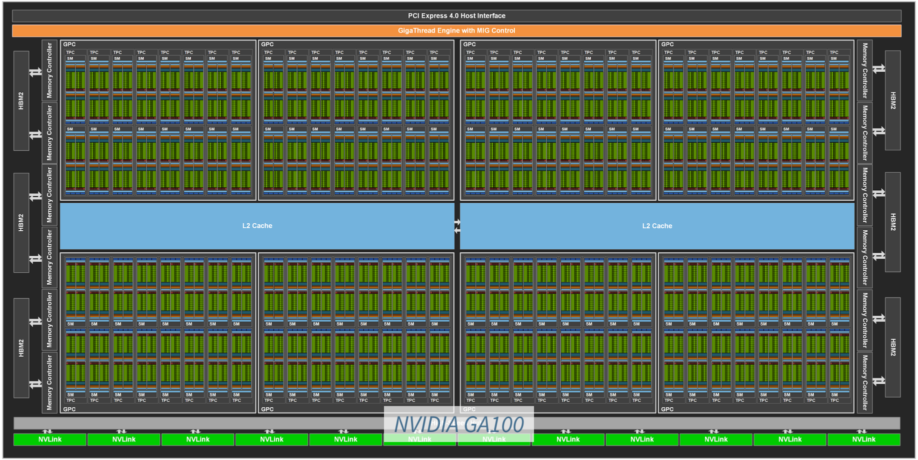


SIMT

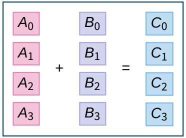


SIMT

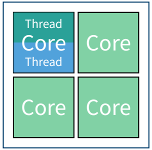
SIMT = SIMD ⊕ SMT



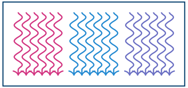
Vector



SMT



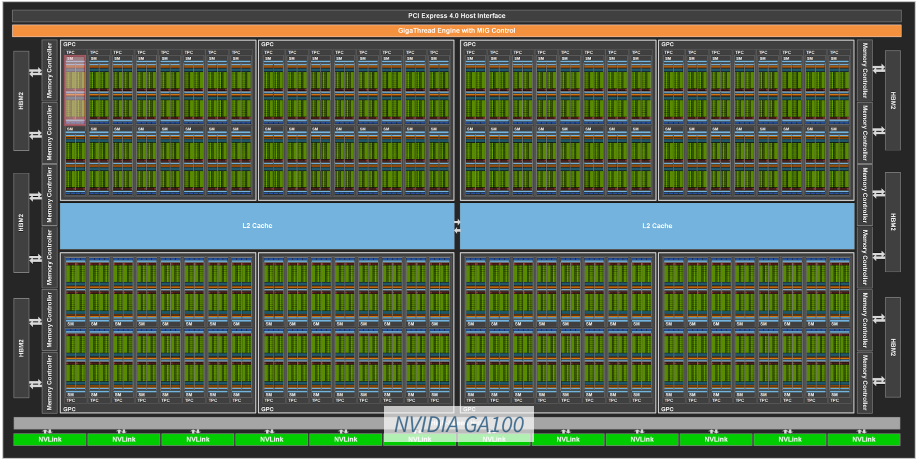
SIMT



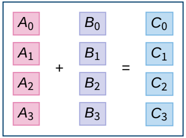
Graphics: img:amperepictures

SIMT

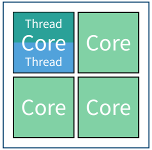
SIMT = SIMD ⊕ SMT



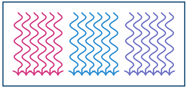
Vector



SMT



SIMT

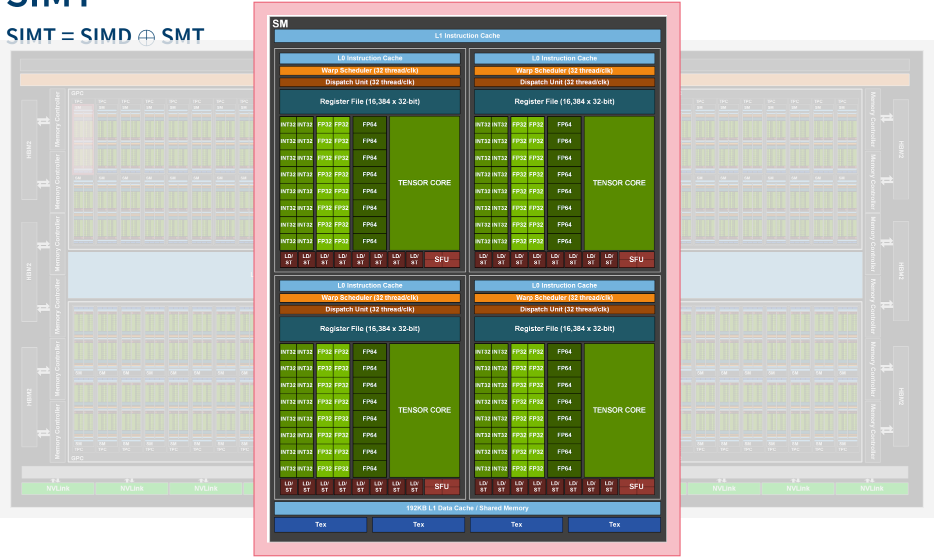


Graphics: img:amperepictures

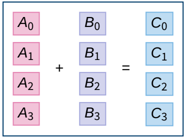
SIMT

SIMT = SIMD ⊕ SMT

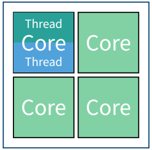
Multiprocessor



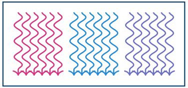
Vector



SMT



SIMT

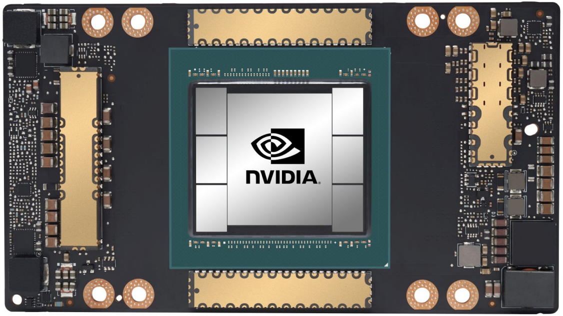


Graphics: img:amperepictures

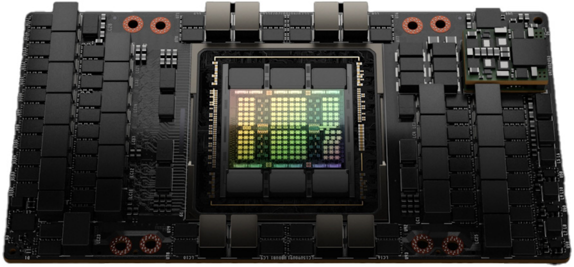
A100 vs H100

Comparison of last vs. current generation

A100



H100



A100 vs H100

Comparison of last vs. current generation

A100

H100

A100 vs H100

Comparison of last vs. current generation

A100

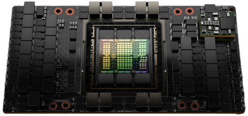
H100

CPU vs. GPU

Let’s summarize this!

Optimized for **low latency**

- \+ Large main memory
- \+ Fast clock rate
- \+ Large caches
- \+ Branch prediction
- \+ Powerful ALU
- – Relatively low memory bandwidth
- – Cache misses costly
- – Low performance per watt

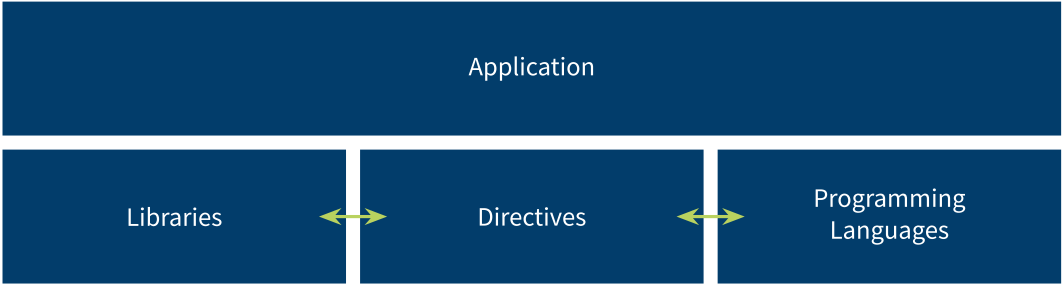


Optimized for **high throughput**

- \+ High bandwidth main memory
- \+ Latency tolerant (parallelism)
- \+ More compute resources
- \+ High performance per watt
- – Limited memory capacity
- – Low per-thread performance
- – Extension card

Programming GPUs

Summary of Acceleration Possibilities



Application

Libraries

Directives

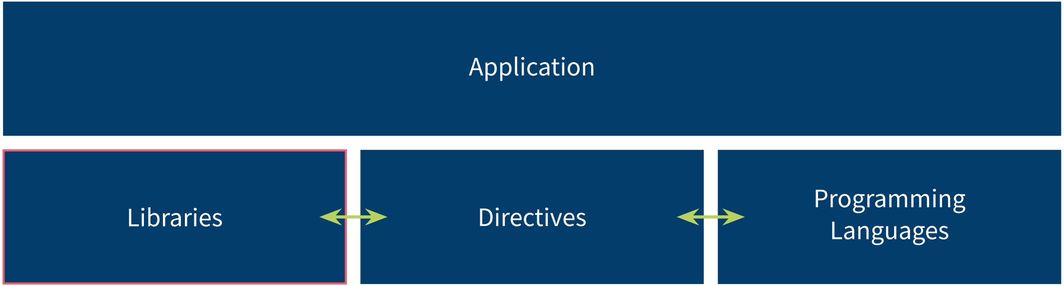
Programming Languages

Drop-in Acceleration

Easy Acceleration

Flexible Acceleration

Summary of Acceleration Possibilities



Application

Libraries

Directives

Programming Languages

Drop-in Acceleration

Easy Acceleration

Flexible Acceleration

Libraries

Programming GPUs is easy: Just don't!

Libraries

Programming GPUs is easy: Just don’t!

Use applications & libraries

Libraries

Programming GPUs is easy: Just don't!

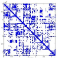
Use applications & libraries

Wizard: wizard

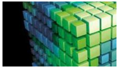
Libraries

Programming GPUs is easy: Just don't!

Use applications & libraries

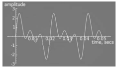


cuSPARSE

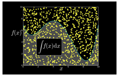


cuBLAS

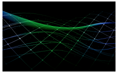
cuDNN



cuFFT



cuRAND



CUDA Math

OpenCV

Thrust

ArrayFire

Numba

CuPy

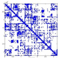
Wizard: wizard

Libraries

Programming GPUs is easy: Just don't!

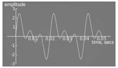
Use applications & libraries

cuBLAS

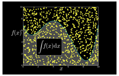


cuSPARSE

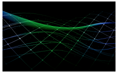
cuDNN



cuFFT



cuRAND



CUDA Math

OpenCV

Thrust

ArrayFire

Numba

CuPy

Wizard: wizard

cuBLAS

Code example

```
int a = 42;  int n = 10;
float x[n], y[n];
// fill x, y

cublasHandle_t handle;
cublasCreate(&handle);

float * d_x, * d_y;
cudaMallocManaged(&d_x, n * sizeof(x[0]));
cudaMallocManaged(&d_y, n * sizeof(y[0]));

cublasSaxpy(handle, n, a, d_x, 1, d_y, 1);

cublasGetVector(n, sizeof(y[0]), d_y, 1, y, 1);

cudaFree(d_x); cudaFree(d_y);
cublasDestroy(handle);
```

cuBLAS

Code example

```
int a = 42;  int n = 10;
float x[n], y[n];
// fill x, y

cublasHandle_t handle;
cublasCreate(&handle);

float * d_x, * d_y;
cudaMallocManaged(&d_x, n * sizeof(x[0]));
cudaMallocManaged(&d_y, n * sizeof(y[0]));

cublasSaxpy(handle, n, a, d_x, 1, d_y, 1);

cublasGetVector(n, sizeof(y[0]), d_y, 1, y, 1);

cudaFree(d_x); cudaFree(d_y);
cublasDestroy(handle);
```

Initialize

cuBLAS

Code example

```
int a = 42;  int n = 10;
float x[n], y[n];
// fill x, y

cublasHandle_t handle;
cublasCreate(&handle);

float * d_x, * d_y;
cudaMallocManaged(&d_x, n * sizeof(x[0]));
cudaMallocManaged(&d_y, n * sizeof(y[0]));

cublasSaxpy(handle, n, a, d_x, 1, d_y, 1);

cublasGetVector(n, sizeof(y[0]), d_y, 1, y, 1);

cudaFree(d_x); cudaFree(d_y);
cublasDestroy(handle);
```

Initialize

Allocate GPU memory

cuBLAS

Code example

```
int a = 42;  int n = 10;
float x[n], y[n];
// fill x, y

cublasHandle_t handle;
cublasCreate(&handle);

float * d_x, * d_y;
cudaMallocManaged(&d_x, n * sizeof(x[0]));
cudaMallocManaged(&d_y, n * sizeof(y[0]));

cublasSaxpy(handle, n, a, d_x, 1, d_y, 1);

cublasGetVector(n, sizeof(y[0]), d_y, 1, y, 1);

cudaFree(d_x); cudaFree(d_y);
cublasDestroy(handle);
```

Initialize

Allocate GPU memory

cuBLAS

Code example

```
int a = 42;  int n = 10;
float x[n], y[n];
// fill x, y

cublasHandle_t handle;
cublasCreate(&handle);

float * d_x, * d_y;
cudaMallocManaged(&d_x, n * sizeof(x[0]));
cudaMallocManaged(&d_y, n * sizeof(y[0]));

cublasSaxpy(handle, n, a, d_x, 1, d_y, 1);

cublasGetVector(n, sizeof(y[0]), d_y, 1, y, 1);

cudaFree(d_x); cudaFree(d_y);
cublasDestroy(handle);
```

Initialize

Allocate GPU memory

Call BLAS routine

cuBLAS

Code example

```
int a = 42;  int n = 10;
float x[n], y[n];
// fill x, y

cublasHandle_t handle;
cublasCreate(&handle);
```

Initialize

```
float * d_x, * d_y;
cudaMallocManaged(&d_x, n * sizeof(x[0]));
cudaMallocManaged(&d_y, n * sizeof(y[0]));
```

Allocate GPU memory

```
cublasSaxpy(handle, n, a, d_x, 1, d_y, 1);
```

Call BLAS routine

```
cublasGetVector(n, sizeof(y[0]), d_y, 1, y, 1);
```

Copy result to host

```
cudaFree(d_x); cudaFree(d_y);
cublasDestroy(handle);
```

cuBLAS

Code example

```c
int a = 42;  int n = 10;
float x[n], y[n];
// fill x, y

cublasHandle_t handle;
cublasCreate(&handle);

float * d_x, * d_y;
cudaMallocManaged(&d_x, n * sizeof(x[0]));
cudaMallocManaged(&d_y, n * sizeof(y[0]));

cublasSaxpy(handle, n, a, d_x, 1, d_y, 1);

cublasGetVector(n, sizeof(y[0]), d_y, 1, y, 1);

cudaFree(d_x); cudaFree(d_y);
cublasDestroy(handle);
```

Initialize

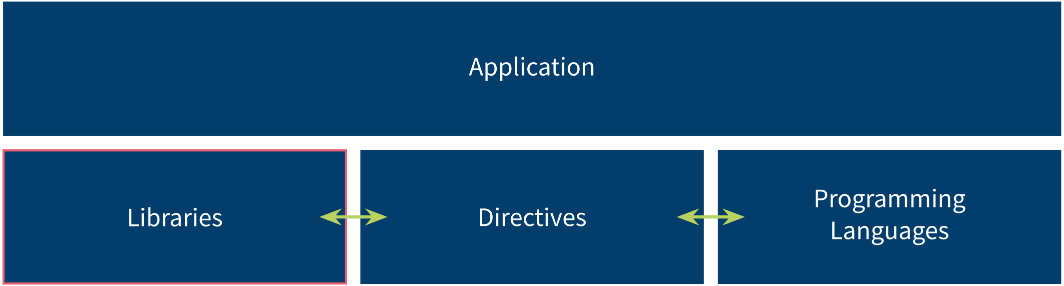
Allocate GPU memory

Call BLAS routine

Copy result to host

Finalize

Summary of Acceleration Possibilities

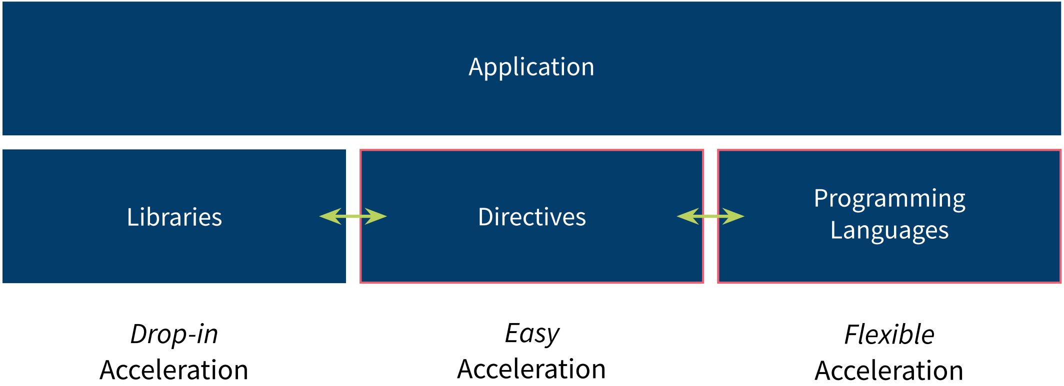


Drop-in
Acceleration

Easy
Acceleration

Flexible
Acceleration

Summary of Acceleration Possibilities



Application

Libraries

Directives

Programming Languages

Drop-in Acceleration

Easy Acceleration

Flexible Acceleration

⚠ Parallelism

Libraries are not enough?

You think you want to write your own GPU code?

Primer on Parallel Scaling

Amdahl's Law

Possible maximum speedup for
N parallel processors

Total Time $t = t_{\mathbf{s}\text{erial}} + t_{\mathbf{p}\text{arallel}}$

Primer on Parallel Scaling

Amdahl’s Law

Possible maximum speedup for
N parallel processors

Total Time $t = t_{\mathbf{s}\text{erial}} + t_{\mathbf{p}\text{arallel}}$

N Processors $t(N) = t_\text{s} + t_\text{p}/N$

Primer on Parallel Scaling

Amdahl's Law

Possible maximum speedup for
N parallel processors

Total Time $t = t_{\mathbf{s}\text{erial}} + t_{\mathbf{p}\text{arallel}}$

N Processors $t(N) = t_\mathrm{s} + t_\mathrm{p}/N$

Speedup $s(N) = t/t(N) = \frac{t_\mathrm{s}+t_\mathrm{p}}{t_\mathrm{s}+t_\mathrm{p}/N}$

Primer on Parallel Scaling

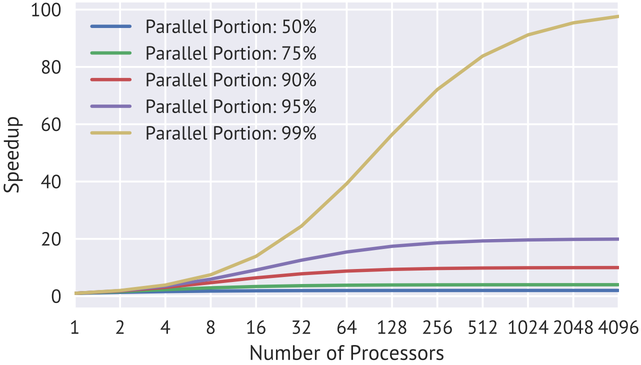
Amdahl's Law

Possible maximum speedup for
N parallel processors

Total Time $t = t_{\mathbf{serial}} + t_{\mathbf{parallel}}$

N Processors $t(N) = t_s + t_p/N$

Speedup $s(N) = t/t(N) = \frac{t_s + t_p}{t_s + t_p/N}$



Parallelism

Parallel programming is not easy!

Things to consider:

- Is my application **computationally intensive** *enough*?
- What are the levels of **parallelism**?
- How much **data** needs to be **transferred**?
- Is the **gain** worth the pain?

Alternatives

The twilight

There are GPU programming models, which **can** ease the *pain*…

- OpenACC, OpenMP
- Thrust
- Kokkos, RAJA, ALPAKA, SYCL, DPC++, pSTL
- PyCUDA, Cupy, Numba
- CUDA Fortran
- HIP, CUDA
- OpenCL

Programming GPUs

Directives

GPU Programming with Directives

Keepin’ you portable

- Annotate serial source code by directives

```
#pragma acc loop
for (int i = 0; i < 1; i++) {};
```

GPU Programming with Directives

Keepin’ you portable

- Annotate serial source code by directives

  ```
  #pragma acc loop
  for (int i = 0; i < 1; i++) {};
  ```

- **OpenACC**: Especially for GPUs; **OpenMP**: Has GPU support
- Compiler interprets directives, creates according instructions

GPU Programming with Directives

Keepin’ you portable

- Annotate serial source code by directives

```
#pragma acc loop
for (int i = 0; i < 1; i++) {};
```

- **OpenACC**: Especially for GPUs; **OpenMP**: Has GPU support
- Compiler interprets directives, creates according instructions

Pro

- Portability
  - Other compiler? No problem! To it, it’s a serial program
  - Different target architectures from same code
- Easy to program

Con

- Only few compilers
- Not all the raw power available
- A little harder to debug

GPU Programming with Directives

The power of… two.

OpenMP Standard for multithread programming on CPU, GPU since 4.0, better since 4.5

```
#pragma omp target map(tofrom:y), map(to:x)
#pragma omp teams num_teams(10) num_threads(10)
#pragma omp distribute
for (   ) {
    #pragma omp parallel for
    for (   ) {
    // …
    }
}
```

OpenACC Similar to OpenMP, but more specifically for GPUs

For C/C++ and Fortran

OpenACC

Code example

```
void saxpy_acc(int n, float a, float * x, float * y) {
  #pragma acc kernels
  for (int i = 0; i < n; i++)
    y[i] = a * x[i] + y[i];
}

int a = 42;
int n = 10;
float x[n], y[n];
// fill x, y

saxpy_acc(n, a, x, y);
```

OpenACC

Code example

```c
void saxpy_acc(int n, float a, float * x, float * y) {
  #pragma acc parallel loop copy(y) copyin(x)
  for (int i = 0; i < n; i++)
    y[i] = a * x[i] + y[i];
}

int a = 42;
int n = 10;
float x[n], y[n];
// fill x, y

saxpy_acc(n, a, x, y);
```

OpenACC / OpenMP

Code example

```
void saxpy_acc(int n, float a, float * x, float * y) {
  #pragma acc kernels
  for (int i = 0; i < n; i++)
    y[i] = a * x[i] + y[i];
}

float a = 42;
int n = 10;
float x[n], y[n];
// fill x, y

saxpy_acc(n, a, x, y);
```

OpenACC / OpenMP

Code example

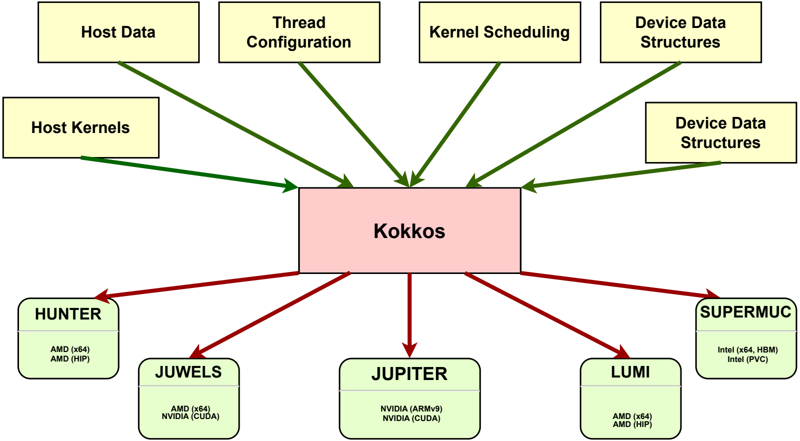
```
void saxpy_acc(int n, float a, float * x, float * y) {
  #pragma omp target map(to:x[0:n]) map(tofrom:y[0:n]) loop
  for (int i = 0; i < n; i++)
    y[i] = a * x[i] + y[i];
}

float a = 42;
int n = 10;
float x[n], y[n];
// fill x, y

saxpy_acc(n, a, x, y);
```

Programming GPUs

Kokkos

Performance Portability

Performant Single Source Implementation



History and Support

- Established 2012
- Widely used in HPC, especially US Exascale Computing Project ECP
- Support for most major HPC platforms

- Now moving into Linux Foundation
- Feedback loop with C++ Standards
  - Parallel STL
  - `std::atomic_ref`
  - `std::mdspan` and `std::mdarray`

Online Presence

- `https://github.com/kokkos`
  - Primary Github Organization
- `https://kokkosteam.slack.com`
  - Slack Channel for Kokkos

First Look

Hello, World!

```cpp
struct functor {
  __host__ __device__ void operator()(const int i) const {
    Kokkos::printf("Hello from i = %i\n", i);
}};

int main(int argc, char* argv[]) {
  Kokkos::initialize(argc, argv);
  Kokkos::parallel_for("HelloWorld", 8, functor());
  Kokkos::finalize();
}
```

Output

```
Hello from i = 0
Hello from i = 1
Hello from i = 2
Hello from i = 3
Hello from i = 4
Hello from i = 5
Hello from i = 6
Hello from i = 7
```

Programming GPUs

CUDA C/C++

Preface: CPU

A simple CPU program!

SAXPY: $\vec{y} = a\vec{x} + \vec{y}$, with single precision

Part of LAPACK BLAS Level 1

```c
void saxpy(int n, float a, float * x, float * y) {
  for (int i = 0; i < n; i++)
    y[i] = a * x[i] + y[i];
}

int a = 42;
int n = 10;
float x[n], y[n];
// fill x, y

saxpy(n, a, x, y);
```

CUDA SAXPY

With runtime-managed data transfers

```
__global__ void saxpy_cuda(int n, float a, float * x, float * y) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < n)
    y[i] = a * x[i] + y[i];
}

int a = 42;
int n = 10;
float x[n], y[n];
// fill x, y
cudaMallocManaged(&x, n * sizeof(float));
cudaMallocManaged(&y, n * sizeof(float));

saxpy_cuda<<<2, 5>>>(n, a, x, y);

cudaDeviceSynchronize();
```

CUDA SAXPY

With runtime-managed data transfers

```
__global__ void saxpy_cuda(int n, float a, float * x, float * y) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < n)
    y[i] = a * x[i] + y[i];
}

int a = 42;
int n = 10;
float x[n], y[n];
// fill x, y
cudaMallocManaged(&x, n * sizeof(float));
cudaMallocManaged(&y, n * sizeof(float));

saxpy_cuda<<<2, 5>>>(n, a, x, y);

cudaDeviceSynchronize();
```

Specify kernel

ID variables

Guard against
too many threads

Allocate GPU-capable
memory

Call kernel
2 blocks, each 5 threads

Wait for
kernel to finish

CUDA’s Parallel Model

In software: Threads, Blocks

- Methods to exploit parallelism:

CUDA’s Parallel Model

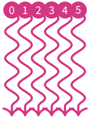
In software: Threads, Blocks

- Methods to exploit parallelism:
  - Thread

CUDA’s Parallel Model

In software: Threads, Blocks

- Methods to exploit parallelism:
  - Threads



CUDA's Parallel Model

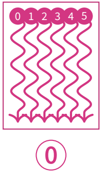
In software: Threads, Blocks

- Methods to exploit parallelism:
  - Threads → $\boxed{\text{Block}}$

CUDA's Parallel Model

In software: Threads, Blocks

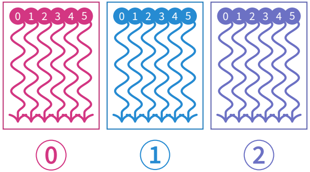
- Methods to exploit parallelism:
  - Threads → Block
  - Block



CUDA's Parallel Model

In software: Threads, Blocks

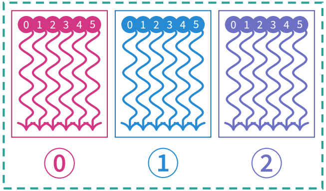
- Methods to exploit parallelism:
  - Threads → Block
  - Blocks



CUDA’s Parallel Model

In software: Threads, Blocks

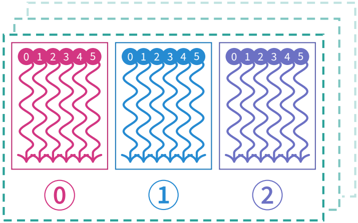
- Methods to exploit parallelism:
  - Threads → Block
  - Blocks → Grid



CUDA's Parallel Model

In software: Threads, Blocks

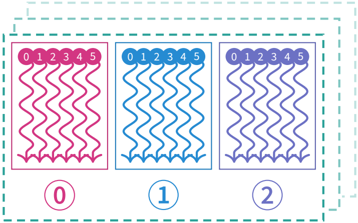
- Methods to exploit parallelism:
  - Threads → Block
  - Blocks → Grid
  - Threads & blocks in 3D



CUDA’s Parallel Model

In software: Threads, Blocks

- Methods to exploit parallelism:
  - Threads → Block
  - Blocks → Grid
  - Threads & blocks in 3D



- Parallel function: **kernel**
  - `__global__ kernel(int a, float * b) { }`
  - Access own ID by global variables `threadIdx.x`, `blockIdx.y`, …
- Execution entity: **threads**
  - Lightweight → fast switchting!
  - 1000s threads execute simultaneously → order non-deterministic!

Kernel Functions

- Kernel: Parallel GPU function
    - Executed by each thread
    - In parallel
    - Called from host or device

Kernel Functions

- Kernel: Parallel GPU function
  - Executed by each thread
  - In parallel
  - Called from host or device
- All threads execute same code; but can take different paths in program flow (some penalty)

Kernel Functions

- Kernel: Parallel GPU function
  - Executed by each thread
  - In parallel
  - Called from host or device
- All threads execute same code; but can take different paths in program flow (some penalty)
- Info about thread: local, global IDs

```
int currentThreadId = threadIdx.x;
float x = input[currentThreadId];
output[currentThreadId] = x*x;
```

Kernel Conversion

Recipe for C Function → CUDA Kernel

Identify Loops

```
void scale(float scale, float * in, float * out, int N) {
    for (int i = 0; i < N; i++)
        out[i] = scale * in[i];
}
```

Kernel Conversion

Recipe for C Function → CUDA Kernel

Identify Loops

```
void scale(float scale, float * in, float * out, int N) {
    for (
        int i = 0;
        i < N;
        i++
    )
        out[i] = scale * in[i];
}
```

Kernel Conversion

Recipe for C Function → CUDA Kernel

Identify Loops Extract Index

```
void scale(float scale, float * in, float * out, int N) {
    int i = 0;
    for ( ;
        i < N;
        i++
    )
        out[i] = scale * in[i];
}
```

Kernel Conversion

Recipe for C Function → CUDA Kernel

Identify Loops | Extract Index | Extract Termination Condition

```
void scale(float scale, float * in, float * out, int N) {
    int i = 0;
    for ( ;
          ;
        i++
    )
        if (i < N)
            out[i] = scale * in[i];
}
```

Kernel Conversion

Recipe for C Function → CUDA Kernel

Identify Loops | Extract Index | Extract Termination Condition | Remove `for`

```
void scale(float scale, float * in, float * out, int N) {
    int i = 0;




        if (i < N)
            out[i] = scale * in[i];
}
```

Kernel Conversion

Recipe for C Function → CUDA Kernel

Identify Loops | Extract Index | Extract Termination Condition | Remove `for` | Add `global`

```
__global__ void scale(float scale, float * in, float * out, int N) {
    int i = 0;




        if (i < N)
            out[i] = scale * in[i];
}
```

Kernel Conversion

Recipe for C Function → CUDA Kernel

Identify Loops | Extract Index | Extract Termination Condition | Remove `for` | Add `global`

Replace i by `threadIdx.x`

```
__global__ void scale(float scale, float * in, float * out, int N) {
    int i = threadIdx.x;




        if (i < N)
            out[i] = scale * in[i];
}
```

Kernel Conversion

Recipe for C Function → CUDA Kernel

Identify Loops | Extract Index | Extract Termination Condition | Remove `for` | Add `global`

Replace `i` by `threadIdx.x` | … including block configuration

```
__global__ void scale(float scale, float * in, float * out, int N) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;





        if (i < N)
            out[i] = scale * in[i];
}
```

Kernel Conversion

Summary

- C function with explicit loop

```
void scale(float scale, float * in, float * out, int N) {
    for (int i = 0; i < N; i++)
        out[i] = scale * in[i];
}
```

- CUDA kernel with implicit loop

```
__global__ void scale(float scale, float * in, float * out, int N) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i < N)
        out[i] = scale * in[i];
}
```

Kernel Launch

```
kernel<<<int gridDim, int blockDim>>>(...)
```

- Parallel threads of kernel launched with *triple-chevron syntax*
- Total number of threads, divided into
  - Number of blocks on the grid (`gridDim`)
  - Number of threads per block (`blockDim`)

Kernel Launch

```
kernel<<<int gridDim, int blockDim>>>(...)
```

- Parallel threads of kernel launched with *triple-chevron syntax*
- Total number of threads, divided into
    - Number of blocks on the grid (`gridDim`)
    - Number of threads per block (`blockDim`)

Kernel Launch

```
kernel<<<int gridDim, int blockDim>>>(...)
```

- Parallel threads of kernel launched with *triple-chevron syntax*
- Total number of threads, divided into
  - Number of blocks on the grid (`gridDim`)
    ×
  - Number of threads per block (`blockDim`)
- Call returns immediately; kernel launch is asynchronous!

Kernel Launch

```
kernel<<<int gridDim, int blockDim>>>(...)
```

- Parallel threads of kernel launched with *triple-chevron syntax*
- Total number of threads, divided into
  - Number of blocks on the grid (`gridDim`)
  - Number of threads per block (`blockDim`)
- Call returns immediately; kernel launch is asynchronous!
- Example:

  ```
  int nThreads = 32;
  scale<<<N/nThreads, nThreads>>>(23, in, out, N)
  ```

Kernel Launch

```
kernel<<<int gridDim, int blockDim>>>(...)
```

- Parallel threads of kernel launched with *triple-chevron syntax*
- Total number of threads, divided into
  - Number of blocks on the grid (`gridDim`)
  - Number of threads per block (`blockDim`)
- Call returns immediately; kernel launch is asynchronous!
- Example:

```
int nThreads = 32;
scale<<<N/nThreads, nThreads>>>(23, in, out, N)
```

- Possibility for too many threads; include termination condition into kernel!

Full Kernel Launch

For Reference

```
kernel<<<dim3 gD, dim3 bD, size_t shared, cudaStream_t stream>>>(...)
```

- 2 additional, optional parameters

Full Kernel Launch

For Reference

```
kernel<<<dim3 gD, dim3 bD, size_t shared, cudaStream_t stream>>>(...)
```

- 2 additional, optional parameters

`shared` Dynamic **shared memory**

  - Small GPU memory space; share data in block (high bandwidth)
  - Shared memory: allocate statically (compile time) or dynamically (run time)
  - `size_t shared`: bytes of shared memory allocated per block (in addition to static shared memory)

Full Kernel Launch

For Reference

```
kernel<<<dim3 gD, dim3 bD, size_t shared, cudaStream_t stream>>>(...)
```

- 2 additional, optional parameters

`shared` Dynamic **shared memory**

- Small GPU memory space; share data in block (high bandwidth)
- Shared memory: allocate statically (compile time) or dynamically (run time)
- `size_t shared`: bytes of shared memory allocated per block (in addition to static shared memory)

`stream` Associated **CUDA stream**

- CUDA streams enable different channels of communication with GPU
- Can overlap in some cases (communication, computation)
- `cudaStream_t stream`: ID of stream to use for this kernel launch

Kernel Conversion

Recipe for C Function → CUDA Kernel

Identify Loops

```
void scale(float scale, float * in, float * out, int N) {
    for (int i = 0; i < N; i++)
        out[i] = scale * in[i];
}
```

Kernel Conversion

Recipe for C Function → CUDA Kernel

Identify Loops

```
void scale(float scale, float * in, float * out, int N) {
    for (
        int i = 0;
        i < N;
        i++
    )
        out[i] = scale * in[i];
}
```

Kernel Conversion

Recipe for C Function → CUDA Kernel

Identify Loops Extract Index

```
void scale(float scale, float * in, float * out, int N) {
    int i = 0;
    for ( ;
        i < N;
        i++
    )
        out[i] = scale * in[i];
}
```

Kernel Conversion

Recipe for C Function → CUDA Kernel

Identify Loops Extract Index Extract Termination Condition

```
void scale(float scale, float * in, float * out, int N) {
    int i = 0;
    for ( ;
          ;
        i++
    )
        if (i < N)
            out[i] = scale * in[i];
}
```

Kernel Conversion

Recipe for C Function → CUDA Kernel

Identify Loops Extract Index Extract Termination Condition Remove `for`

```
void scale(float scale, float * in, float * out, int N) {
    int i = 0;




        if (i < N)
            out[i] = scale * in[i];
}
```

Kernel Conversion

Recipe for C Function → CUDA Kernel

Identify Loops | Extract Index | Extract Termination Condition | Remove `for` | Add `global`

```
__global__ void scale(float scale, float * in, float * out, int N) {
    int i = 0;




        if (i < N)
            out[i] = scale * in[i];
}
```

Kernel Conversion

Recipe for C Function → CUDA Kernel

Identify Loops Extract Index Extract Termination Condition Remove `for` Add `global`

Replace i by `threadIdx.x`

```
__global__ void scale(float scale, float * in, float * out, int N) {
    int i = threadIdx.x;




        if (i < N)
            out[i] = scale * in[i];
}
```

Kernel Conversion

Recipe for C Function → CUDA Kernel

Identify Loops | Extract Index | Extract Termination Condition | Remove `for` | Add `global`

Replace i by `threadIdx.x` | … including block configuration

```
__global__ void scale(float scale, float * in, float * out, int N) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;




        if (i < N)
            out[i] = scale * in[i];
}
```

Kernel Conversion

Summary

- C function with explicit loop

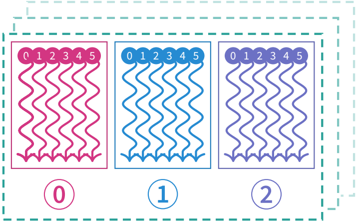
```
void scale(float scale, float * in, float * out, int N) {
    for (int i = 0; i < N; i++)
        out[i] = scale * in[i];
}
```

- CUDA kernel with implicit loop

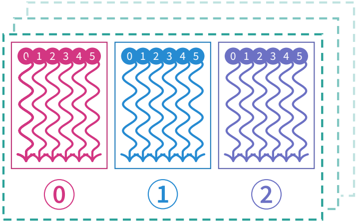
```
__global__ void scale(float scale, float * in, float * out, int N) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    if (i < N)
        out[i] = scale * in[i];
}
```

Grid Dimensions

- Threads & blocks in 3D



Grid Dimensions

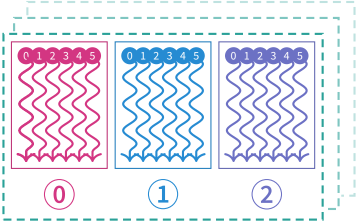


- Threads & blocks in 3D
- Create 3D configurations with `struct dim3`

```
dim3 blockOrGridDim(size_t dimX, size_t dimY, size_t dimZ)
```

Any unspecified component initialized to 1

Grid Dimensions



- Threads & blocks in 3D
- Create 3D configurations with `struct dim3`

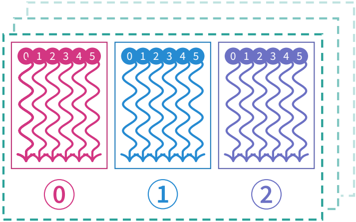
```
dim3 blockOrGridDim(size_t dimX, size_t dimY, size_t dimZ)
```

Any unspecified component initialized to 1

- Example:

```
dim3 blockDim(32, 32);
dim3 gridDim = {1000, 100};
```

Grid Dimensions



- Threads & blocks in 3D
- Create 3D configurations with `struct dim3`

```
dim3 blockOrGridDim(size_t dimX, size_t dimY, size_t dimZ)
```

Any unspecified component initialized to 1

- Example:

```
dim3 blockDim(32, 32);
dim3 gridDim = {1000, 100};
```

- Kernel call with `dim3`

```
kernel<<<dim3 gridDim, dim3 blockDim>>>(...)
```

Grid Sizes

- Block and grid sizes are hardware-dependent

Grid Sizes

- Block and grid sizes are hardware-dependent
- For JSC GPUs: Tesla V100, A100, H100

Block
  - $\vec{N}_{\mathrm{Thread}} \leq (1024_{\mathrm{x}}, 1024_{\mathrm{y}}, 64_{\mathrm{z}})$
  - $|\vec{N}_{\mathrm{Thread}}| = N_{\mathrm{Thread}} \leq 1024$

Grid Sizes

- Block and grid sizes are hardware-dependent
- For JSC GPUs: Tesla V100, A100, H100

Block
- $\vec{N}_{\text{Thread}} \leq (1024_{\text{x}}, 1024_{\text{y}}, 64_{\text{z}})$
- $|\vec{N}_{\text{Thread}}| = N_{\text{Thread}} \leq 1024$

Grid
- $\vec{N}_{\text{Blocks}} \leq (2147483647_{\text{x}}, 65535_{\text{y}}, 65535_{\text{z}}) = (2^{31}, 2^{16}, 2^{16}) - \vec{1}$

Grid Sizes

- Block and grid sizes are hardware-dependent
- For JSC GPUs: Tesla V100, A100, H100
  - Block
    - $\vec{N}_{\text{Thread}} \leq (1024_{\text{x}}, 1024_{\text{y}}, 64_{\text{z}})$
    - $|\vec{N}_{\text{Thread}}| = N_{\text{Thread}} \leq 1024$
  - Grid
    - $\vec{N}_{\text{Blocks}} \leq (2147483647_{\text{x}}, 65535_{\text{y}}, 65535_{\text{z}}) = (2^{31}, 2^{16}, 2^{16}) - \vec{1}$
- Find out yourself: `deviceQuery` example from CUDA Samples

Grid Sizes

- Block and grid sizes are hardware-dependent
- For JSC GPUs: Tesla V100, A100, H100

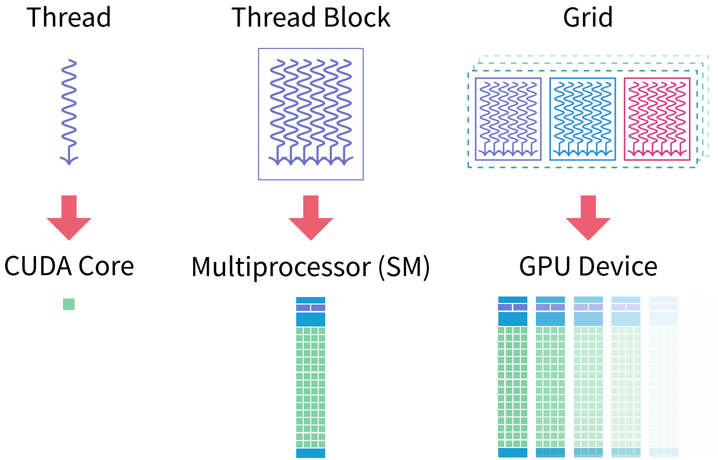
  Block
    - $\vec{N}_{\text{Thread}} \leq (1024_{\text{x}}, 1024_{\text{y}}, 64_{\text{z}})$
    - $|\vec{N}_{\text{Thread}}| = N_{\text{Thread}} \leq 1024$

  Grid
    - $\vec{N}_{\text{Blocks}} \leq (2147483647_{\text{x}}, 65535_{\text{y}}, 65535_{\text{z}}) = (2^{31}, 2^{16}, 2^{16}) - \vec{1}$
- Find out yourself: `deviceQuery` example from CUDA Samples
- Workflow: Chose 128 or 256 as block dim; calculate grid dim from problem size

```
int Nx = 1000, Ny = 1000;
dim3 blockDim(16, 16);
int gx = (Nx % blockDim.x == 0)  Nx / blockDim.x : Nx / blockDim.x + 1;
int gy = (Ny % blockDim.y == 0)  Ny / blockDim.y : Ny / blockDim.y + 1;
dim3 gridDim(gx, gy);
kernel<<<gridDim, blockDim>>>();
```

Hardware Threads

Mapping Software Threads to Hardware



Thread → CUDA Core

Thread Block → Multiprocessor (SM)

Grid → GPU Device

GPU Memory

- Data needs to reach the GPU; many ways to do so
- Progression

`cudaMalloc()`	First: Manual transfers via dedicated API
`cudaMallocManaged()`	Then: Automated transfers via dedicated API
`malloc()`	Now: Automated transfers via usual API

- `malloc()` has some caveats (system support) → *Full CUDA Unified Memory Support*

→ CUDA documentation *Unified Memory Programming*

Memory Management

With Automated Transfers

- Allocate memory to be used on GPU or CPU

```
cudaMallocManaged(T** ptr, size_t nBytes)
```

- Data is copied to GPU or to CPU automatically *(managed)*

Memory Management

With Automated Transfers

- Allocate memory to be used on GPU or CPU

```
cudaMallocManaged(T** ptr, size_t nBytes)
```

- Data is copied to GPU or to CPU automatically *(managed)*
- Example:

```
float * a;
int N = 2048;
cudaMallocManaged(&a, N * sizeof(float));
```

Memory Management

With Automated Transfers

- Allocate memory to be used on GPU or CPU

```
cudaMallocManaged(T** ptr, size_t nBytes)
```

- Data is copied to GPU or to CPU automatically *(managed)*
- Example:

```
float * a;
int N = 2048;
cudaMallocManaged(&a, N * sizeof(float));
```

- Free device memory

```
cudaFree(void* ptr)
```

Memory Management

With Manual Transfers

- Allocate memory to be used on GPU

```
cudaMalloc(T** ptr, size_t nBytes)
```

Memory Management

With Manual Transfers

- Allocate memory to be used on GPU

```
cudaMalloc(T** ptr, size_t nBytes)
```

- Copy data between host ↔ device

```
cudaMemcpy(void* dst, void* src, size_t nByte, enum cudaMemcpyKind dir)
```

Memory Management

With Manual Transfers

- Allocate memory to be used on GPU

```
cudaMalloc(T** ptr, size_t nBytes)
```

- Copy data between host ↔ device

```
cudaMemcpy(void* dst, void* src, size_t nByte, enum cudaMemcpyKind dir)
```

- Example:

```
float * a, * a_d;
int N = 2048;
// fill a
cudaMalloc(&a_d, N * sizeof(float));
cudaMemcpy(a_d, a,   N * sizeof(float), cudaMemcpyHostToDevice);
kernel<<<1,1>>>(a_d, N);
cudaMemcpy(a   , a_d, N * sizeof(float), cudaMemcpyDeviceToHost);
```

Manual Memory vs. Unified Memory

```
void sortfile(FILE *fp, int N) {
    char *data;
    char *data_d;

    data = (char *)malloc(N);
    cudaMalloc(&data_d, N);

    fread(data, 1, N, fp);

    cudaMemcpy(data_d, data, N, cudaMemcpyHostToDevice);
    kernel<<<...>>>(data, N);

    cudaMemcpy(data, data_d, N, cudaMemcpyDeviceToHost);
    host_func(data)
    cudaFree(data_d); free(data);
}
```

```
void sortfile(FILE *fp, int N) {
    char *data;


    cudaMallocManaged(&data, N);


    fread(data, 1, N, fp);


    kernel<<<...>>>(data, N);
    cudaDeviceSynchronize();

    host_func(data);
    cudaFree(data);
}
```

Exercises

Exercises

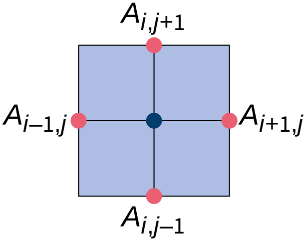
Task

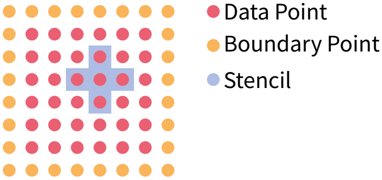
- Open fresh shell for today (reservation)
- Call `jsc-material-sync` (pull in recent changes)
- See `$HOME/natESM/GPU-Course/CUDA`
- Read instructions!
- Solutions given; you tinker as long as you want, then ask or check solutions
- Timeline
  - CUDA until coffee break; solutions after break
  - OpenACC until lunch, solutions before/after?
  - Kokkos in afternoon

Jacobi Solver

Algorithmic description

- Example for acceleration: **Jacobi solver**
- Iterative solver, converges to correct value
- Each iteration step: compute average of neighboring points
- Example: 2D Poisson equation: $\nabla^2 A(x,y) = B(x,y)$





$$A_{k+1}(i,j) = -\frac{1}{4}\left(B(i,j) - (A_k(i-1,j) + A_k(i,j+1), + A_k(i+1,j) + A_k(i,j-1))\right)$$

GPU Programming

- Many ways of doing it!
- **CUDA**: Native programming model
- **OpenACC**: High-level abstraction, with some portability; simple
- **Kokkos**: Dedicated programming model, performance-portability, C++
- Pick your poison!

Appendix

Glossary I

CUDA	Computing platform for GPUs from NVIDIA. Provides, among others, CUDA C/C++. 2, 48, 62, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 92, 93, 94, 108, 109, 110, 111, 112, 128
NVIDIA	US technology company creating GPUs. 18, 19, 20, 128, 129, 130
NVLink	NVIDIA's communication protocol connecting CPU ↔ GPU and GPU ↔ GPU with high bandwidth. 130
OpenACC	Directive-based programming, primarily for many-core machines. 48, 50, 51, 52, 53, 54, 55, 56, 57
OpenCL	The *Open Computing Language*. Framework for writing code for heterogeneous architectures (CPU, GPU, DSP, FPGA). The alternative to CUDA. 48

Glossary II

OpenMP	Directive-based programming, primarily for multi-threaded machines. 48, 50, 51, 52, 53, 56, 57
SAXPY	Single-precision $A \times X + Y$. A simple code example of scaling a vector and adding an offset. 63, 64, 65
Tesla	The GPU product line for general purpose computing computing of NVIDIA. 108, 109, 110, 111, 112
Thrust	A parallel algorithms library for (among others) GPUs. See `https://thrust.github.io/`. 48

Glossary III

V100	A large GPU with the Volta architecture from NVIDIA. It employs NVLink 2 as its interconnect and has fast *HBM2* memory. Additionally, it features *Tensorcores* for Deep Learning and Independent Thread Scheduling. 108, 109, 110, 111, 112
Volta	GPU architecture from NVIDIA (announced 2017). 130
CPU	Central Processing Unit. 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 53, 63, 115, 116, 117, 128
GPU	Graphics Processing Unit. 2, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 25, 28, 29, 30, 31, 32, 42, 49, 50, 51, 52, 53, 58, 62, 65, 75, 76, 77, 92, 93, 94, 108, 109, 110, 111, 112, 115, 116, 117, 118, 119, 120, 128, 129, 130
SIMD	Single Instruction, Multiple Data. 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

Glossary IV

SIMT	Single Instruction, Multiple Threads. 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
SM	Streaming Multiprocessor. 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
SMT	Simultaneous Multithreading. 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

References I

References: Images, Graphics I