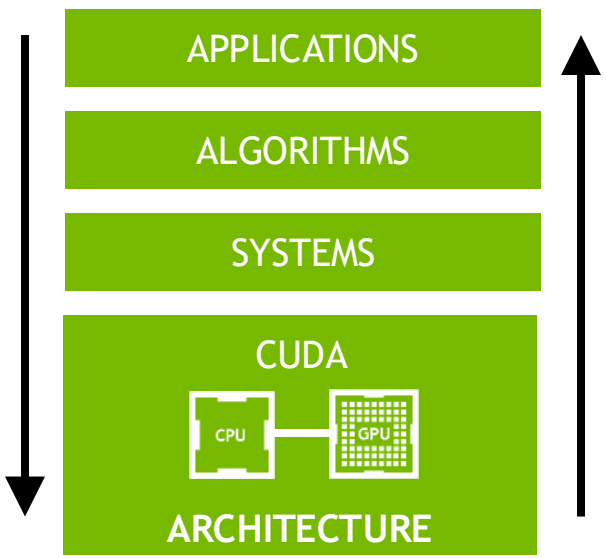
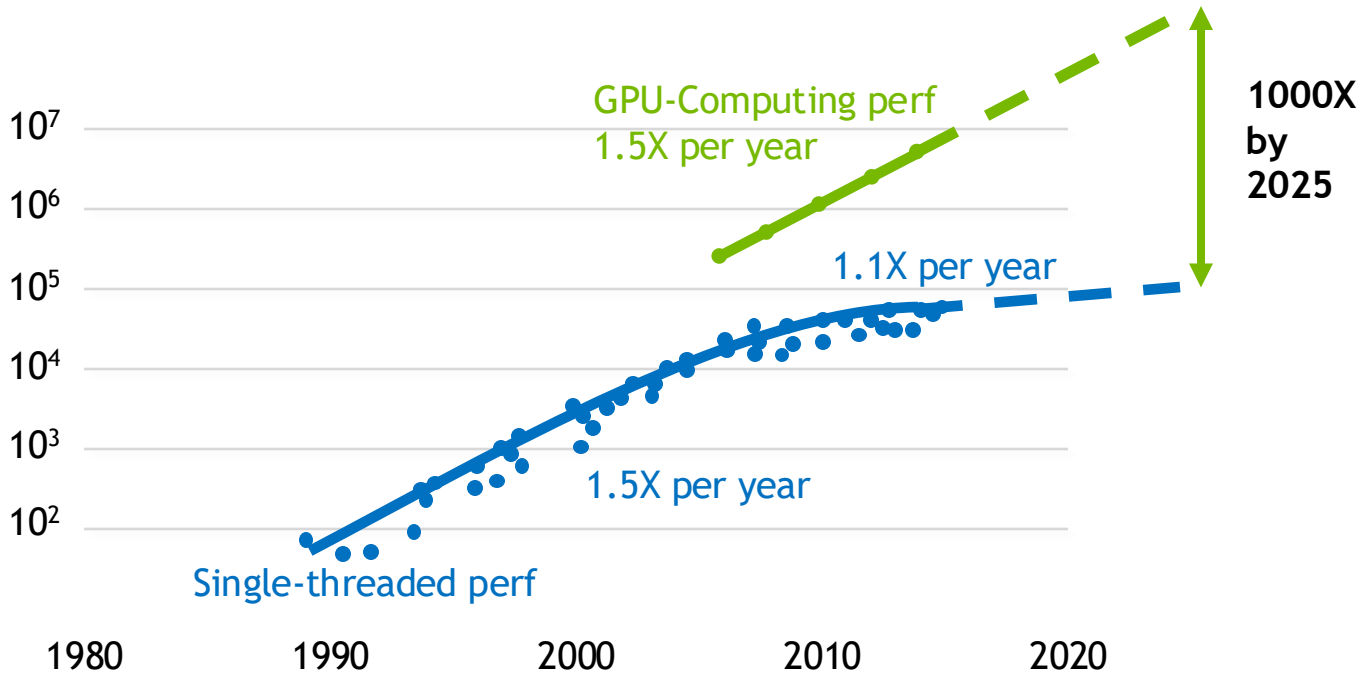
GPU COMPUTING
1 - PARALLELIZE

MATHIAS WAGNER, LATTICE PRACTICES 2024

RISE OF GPU COMPUTING

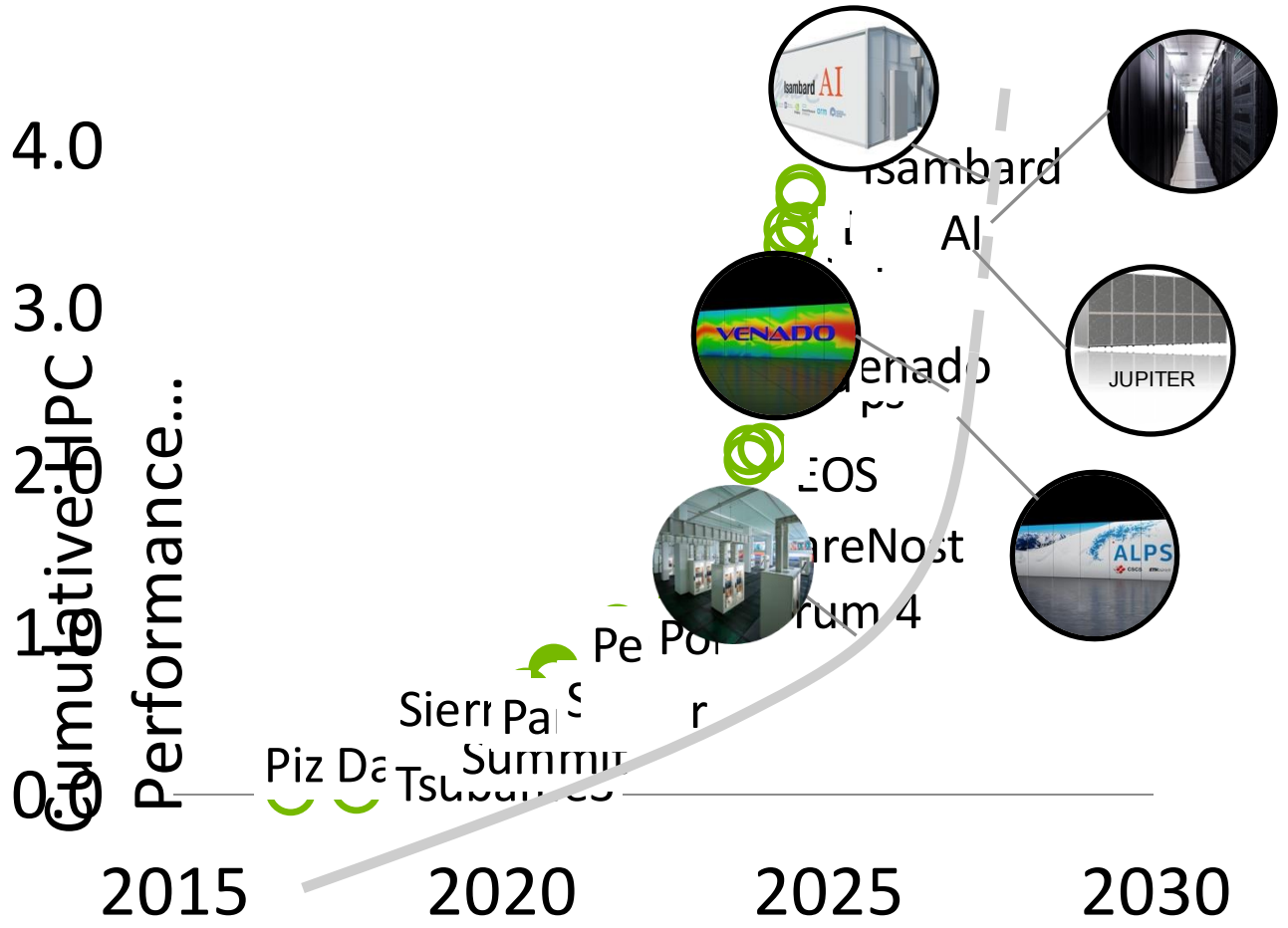


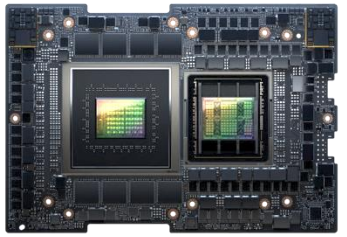


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten New plot and data collected for 2010-2015 by K. Rupp

Next-Gen Supercomputing Datacenter

4 ExaFLOPs of HPC Performance Driving Scientific Innovation

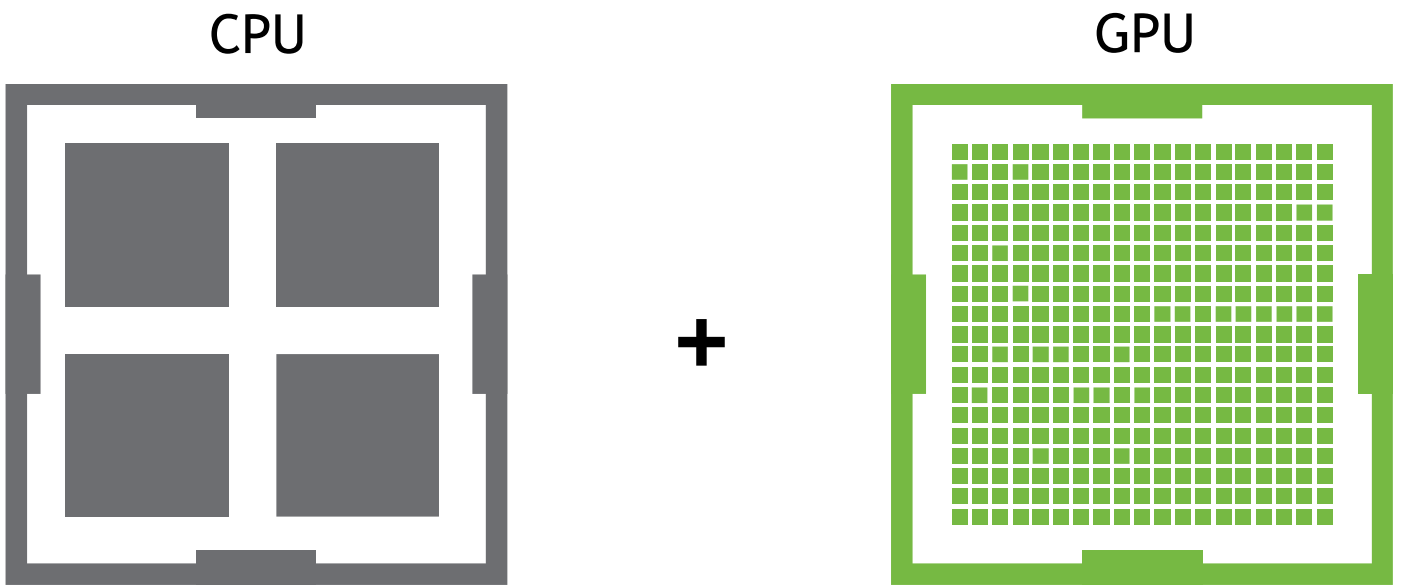




1.7 Exaflops Grace Hopper
Coming online 2024

ADD GPUS

Accelerate science applications



ADD GPUS

Accelerate science applications

CPU

CPU Strengths

- Very large main memory
- Very fast clock speeds
- Latency optimized via large caches
- Small number of threads can run very quickly

CPU Weaknesses

- Relatively low memory bandwidth
- Cache misses very costly
- Low performance per watt

ADD GPUS

Accelerate science applications

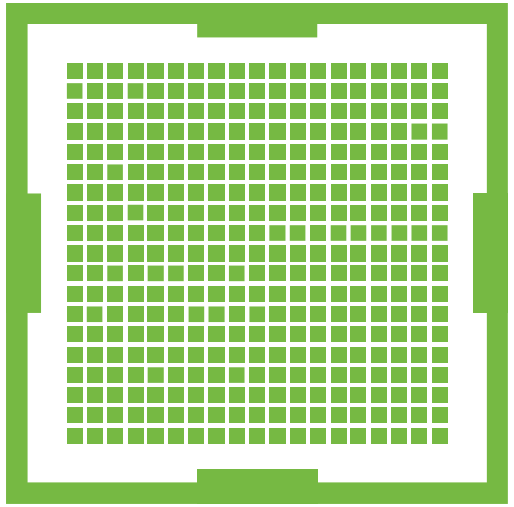
GPU Strengths

- High bandwidth main memory
- Latency tolerant via parallelism
- Significantly more compute resources
- High throughput
- High performance per watt

GPU Weaknesses

- Relatively low memory capacity
- Low per-thread performance

GPU



SPEED VS THROUGHPUT

Speed

Throughput

Choose the right tool for the job

*Images from Wikimedia Commons via Creative Commons

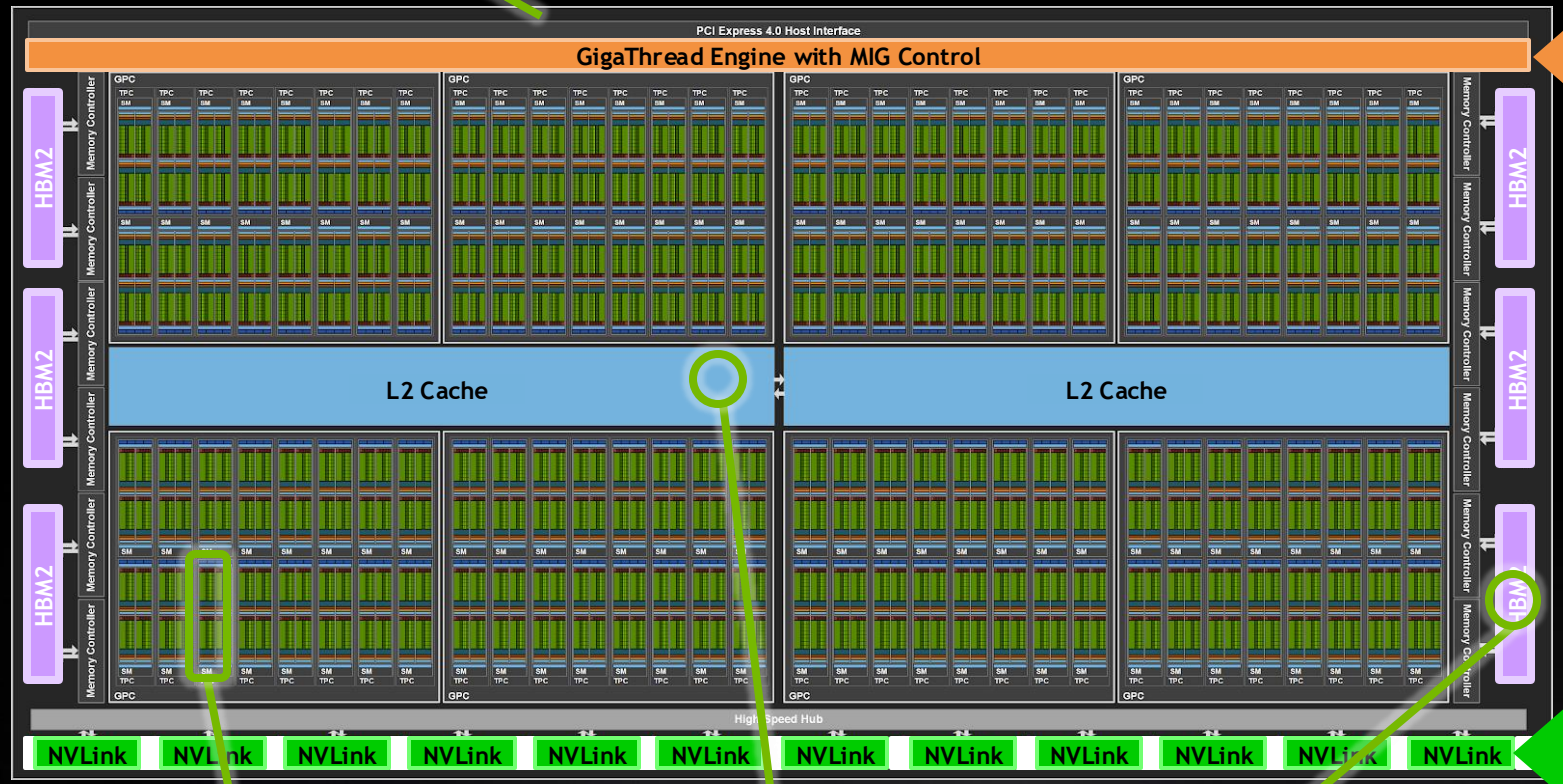
A100 TENSOR CORE GPU

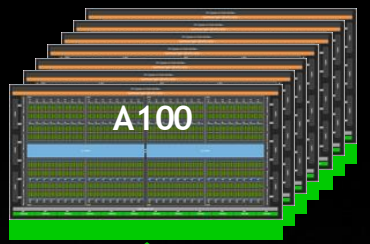
54B XTOR | 826mm2 | TSMC 7N

PCIGen4
with SR-IOV
31.5 GB/s

7x

Multi-Instance GPU





2x BW

3rd Gen NVLINK

108 SMs
6912 CUDA Cores

40MB L2
6.7x capacity

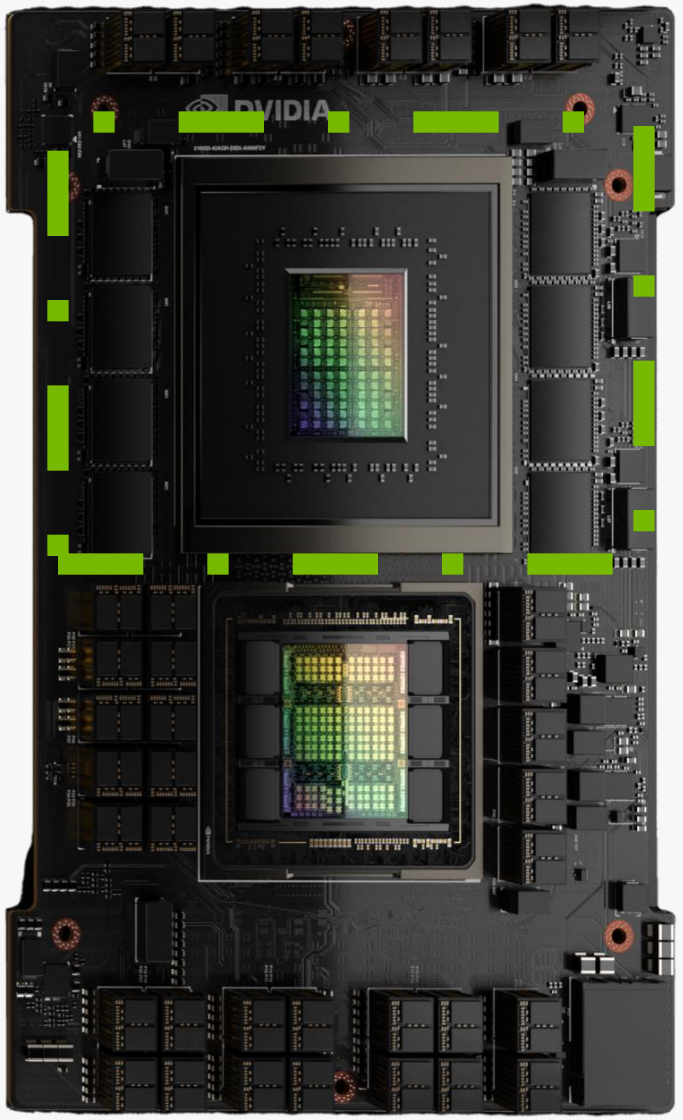
40 GB HBM2
1.6 TB/s HBM2

NVIDIA

A100 SM

- 32 FP64 CUDA Cores/SM (3456 in total per GPU)
- 64 FP32 CUDA Cores/SM (6912 in total per GPU)
- 64 INT32 CUDA Cores/SM (6912 in total per GPU)
- 4 3rd Tensor Cores/SM (432 in total per GPU)
  - Faster and more efficient
  - Comprehensive data types
  - Sparsity acceleration
- Asynchronous data movement and synchronization
- Increased L1/SMEM capacity

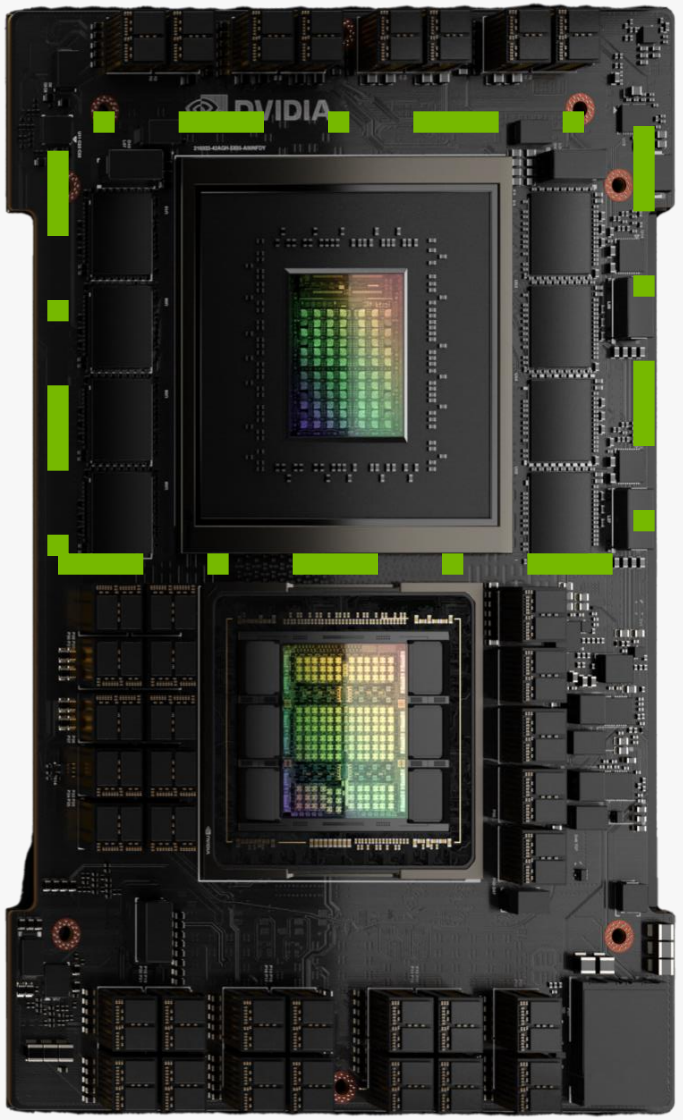
NVIDIA.



NVIDIA Grace Hopper Superchip

"super" - more than a "chip"

NVIDIA CPU + NVIDIA GPU w/o compromises



NVIDIA Grace Hopper Superchip

"super" - more than a "chip"

NVIDIA CPU + NVIDIA GPU w/o compromises

- **NVIDIA Grace CPU**
  - 72 Arm-v9 Neoverse V2 CPU cores with SVE2.
    - → Throughput: 3.6 TFLOP/s
  - Memory:
    - → High capacity: ≤ 480 GB LPDDR5X
    - → High System Memory bandwidth: ≤ 500 GB/s

NVIDIA Grace Hopper Superchip

"super" - more than a "chip"

NVIDIA CPU + NVIDIA GPU w/o compromises

- **NVIDIA Grace CPU**
  - 72 Arm-v9 Neoverse V2 CPU cores with SVE2.
    - → Throughput: 3.6 TFLOP/s
  - Memory:
    - → High capacity: ≤ 480 GB LPDDR5X
    - → High System Memory bandwidth: ≤ 500 GB/s

- **NVIDIA Hopper GPU**
  - → High throughput: 60 TFLOP/s
  - Memory:
    - → Capacity: 96 GB HBM3 / 144 GB HBM3e
    - → Extreme bandwidth ≤ 4000 GB/s / 5000 GB/s
  - ≤ 18x NVLink 4 → 900 GB/s
  - → Threads are threads

NVIDIA Grace Hopper Superchip

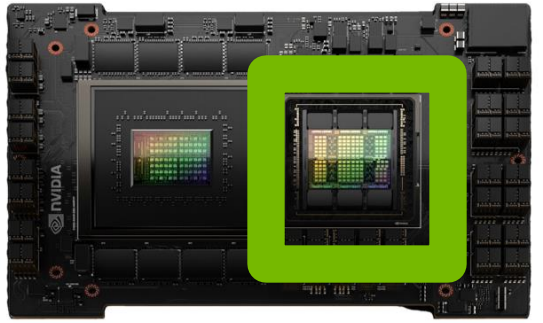
Soul is the new **NVLink-C2C** CPU ←→ GPU interconnect

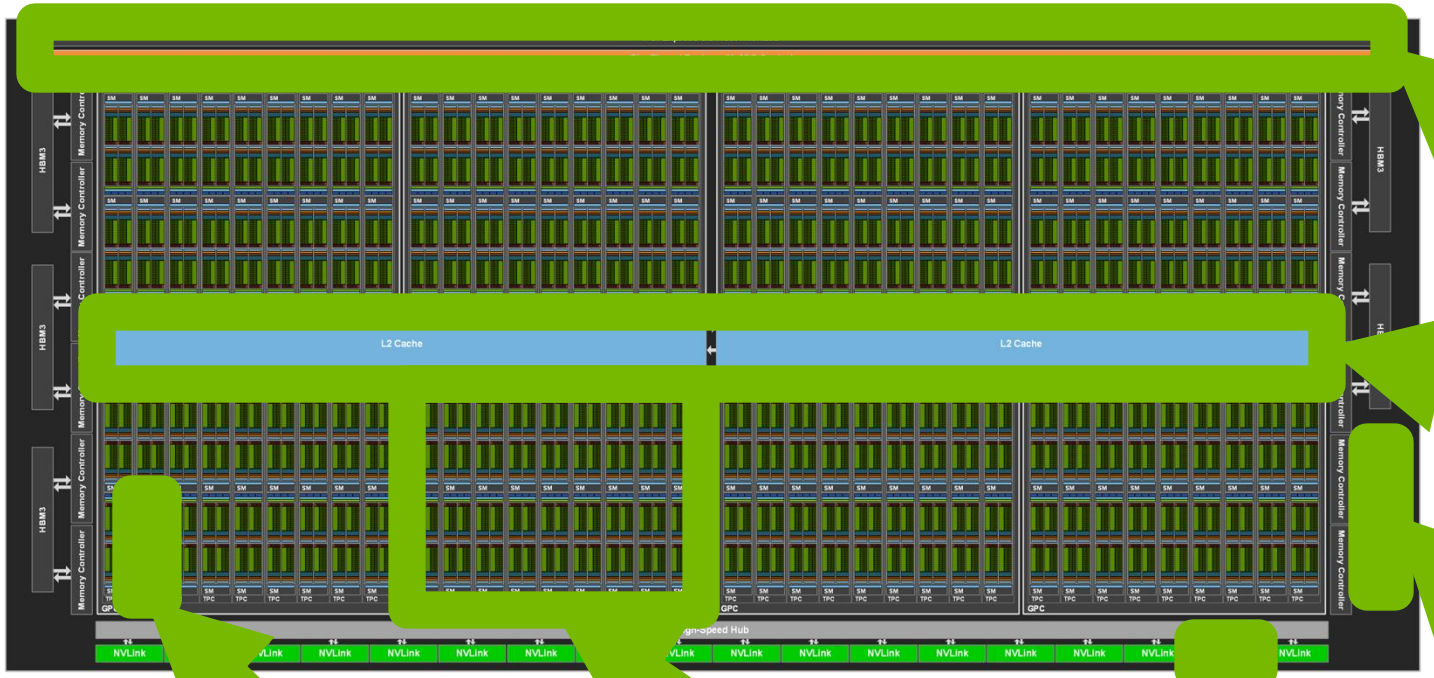
- **Memory consistency**: ease of use
  - →*All* threads – GPU and CPU – access system memory: C++ `new`, `malloc`, `mmap`'ed files, atomics, …
  - → Fast automatic page migrations
  - → Threads cache peer memory → Less migrations
- **High-bandwidth**: 900 GB/s (same as peer NVLink 4)
  - → GPU reads or writes local/peer LPDDR5X at ~peak BW
- **Low-latency**: GPU→HBM latency
  - →GPU reads or writes LPDDR5X at ~HBM3 latency

For all threads in the system
memory tastes like memory
expected behavior + latency + bandwidth.

Hopper Architecture

H100 GPU Key features





2nd Gen Multi-Instance GPU
Confidential Computing
PCIe Gen5

Larger 60 MB L2

96GB HBM3, 4 TB/s bandwidth

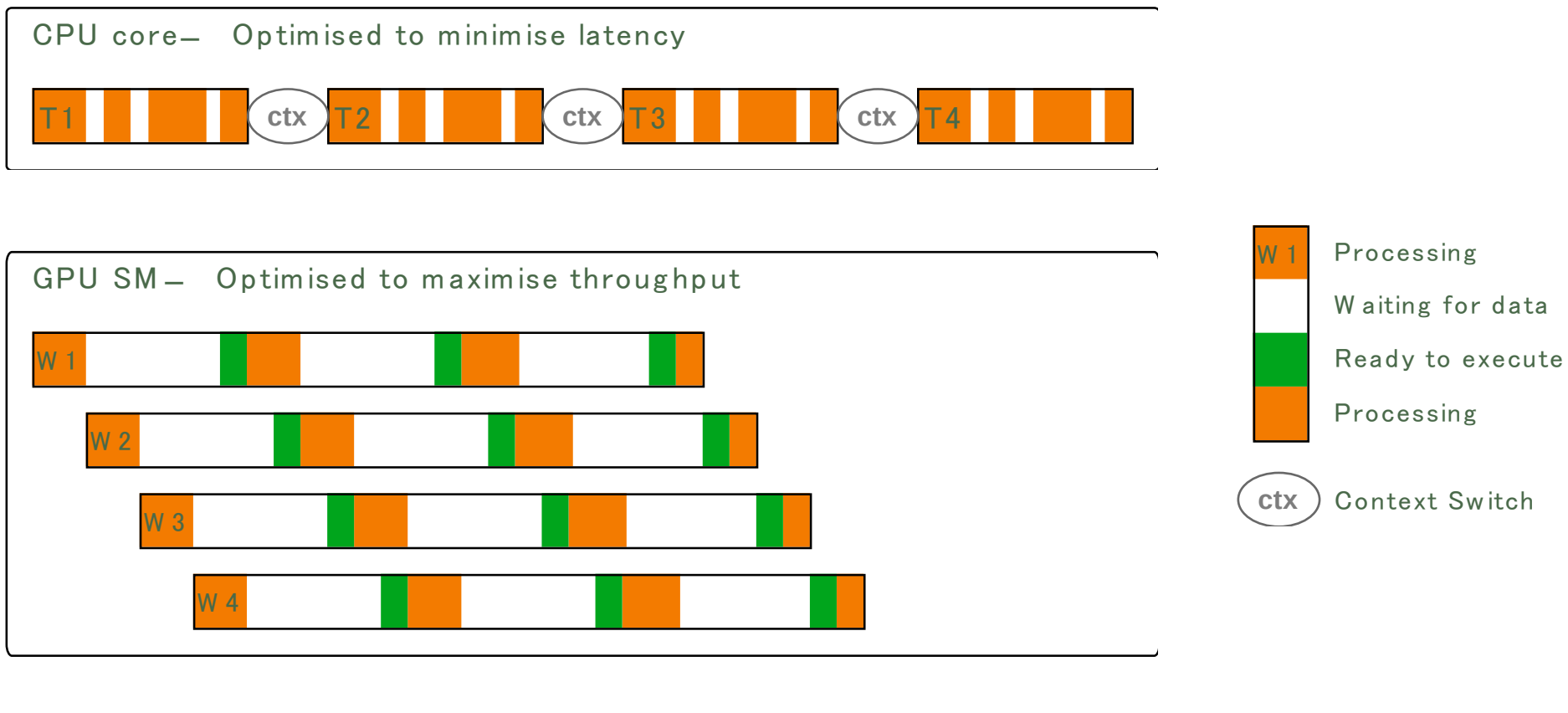
132 SMs
4th Gen Tensor Core

Thread Block Clusters

4th Gen NVLink
900 GB/s total bandwidth

CUDA Programming Model for Hopper Architecture
Optimizing Applications for Hopper Architecture

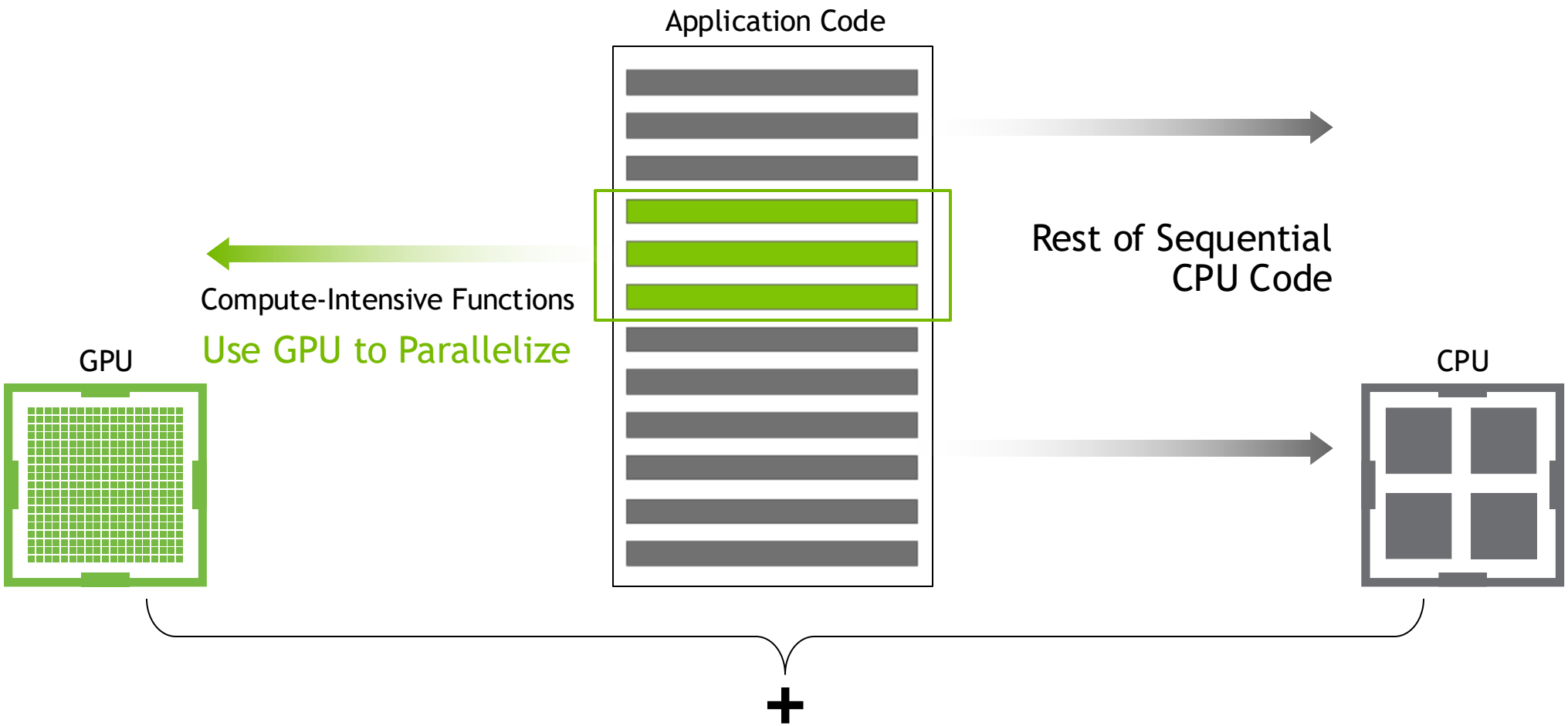
THROUGHPUT COMPUTING

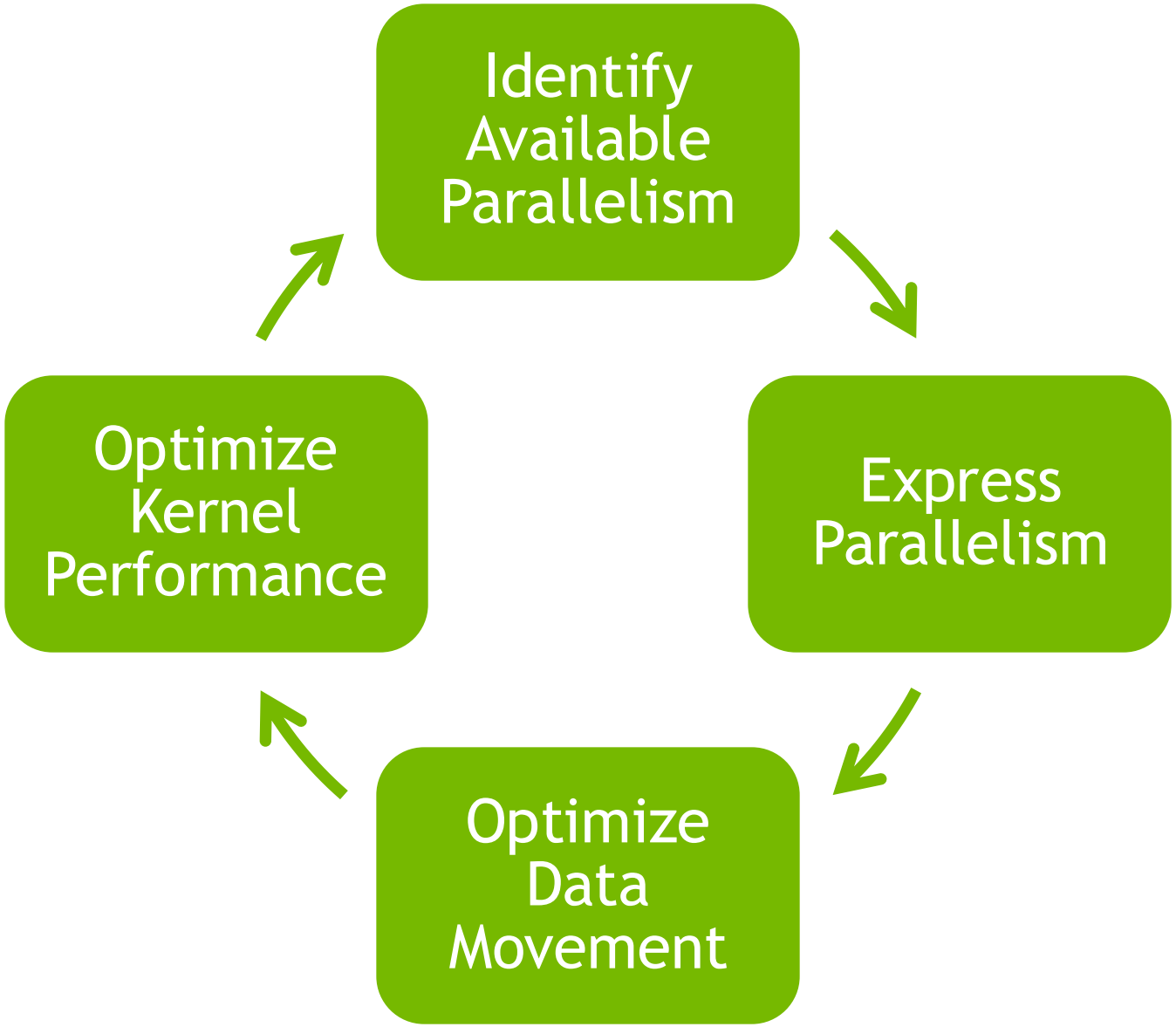


NVIDIA.

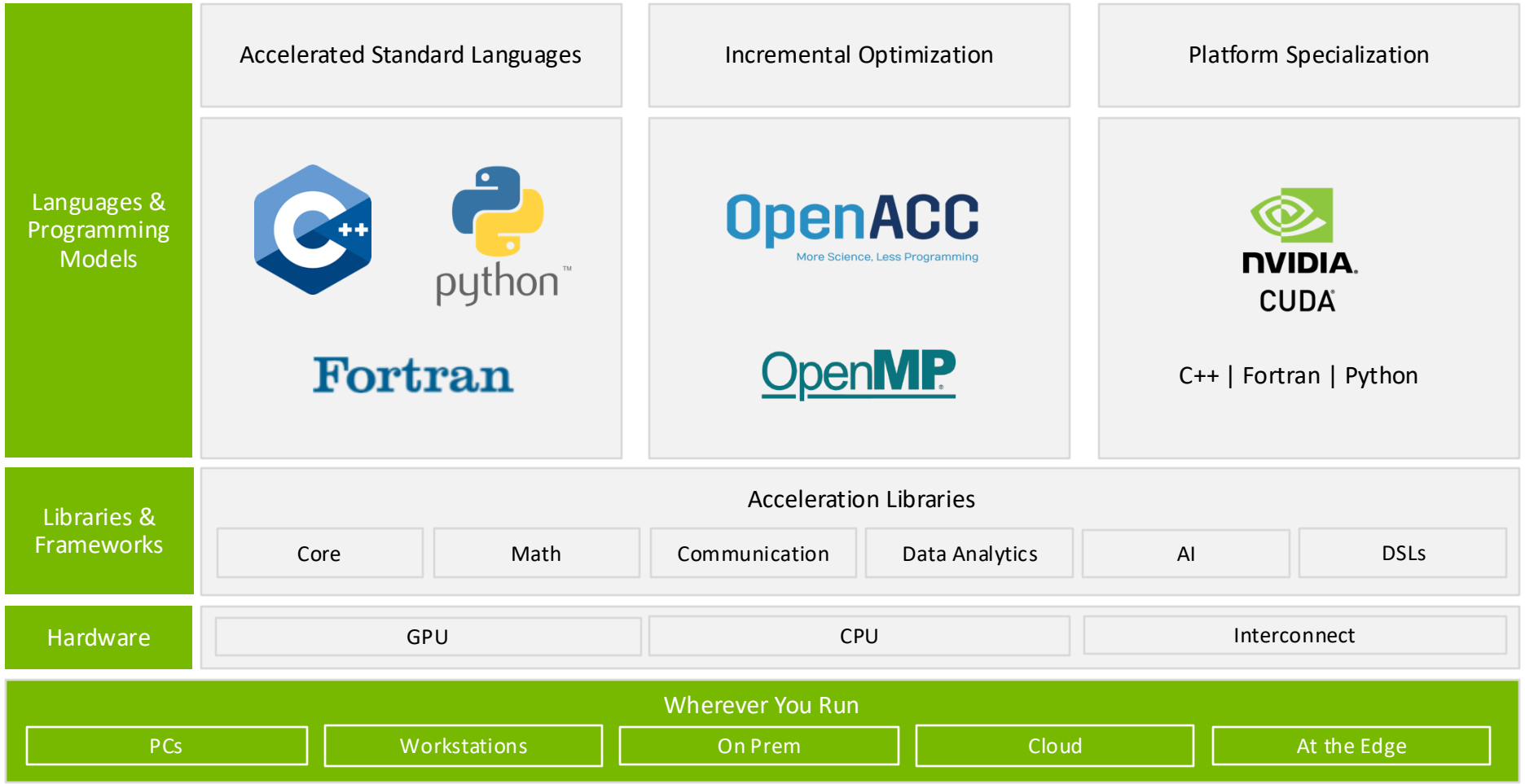
EXPRESSING PARALLELISM

SMALL CHANGES, BIG SPEED-UP




Identify Available Parallelism
Express Parallelism
Optimize Data Movement
Optimize Kernel Performance

Programming the NVIDIA Platform



Choosing A Programming Model

There can be ~~only~~ *more than* one.

Libraries	Standard Languages	Compiler Directives	CUDA Languages
• Accelerate common operations with little/no code changes. • Expert-tuned performance. • Forward support guarantees.	• Strong cross-platform support. • Single source code for multiple platforms. • Reduced learning curve.	• High cross-platform support. • Single source code for multiple platforms. • Reduced learning curve. • Additional programmer control.	• Exposes full GPU capabilities. • Trades portability for performance. • Distinct GPU/CPU code paths. • Full programmer control.
Programmer Productivity			Programmer Control

Approaches are interoperable.

NVIDIA

LIBRARIES: EASY, HIGH-QUALITY ACCELERATION

EASE OF USE	Using libraries enables GPU acceleration without in-depth knowledge of GPU programming
"DROP-IN"	Many GPU-accelerated libraries follow standard APIs, thus enabling acceleration with minimal code changes
QUALITY	Libraries offer high-quality implementations of functions encountered in a broad range of applications
PERFORMANCE	NVIDIA libraries are tuned by experts

DROP-IN ACCELERATION

In Two Easy Steps

```
int N = 1 << 20;         // 1M elements

x = (float *)malloc(N * sizeof(float));
y = (float *)malloc(N * sizeof(float));
initData(x, y);

// Perform SAXPY on 1M elements: y[]=a*x[]+y[]
saxpy(N, 2.0, x, 1, y, 1);

useResult(y);
```

```
int N = 1 << 20;         // 1M elements

x = (float *)malloc(N * sizeof(float));
y = (float *)malloc(N * sizeof(float));
initData(x, y);

// Perform SAXPY on 1M elements: y[]=a*x[]+y[]
saxpy(N, 2.0, x, 1, y, 1);

useResult(y);
```

DROP-IN ACCELERATION

In Two Easy Steps

```
int N = 1 << 20;         // 1M elements

x = (float *)malloc(N * sizeof(float));
y = (float *)malloc(N * sizeof(float));
initData(x, y);

// Perform SAXPY on 1M elements: y[]=a*x[]+y[]
saxpy(N, 2.0, x, 1, y, 1);

useResult(y);
```

Step 1: Manage Data

```
int N = 1 << 20;         // 1M elements

x = (float *)malloc(N * sizeof(float));
y = (float *)malloc(N * sizeof(float));
cudaMalloc(&d_x, N * sizeof(float));
cudaMalloc(&d_y, N * sizeof(float));
initData(x, y);
// init cublas
cublasStatus_t status = cublasCreate(&handle);

// Copy working data from CPU->GPU
cublasSetVector(N, sizeof(x[0]), x, 1, d_x, 1);
cublasSetVector(N, sizeof(y[0]), y, 1, d_y, 1);

saxpy(N, 2.0, x, 1, y, 1);

// Bring the result back to the CPU
cublasGetVector(N, sizeof(y[0]), d_y, 1, y, 1);
useResult(y);
```

DROP-IN ACCELERATION

In Two Easy Steps

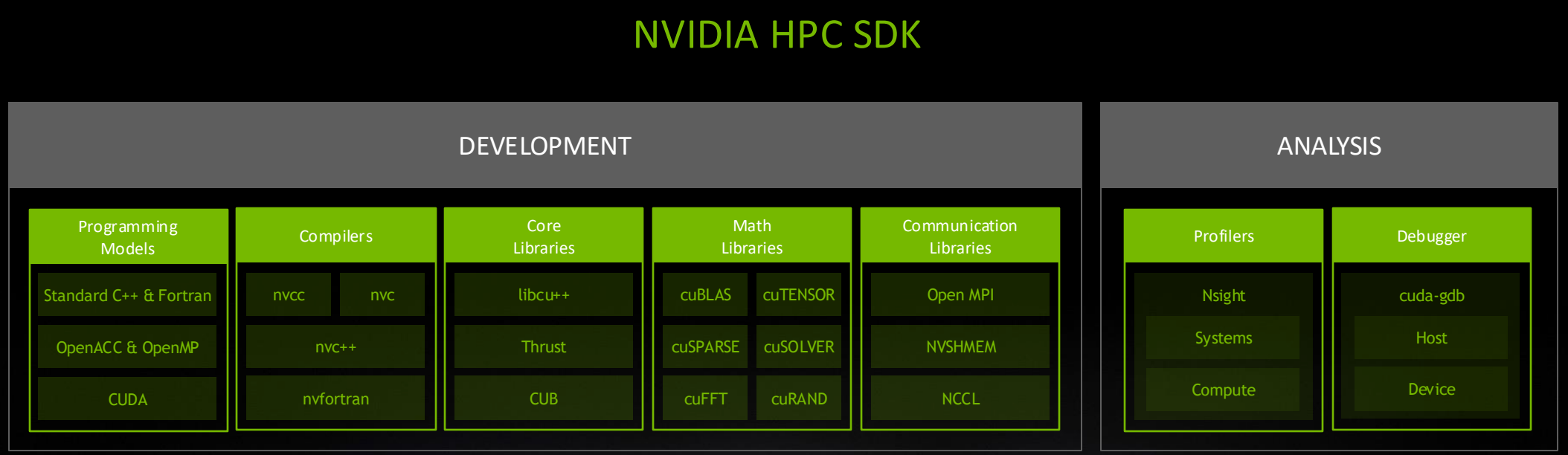
```
int N = 1 << 20;         // 1M elements

x = (float *)malloc(N * sizeof(float));
y = (float *)malloc(N * sizeof(float));
initData(x, y);

// Perform SAXPY on 1M elements: y[]=a*x[]+y[]
saxpy(N, 2.0, x, 1, y, 1);

useResult(y);
```

Step 2: Call Device library

```
int N = 1 << 20;         // 1M elements

x = (float *)malloc(N * sizeof(float));
y = (float *)malloc(N * sizeof(float));
cudaMalloc(&d_x, N * sizeof(float));
cudaMalloc(&d_y, N * sizeof(float));
initData(x, y);
// init cublas
cublasStatus_t status = cublasCreate(&handle);

// Copy working data from CPU->GPU
cublasSetVector(N, sizeof(x[0]), x, 1, d_x, 1);
cublasSetVector(N, sizeof(y[0]), y, 1, d_y, 1);

cublasSaxpy(handle, N, 2.0, d_x, 1, d_y, 1);

// Bring the result back to the CPU
cublasGetVector(N, sizeof(y[0]), d_y, 1, y, 1);
useResult(y);
```

THE NVIDIA HPC SDK

Available at developer.nvidia.com/hpc-sdk



Develop for the NVIDIA HPC Platform: GPU, CPU and Interconnect

HPC Libraries | GPU Accelerated C++ and Fortran | Directives | CUDA

7-8 Releases Per Year | Freely Available

HPC PROGRAMMING IN ISO C++

C++ Parallel Algorithms

```
std::sort(std::execution::par, c.begin(), c.end());

std::unique(std::execution::par, c.begin(), c.end());
```

- Introduced in C++17
- Parallel and vector concurrency via execution policies

  `std::execution::par, std::execution::par_seq, std::execution::seq`
- Several new algorithms in C++17 including
  - `std::for_each_n(POLICY, first, size, func)`
- Insert `std::execution::par` as first parameter when calling algorithms
- NVC++ 20.5: automatic GPU acceleration of C++17 parallel algorithms
  - Leverages CUDA Unified Memory

NVIDIA.

OpenACC COMPILER DIRECTIVES

Parallel C Code

```
void saxpy(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];
}

...
// Perform SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
...
```

Parallel Fortran Code

```
subroutine saxpy(n, a, x, y)
  real :: x(:), y(:), a
  integer :: n, i
!$acc kernels
  do i=1,n
    y(i) = a*x(i)+y(i)
  enddo
!$acc end kernels
end subroutine saxpy



...
! Perform SAXPY on 1M elements
call saxpy(2**20, 2.0, x_d, y_d)
...
```

http://developer.nvidia.com/openacc or http://openacc.org

CUDA C

```
void saxpy_serial(int n,
                  float a,
                  float *x,
                  float *y)
{

  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}

// Perform SAXPY on 1M elements
saxpy_serial(4096*256, 2.0, x, y);
```

```
__global__
void saxpy_parallel(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];
}

// Perform SAXPY on 1M elements
saxpy_parallel<<<4096,256>>>(n,2.0,x,y);
```

http://developer.nvidia.com/cuda-toolkit

CUDA C++: DEVELOP GENERIC PARALLEL CODE

CUDA C++ features enable sophisticated and flexible applications and middleware

- Class hierarchies
- __device__ methods
- Templates
- Operator overloading
- Functors (function objects)
- Device-side new/delete
- More…

http://developer.nvidia.com/cuda-toolkit

```
template <typename T>
struct Functor {
  __device__ Functor(_a) : a(_a) {}
  __device__ T operator(T x) { return a*x; }
  T a;
}

template <typename T, typename Oper>
__global__ void kernel(T *output, int n) {
  Oper op(3.7);
  output = new T[n]; // dynamic allocation
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n)
    output[i] = op(i);  // apply functor
}
```

CUDA FORTRAN

- Program GPU using Fortran
  - Key language for HPC
- Simple language extensions
  - Kernel functions
  - Thread / block IDs
  - Device & data management
  - Parallel loop directives
- Familiar syntax
  - Use allocate, deallocate
  - Copy CPU-to-GPU with assignment (=)

http://developer.nvidia.com/cuda-fortran

```
module mymodule contains
  attributes(global) subroutine saxpy(n,a,x,y)
    real :: x(:), y(:), a,
    integer n, i
    attributes(value) :: a, n
    i = threadIdx%x+(blockIdx%x-1)*blockDim%x
    if (i<=n) y(i) = a*x(i) + y(i);
  end subroutine saxpy
end module mymodule

program main
  use cudafor; use mymodule
  real, device :: x_d(2**20), y_d(2**20)
  x_d = 1.0; y_d = 2.0
  call saxpy<<<4096,256>>>(2**20,3.0,x_d,y_d,)
  y = y_d
  write(*,*) 'max error=', maxval(abs(y-5.0))
end program main
```


INTRO TO CUDA

NVCC COMPILER

NVIDIA provides a CUDA C++ compiler: nvcc

Specify GPU architecture with: `-arch`, e.g. `-arch sm_80` (NVIDIA A100)

NVCC splits your code in 2: Host code and Device code.

Host code forwarded to CPU compiler (usually g++)

Device code sent to NVIDIA device compiler

NVCC is capable of linking together both host and device code into a single executable

Convention: C++ source files containing CUDA syntax typically use the extension .cu.

OUR FIRST KERNEL

```
__global__ void mykernel(void) {
}

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

The __global__ annotation informs the compiler that this is a kernel, which will be invoked on the device from the host.

The angle bracket, or “chevron”, syntax informs the compiler how many copies of the kernel “mykernel” to invoke. Here we will use 1 instance.

OUR FIRST KERNEL

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

int main(void) {
        mykernel<<<1,1>>>();
        cudaDeviceSynchronize();
        return 0;
}
```

Move the work into the kernel.

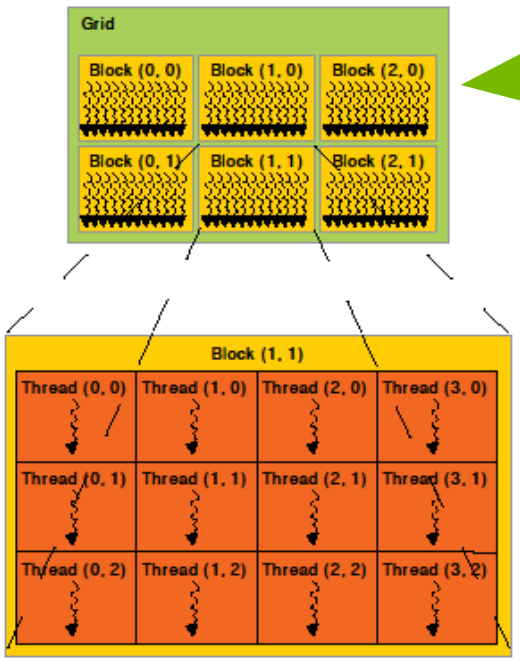
Tell the host to wait until the device is finished.

WHERE IS THE PARALLELSIM

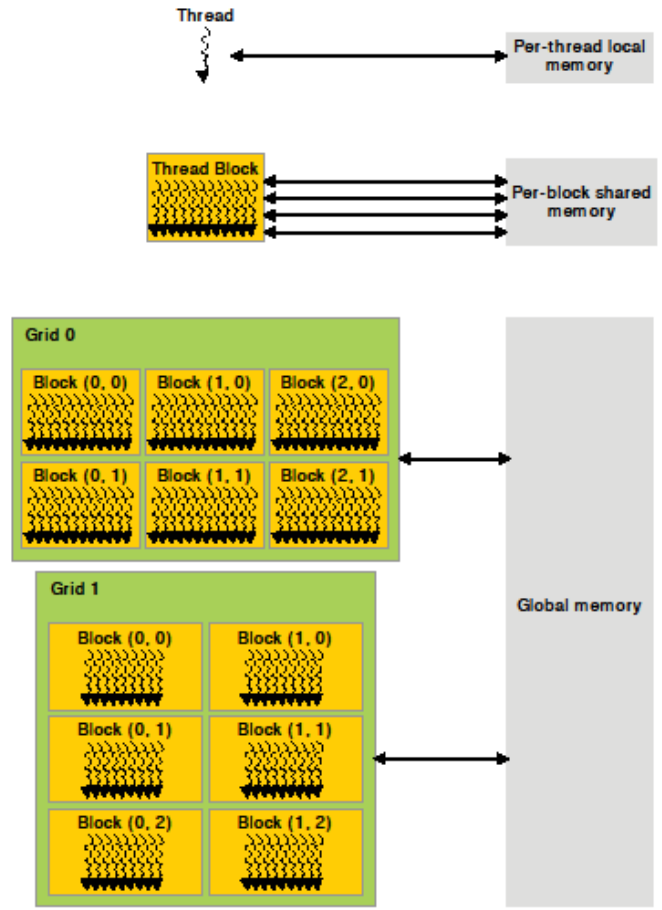
```
void do_something_parallel(…) {
        kernel<<<NUMBER_BLOCKS,BLOCKSIZE>>>(…);
        cudaDeviceSynchronize();
}
```

launches
NUMBER_BLOCKS x BLOCKSIZE
copies (threads)

THREADS? BLOCKS? GRID?



Scalable: Blocks are assigned to available SMs



WHICH COPY AM I ?

Built-in variables:

threadIdx.x	Thread index within the block
blockIdx.x	Block index within the grid
blockDim.x	Number of threads in a block
gridDim.x	Number of blocks in a grid

Note: For programmer convenience these are all 3-d. (e.g. `threadIdx.{x,y,z}`)

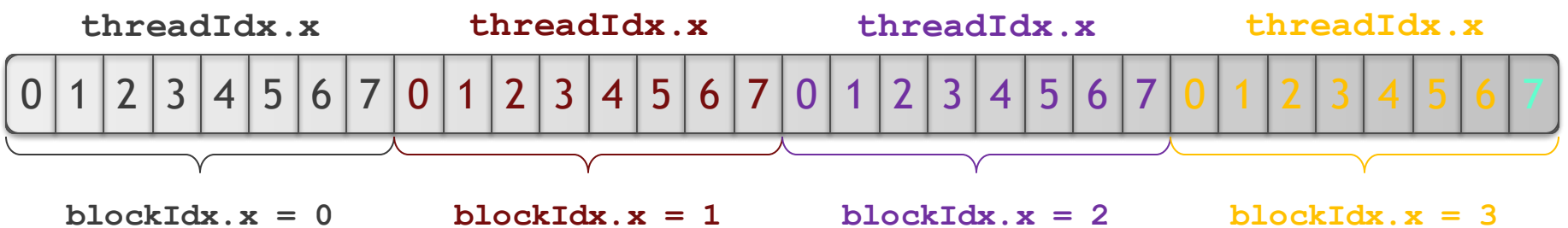
These exist automatically in CUDA kernels

Read only (set by the runtime)

INDEXING ARRAYS WITH BLOCKS AND THREADS

No longer as simple as using `blockIdx.x` and `threadIdx.x`

Consider indexing an array with one element per thread (8 threads/block)



With `blockDim.x` threads per block, a unique index for each thread is given by:

```
int index = blockIdx.x * blockDim.x + threadIdx.x
```


NVIDIA.

SOME BEST PRACTICES

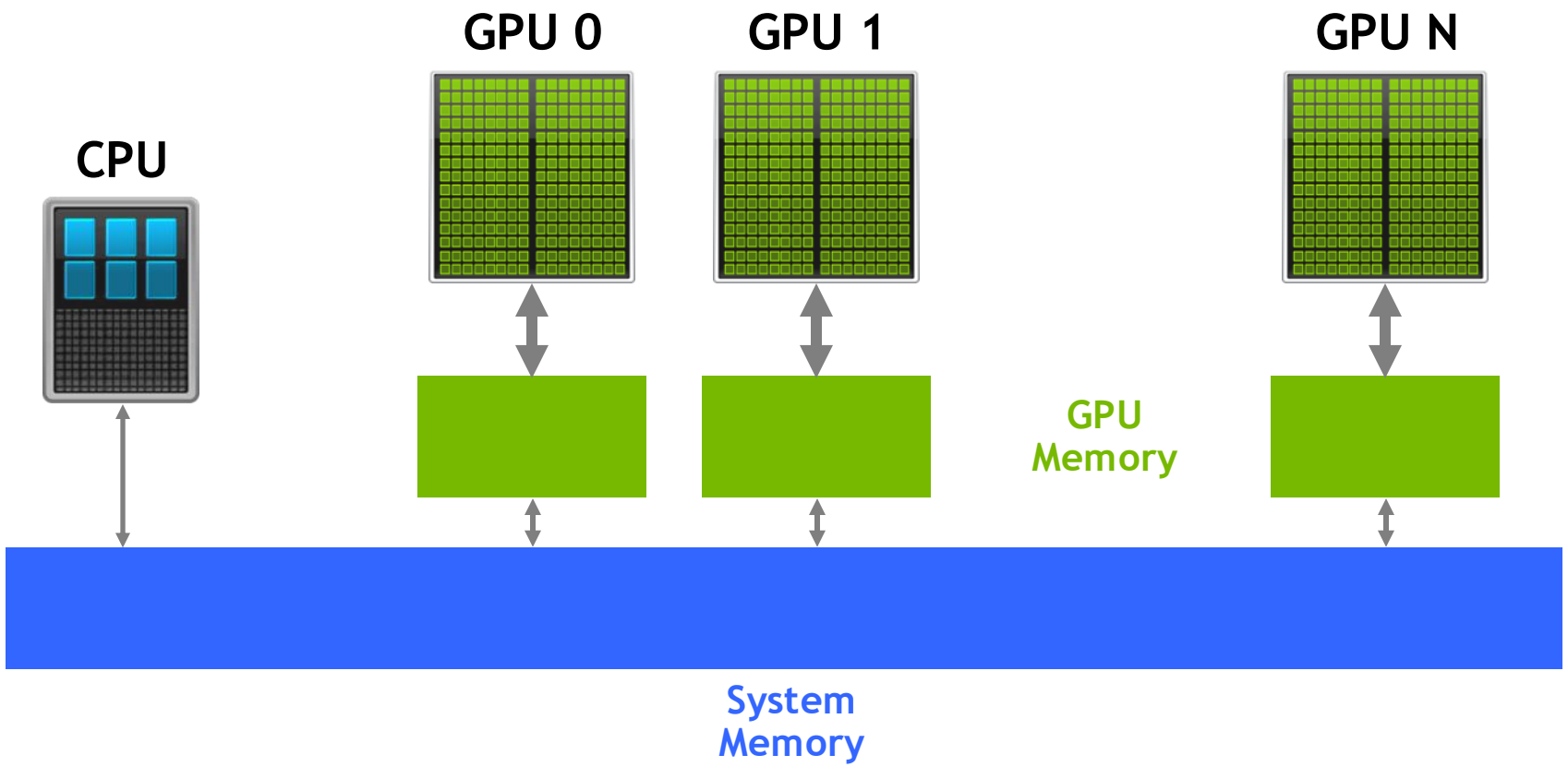
```
__global__ void kernel(… , int N)) {
        int i = blockIdx.x * blockDim.x + threadIdx.x;
        if ( i < N ) // Protect against out-of-bounds error
            …
}

void do_something_parallel(…, int N) {
        kernel<<<(N+BLOCKSIZE-1)/BLOCKSIZE,BLOCKSIZE>>>(…, N);
       cudaDeviceSynchronize();
}
```

If N is not evenly divisible by BLOCKSIZE, this will ensure enough blocks are created to cover all data elements.


DATA MANAGEMENT

HETEROGENEOUS PLATFORM

Memory hierarchy

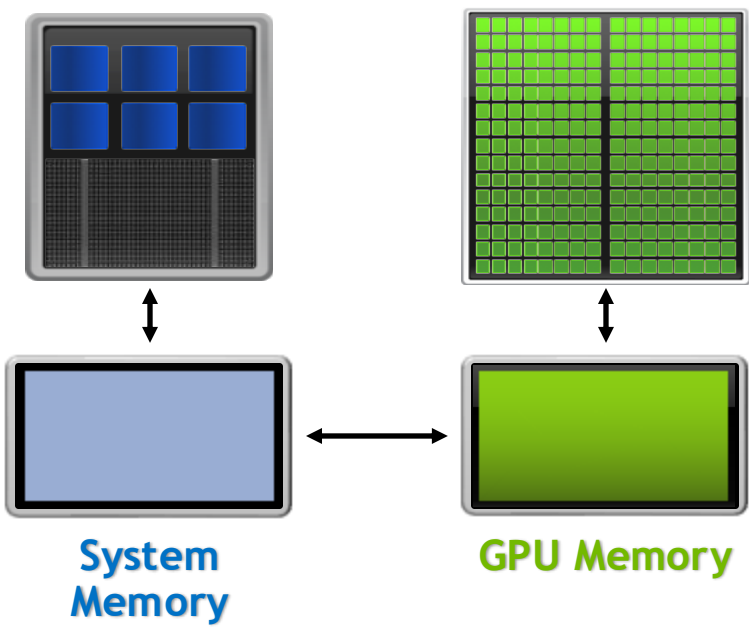


NVIDIA.

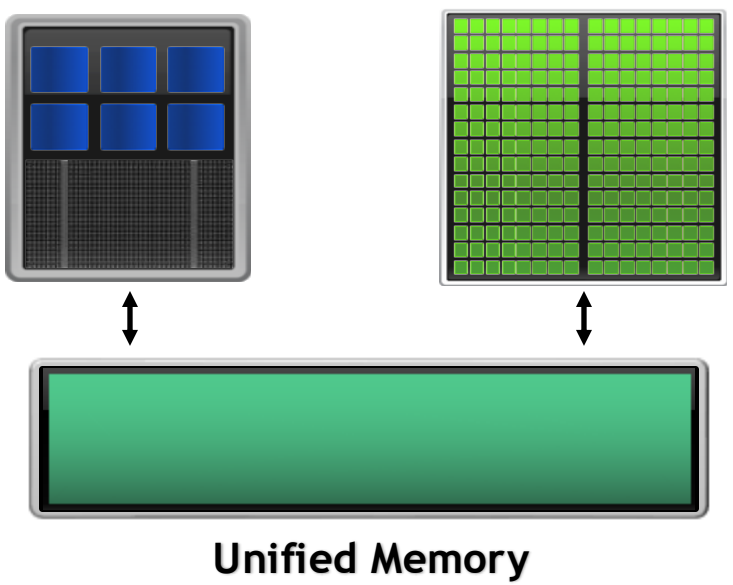
CUDA UNIFIED MEMORY

AKA Managed Memory

Without Unified Memory



With Unified Memory



UNIFIED MEMORY

Single pointer for CPU and GPU

CPU code

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

  fread(data, 1, N, fp);

  qsort(data, N, 1, compare);


  use_data(data);

  free(data);
}
```

GPU code with Unified Memory

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

  fread(data, 1, N, fp);

  qsort<<<...>>>(data,N,1,compare);
  cudaDeviceSynchronize();

  use_data(data);

  cudaFree(data);
}
```

MANAGING DEVICE MEMORY

cudaMallocManaged/cudaFree

Allocate `size` bytes on the current device

C++:

```
template<class T> cudaError_t cudaMallocManaged( T **Ptr, size_t size );
```

C:

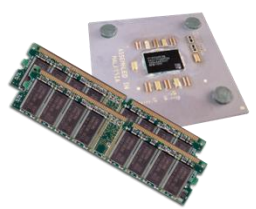
```
cudaError_t cudaMallocManaged(void **Ptr, size_t size);
```

Deallocate C/C++:

```
cudaError_t cudaFree(void *devPtr);
```

EXPLICIT MEMORY MANAGEMENT

- Host and device memory are distinct entities
  - Device pointers point to GPU memory
    - May be passed to and from host code
    - May not be dereferenced from host code
  - Host pointers point to CPU memory
    - May be passed to and from device code
    - May not be dereferenced from device code



- Basic CUDA API for dealing with device memory
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`
  - Similar to their C equivalents, `malloc()`, `free()`, `memcpy()`

MANAGING DEVICE MEMORY

cudaMalloc/cudaFree

Allocate `size` bytes on the current device

C++:

```
template<class T> cudaError_t cudaMalloc( T **devPtr, size_t size );
```

C:

```
cudaError_t cudaMalloc(void **devPtr, size_t size);
```

Deallocate C/C++:

```
cudaError_t cudaFree(void *devPtr);
```

MANAGING DEVICE MEMORY

cudaMemcpy

Copy `count` bytes from `src` to `dst`:

C/C++:

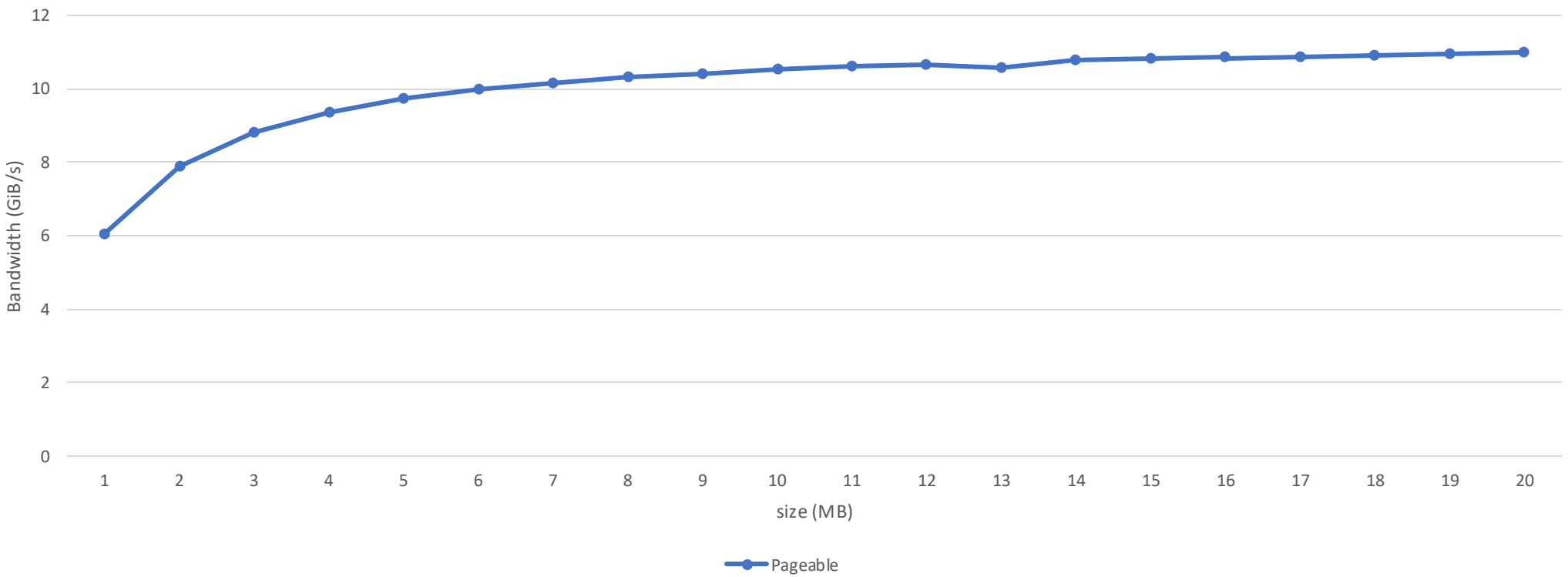
```
cudaError_t cudaMemcpy(void *dst, const void *src, size_t count, enum cudaMemcpyKind kind);
```

`cudaMemcpyKind`:

```
enum cudaMemcpyKind {
    cudaMemcpyHostToHost = 0, /**< Host -> Host */
    cudaMemcpyHostToDevice = 1, /**< Host -> Device */
    cudaMemcpyDeviceToHost = 2, /**< Device -> Host */
    cudaMemcpyDeviceToDevice = 3, /**< Device -> Device */
    cudaMemcpyDefault = 4 /**< Direction of the transfer is inferred from the pointer values. Requires unified virtual addressing (UVA) */
};
```

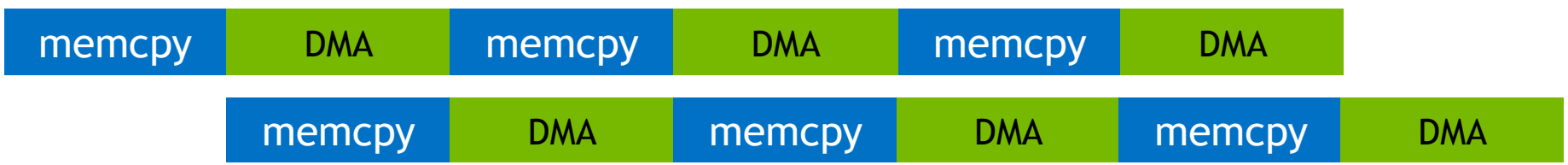
COPY DATA FROM CPU TO GPU

Host to Device Bandwidth on x86 + P100 PCI-E 16GB



COPY DATA FROM CPU TO GPU

Pageable host memory



REGISTERING HOST MEMORY

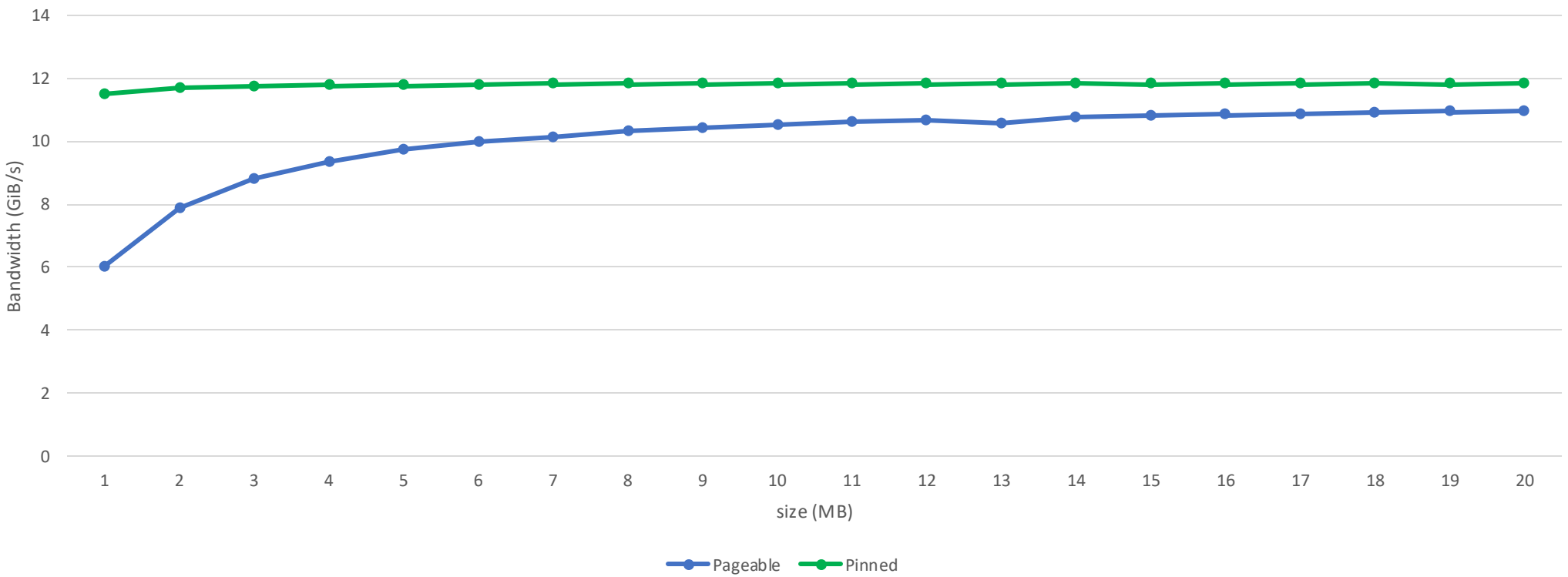
```
int* ptr;
cudaMallocHost ( &ptr, n*sizeof(int) );




cudaFreeHost ( ptr );
```

```
int* ptr = new int[n];
cudaHostRegister ( &ptr, n*sizeof(int),
                   cudaHostRegisterMapped );
int* ptr_d;
cudaHostGetDevicePointer ( &ptr_d, ptr, 0 );

cudaHostUnregister( ptr );
ptr_d = 0;
delete[] ptr;
```

- Accelerates Host to Device and Device to Host copies
- Avoids unwanted tax of CPU memcpy to CPU memory bandwidth
- Required for truly asynchronous Host to Device and Device to Host copies
- Allows direct access from the GPU without a copy (Zero Copy)

REGISTERING HOST MEMORY

Host to Device Bandwidth on x86 + P100 PCI-E 16GB



NVIDIA.

LINKS

https://docs.nvidia.com/cuda/index.html

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


NVIDIA