Introduction to GPU Computing with CUDA

Siegfried Höfinger

ASC Research Center, TU Wien

October 20, 2025

→ https://tinyurl.com/cudafordummies/i/l1/notes-l1.pdf

Outline

Current Situation — Glimpse into top500

Components

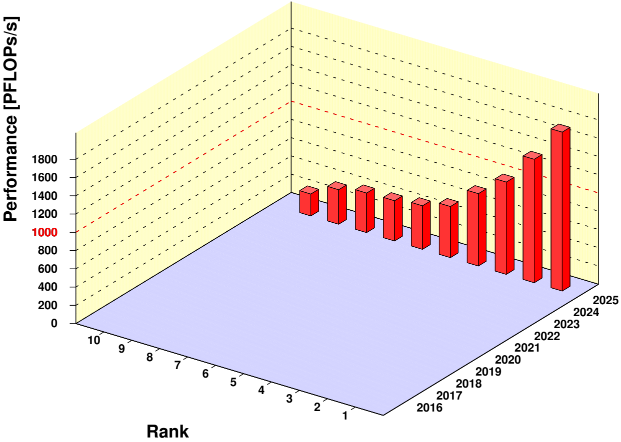
Historical

Consumer/Enterprise-Grade GPUs

CUDA — Basic Design Principles

Take Home Messages

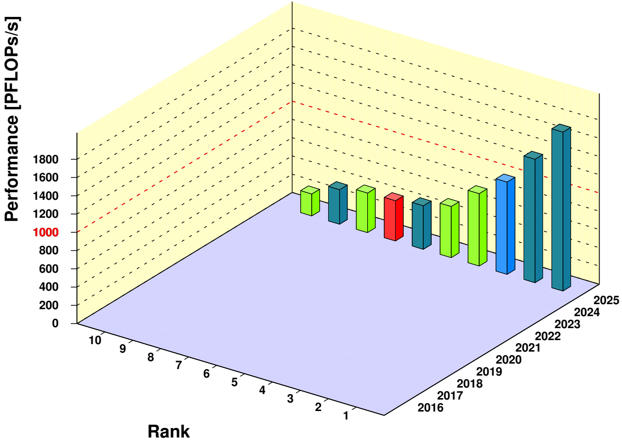
Current Situation — Glimpse into top500



- HPC — 3rd pillar of scientific discovery

→ PRACE Software Strategy for European Exascale Systems

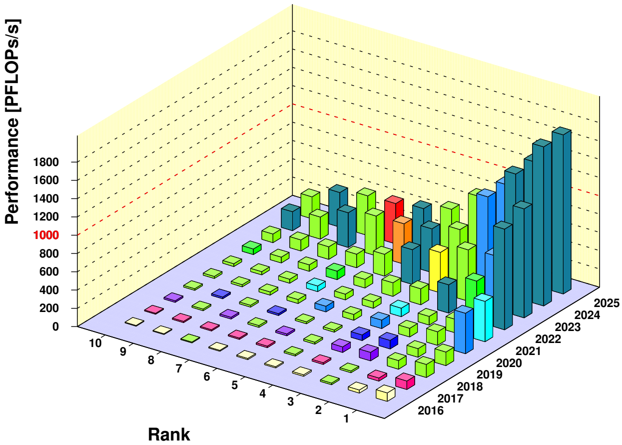
Current Situation — Glimpse into top500



- HPC — 3rd pillar of scientific discovery
- Supercomputers — frequently made of GPUs !

→ PRACE Software Strategy for European Exascale Systems

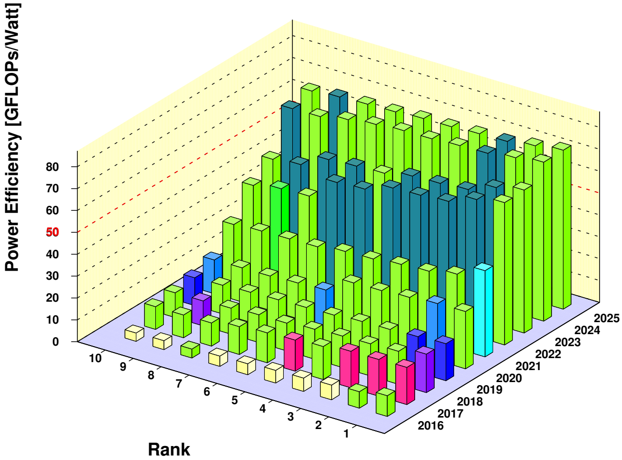
Current Situation — Glimpse into top500



- HPC — 3rd pillar of scientific discovery
- Supercomputers — frequently made of GPUs !
- Trend is likely to continue

→ PRACE Software Strategy for European Exascale Systems

Current Situation — Glimpse into top500

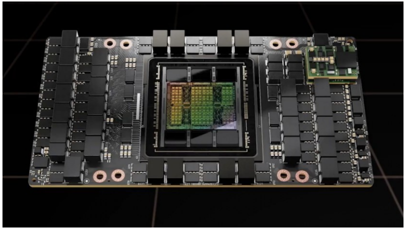


- HPC — 3rd pillar of scientific discovery
- Supercomputers — frequently made of GPUs !
- Trend is likely to continue
- Power efficiency is key

→ PRACE Software Strategy for European Exascale Systems

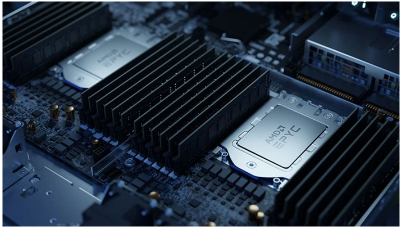
Components

GPU/Accelerator:



Specs (H100):
15872 cores, clock freq 1.8 GHz, 80 GB HBM3, 3.4 TB/s, FP64/FP32/TC-FP64 34/67/67 TFLOPs/s, TDP 700 W, PCIe5/NVLink3 128/900 GB/s;

HPC/Server:

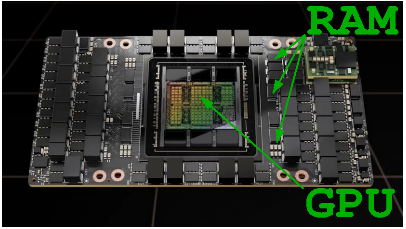


Specs (AMD EPYC 3rd Milan):
64 cores, clock freq 2.0 GHz, up to 2048 GB DDR4, up to 205 GB/s, FP64 2.3 TFLOPs/s, TDP 225 W;

→ https://en.wikipedia.org/wiki/Supercomputer
→ https://www.nvidia.com/en-us/data-center/h100

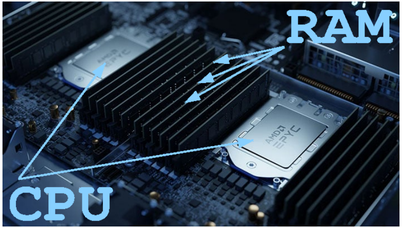
Components

GPU/Accelerator:



Specs (H100):
15872 cores, clock freq 1.8 GHz, 80 GB HBM3, 3.4 TB/s, FP64/FP32/TC-FP64 34/67/67 TFLOPs/s, TDP 700 W, PCIe5/NVLink3 128/900 GB/s;

HPC/Server:



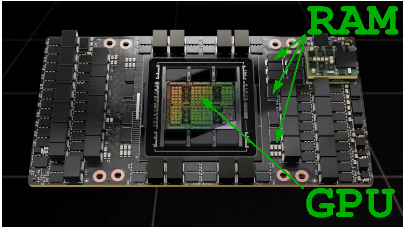
Specs (AMD EPYC 3rd Milan):
64 cores, clock freq 2.0 GHz, up to 2048 GB DDR4, up to 205 GB/s, FP64 2.3 TFLOPs/s, TDP 225 W;

- Identical basic components

→ https://en.wikipedia.org/wiki/Supercomputer
→ https://www.nvidia.com/en-us/data-center/h100

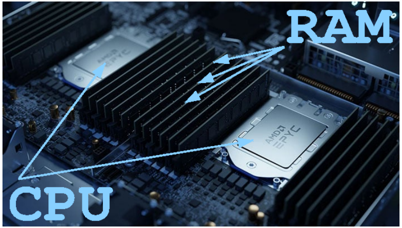
Components

GPU/Accelerator:



Specs (H100):
15872 cores, clock freq 1.8 GHz, 80 GB HBM3, 3.4 TB/s, FP64/FP32/TC-FP64 34/67/67 TFLOPs/s, TDP 700 W, PCIe5/NVLink3 128/900 GB/s;

HPC/Server:



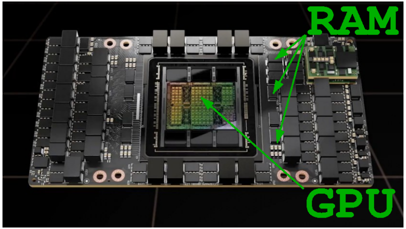
Specs (AMD EPYC 3rd Milan):
64 cores, clock freq 2.0 GHz, up to 2048 GB DDR4, up to 205 GB/s, FP64 2.3 TFLOPs/s, TDP 225 W;

- Identical basic components
- GPU has much more cores, but less RAM

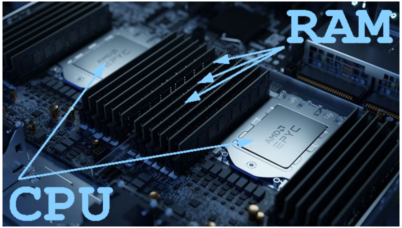
→ https://en.wikipedia.org/wiki/Supercomputer
→ https://www.nvidia.com/en-us/data-center/h100

Components

GPU/Accelerator:



HPC/Server:



Specs (H100):
15872 cores, clock freq 1.8 GHz, 80 GB HBM3, 3.4 TB/s, FP64/FP32/TC-FP64 34/67/67 TFLOPs/s, TDP 700 W, PCIe5/NVLink3 128/900 GB/s;

Specs (AMD EPYC 3rd Milan):
64 cores, clock freq 2.0 GHz, up to 2048 GB DDR4, up to 205 GB/s, FP64 2.3 TFLOPs/s, TDP 225 W;

- Identical basic components
- GPU has much more cores, but less RAM
- No network on the GPU (massive parallelism onboard)

→ https://en.wikipedia.org/wiki/Supercomputer
→ https://www.nvidia.com/en-us/data-center/h100

Historical

Fermi

2010

Tsubame 2.0 (M2050)

GSIC/TITech
2.3 PFLOPs/s

→ https://www.nvidia.com

Historical

Fermi

2010

Tsubame 2.0 (M2050)

GSIC/TITech
2.3 PFLOPs/s

Kepler

2012

Titan (k20x)

ORNL
17.6 PFLOPs/s
⋆ 3x #cores (1536)
⋆ improved power efficiency

→ https://www.nvidia.com

Historical

Fermi

2010

Tsubame 2.0 (M2050)

GSIC/TITech
2.3 PFLOPs/s

Kepler

2012

Titan (k20x)

ORNL
17.6 PFLOPs/s
⋆ 3x #cores (1536)
⋆ improved power efficiency

Pascal

2016

Piz Daint (P100)

CSCS
25.4 PFLOPs/s
⋆ NVLink, 5x PCIe bw
⋆ HBM2, 3x memory bw
⋆ Unified memory, multi-GPU/CPU

→ https://www.nvidia.com

Fermi

2010

Tsubame 2.0 (M2050)

GSIC/TITech
2.3 PFLOPs/s

Kepler

2012

Titan (k20x)

ORNL
17.6 PFLOPs/s
- ⋆ 3x #cores (1536)
- ⋆ improved power efficiency

Pascal

2016

Piz Daint (P100)

CSCS
25.4 PFLOPs/s
- ⋆ NVLink, 5x PCIe bw
- ⋆ HBM2, 3x memory bw
- ⋆ Unified memory, multi-GPU/CPU

Volta

2018

Summit/Sierra (V100)

ORNL/LLNL
148.6/94.6 PFLOPs/s
- ⋆ NVLink2, 2x previous
- ⋆ AI, 640 tensor cores

→ https://www.nvidia.com

Historical

Fermi

2010

Tsubame 2.0 (M2050)

GSIC/TITech
2.3 PFLOPs/s

Kepler

2012

Titan (k20x)

ORNL
17.6 PFLOPs/s
⋆ 3x #cores (1536)
⋆ improved power efficiency

Pascal

2016

Piz Daint (P100)

CSCS
25.4 PFLOPs/s
⋆ NVLink, 5x PCIe bw
⋆ HBM2, 3x memory bw
⋆ Unified memory, multi-GPU/CPU

Volta

2018

Summit/Sierra (V100)

ORNL/LLNL
148.6/94.6 PFLOPs/s
⋆ NVLink2, 2x previous
⋆ AI, 640 tensor cores

Ampere

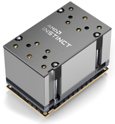
2020

Perlmutter/JuwelsBooster(A100)

NERSC/FZJ
64.6/44.1 PFLOPs/s (#5/8)
⋆ FP64@TC (19.5 TFLOPS/s)
⋆ all up by 1.5x
⋆ ≈25 GFLOPs/Watt (#7/11)

→ https://www.nvidia.com

AMD Instinct MI250X



2022

Frontier/Lumi/Adastra(MI250X)

ORNL/CSC/GENCI-CINES
1102/152/46 PFLOPs/s (#1/3/10)

⋆ FP64 (47.9 TFLOPS/s)

⋆ FP64@Mx (95.7 TFLOPS/s)

⋆ ≈50 GFLOPs/Watt (#2/3/4)

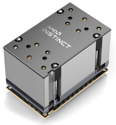
→ https://www.amd.com/en/products/server-accelerators/instinct-mi250x

→ https://www.intel.com/content/www/us/en/products/sku/232873/intel-data-center-gpu-max-1550/specifications.html

→ https://resources.nvidia.com/en-us-hopper-architecture/nvidia-h100-tensor-c

Historical cont.

AMD Instinct MI250X



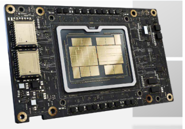
2022

Frontier/Lumi/Adastra(MI250X)

ORNL/CSC/GENCI-CINES
1102/152/46 PFLOPs/s (#1/3/10)

⋆ FP64 (47.9 TFLOPS/s)
⋆ FP64@Mx (95.7 TFLOPS/s)
⋆ ≈50 GFLOPs/Watt (#2/3/4)

Intel Data Center GPU Max



2023

Aurora(Intel Ponte Vecchio)

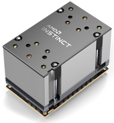
ANL
1012 PFLOPs/s (#2)

⋆ FP64 (52 TFLOPS/s)
⋆ 26 GFLOPs/Watt (#42)

→ https://www.amd.com/en/products/server-accelerators/instinct-mi250x
→ https://www.intel.com/content/www/us/en/products/sku/232873/intel-data-center-gpu-max-1550/specifications.html
→ https://resources.nvidia.com/en-us-hopper-architecture/nvidia-h100-tensor-c

Historical cont.

AMD Instinct MI250X

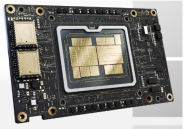


2022

Frontier/Lumi/Adastra(MI250X)

ORNL/CSC/GENCI-CINES
1102/152/46 PFLOPs/s (#1/3/10)
⋆ FP64 (47.9 TFLOPS/s)
⋆ FP64@Mx (95.7 TFLOPS/s)
⋆ ≈50 GFLOPs/Watt (#2/3/4)

Intel Data Center GPU Max

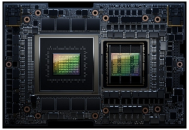


2023

Aurora(Intel Ponte Vecchio)

ANL
1012 PFLOPs/s (#2)
⋆ FP64 (52 TFLOPS/s)
⋆ 26 GFLOPs/Watt (#42)

Grace-Hopper



2024

Jupiter/Alps(H100)

JSC/CSCS
793/PFLOPs/s (#4/6)
⋆ FP64 (34 TFLOPS/s)
⋆ FP64@TC (67 TFLOPS/s)
⋆ 52 GFLOPs/Watt (#14)

→ https://www.amd.com/en/products/server-accelerators/instinct-mi250x
→ https://www.intel.com/content/www/us/en/products/sku/232873/intel-data-center-gpu-max-1550/specifications.html
→ https://resources.nvidia.com/en-us-hopper-architecture/nvidia-h100-tensor-c

Historical cont.

Compute Capabilities

Version Number	GPU Architecture
9.0	Hopper
8.9	Ada Lovelace
8.0	Ampere
7.5	Turing
7.x	Volta
6.x	Pascal
5.x	Maxwell
3.x	Kepler
2.x	Fermi
1.x	Tesla

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#application-compatibility

Historical cont.

Compute Capabilities

Version Number	GPU Architecture
9.0	Hopper
8.9	Ada Lovelace
8.0	Ampere
7.5	Turing
7.x	Volta
6.x	Pascal
5.x	Maxwell
3.x	Kepler
2.x	Fermi
1.x	Tesla

Major revision number: identifies core architecture/hardware features

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#application-compatibility

Historical cont.

Compute Capabilities

Version Number	GPU Architecture
9.0	Hopper
8.9	Ada Lovelace
8.0	Ampere
7.5	Turing
7.x	Volta
6.x	Pascal
5.x	Maxwell
3.x	Kepler
2.x	Fermi
1.x	Tesla

Major revision number: identifies core architecture/hardware features

Minor revision number: incremental update to core architecture, e.g. Turing-Volta

→ https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#application-compatibility

Consumer/Enterprise-Grade GPUs

- Consumer grade: made for gaming, cheaper devices with lower specs (FP64, HBM*) and prohibited 24x7 usage in datacentres (EULA change since 12/2017 affecting NVIDIA driver); GeForce, Titan, Tegra
- Enterprise grade: heavy HPC workloads and large-scale AI, expensive (10:1) high-end devices with certified top notch components and explicit warranty for stable and reliable 24x7 operation; Tesla, Quadro, DGX/HGX
- Academia perhaps fine; NVIDIA doesn't want to ban non-commercial uses and research, key question is what qualifies as a *"data center"*

→ https://www.nvidia.com/enterpriseservices
→ https://www.theregister.co.uk/2018/01/03/nvidia_server_gpus

Consumer/Enterprise-Grade GPUs cont.

Figuring Out Own Setup

```
cuda—zen sh@n3073—004:~$  nvidia-smi

Wed Oct 15 17:02:44 2025
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 545.23.08              Driver Version: 545.23.08    CUDA Version: 12.3     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  NVIDIA A100-PCIE-40GB          Off | 00000000:01:00.0 Off |                  Off |
| N/A   45C    P0              41W / 250W |      4MiB / 40960MiB |      0%      Default |
|                                         |                      |             Disabled |
+-----------------------------------------+----------------------+----------------------+
|   1  NVIDIA A100-PCIE-40GB          Off | 00000000:81:00.0 Off |                  Off |
| N/A   41C    P0              39W / 250W |      4MiB / 40960MiB |      4%      Default |
|                                         |                      |             Disabled |
+-----------------------------------------+----------------------+----------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|  No running processes found                                                           |
```

Consumer/Enterprise-Grade GPUs cont.

Figuring Out Own Setup cont.

```
cuda-zen sh@n3073-004:~$  nvidia-smi topo --matrix

        GPU0    GPU1    NIC0    CPU Affinity     NUMA Affinity    GPU NUMA ID
GPU0     X      SYS     SYS     48-63,176-191    3                N/A
GPU1    SYS      X      SYS     112-127,240-255  7                N/A
NIC0    SYS     SYS      X

Legend:

  X    = Self
  SYS  = Connection traversing PCIe as well as the SMP interconnect between NUMA nodes (e.g., QPI/UPI)
  NODE = Connection traversing PCIe as well as the interconnect between PCIe Host Bridges within a NUMA node
  PHB  = Connection traversing PCIe as well as a PCIe Host Bridge (typically the CPU)
  PXB  = Connection traversing multiple PCIe bridges (without traversing the PCIe Host Bridge)
  PIX  = Connection traversing at most a single PCIe bridge
  NV#  = Connection traversing a bonded set of # NVLinks
```

→ https://stackoverflow.com/questions/55364149/understanding-nvidia-smi-topo-m-output

CUDA — Basic Design Principles

3 Basic Components

Driver

- kernel modules: nvidia.ko nvidia-uvm.ko
- also includes libcuda.so
- considers compute capability !

CUDA Toolkit

- nvcc cuda-gdb nsight...
- libcudart.so libcublas.so...
- also considers compute capability !

CUDA SDK

- examples in 9 sub-directories

→ https://www.nvidia.com/Download/index.aspx?lang=en-us
→ https://developer.nvidia.com/cuda-gpus
→ https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html

CUDA — Basic Design Principles cont.

Good to Know Facts

- CUDA: Compute Uniform Device Architecture (NVIDIA 2006)
- GPU programming model based on threads, shared memory and barrier synchronization
- Linux, Mac OS and Windows supported
- Simple extensions to C functions (becoming kernels) to run on the GPU in parallel as N independent CUDA threads
- Multiple GPUs per host supported
- CUDA the de-facto standard in HPC for science
- CUDA uses simplified logic, more focus on ALU rather than out-of-order execution, branch prediction etc

→ https://developer.nvidia.com/cuda-faq

CUDA — Basic Design Principles cont.

Good to Know Facts cont.

- SIMD operations, single instruction multiple data
- CUDA programs can be called from C, C++, Fortran, Python...
- PTX (parallel thread execution) format is forward-compatible with upcoming GPU generations *.cu $\xrightarrow[\text{nvcc}]{}$ PTX $\xrightarrow[\text{cudart}]{\text{driver}}$ *.exe
- Provides a mini-HPC-cluster on the desktop-computer
- Data movement over PCIe still critical
- OpenCL (alternative API) supported too
- Competitors, AMD (ATI), Intel

→ https://developer.nvidia.com/cuda-faq

CUDA — Basic Design Principles cont.

CUDA C-Programming Guide, Introduction

- GPU specialized for compute-intensive, highly parallel computation

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

- GPU specialized for compute-intensive, highly parallel computation
- More transistors are dedicated to data processing rather than data caching and flow control

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

CUDA — Basic Design Principles cont.

CUDA C-Programming Guide, Introduction

- GPU specialized for compute-intensive, highly parallel computation
- More transistors are dedicated to data processing rather than data caching and flow control
- GPU is a highly parallel, multithreaded, manycore processor with very high memory bandwidth

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

CUDA — Basic Design Principles cont.

CUDA C-Programming Guide, Introduction

- GPU specialized for compute-intensive, highly parallel computation
- More transistors are dedicated to data processing rather than data caching and flow control
- GPU is a highly parallel, multithreaded, manycore processor with very high memory bandwidth
- Power efficiency is key — eco-friendly computing

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

Single Thread Block Vector Addition

```
// kernel definition;
__global__ void VecAdd(float *A, float *B, float *C)
{
    int i;
    i = threadIdx.x;
    C[i] = A[i] + B[i];
}


int main()
{
    ...
    // kernel invocation with N threads
    N = 100;
    VecAdd <<< 1, N >>> (A, B, C);
    ...
}
```

→ https://tinyurl.com/cudafordummies/i/l1/single_thread_block_vector_addition.cu

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

only 3 basic extensions

Single Thread Block Vector Addition

```
// kernel definition;
__global__ void VecAdd(float *A, float *B, float *C)
{
    int i;
    i = threadIdx.x;
    C[i] = A[i] + B[i];
}


int main()
{
    ...
    // kernel invocation with N threads
    N = 100;
    VecAdd <<< 1, N >>> (A, B, C);
    ...
}
```

→ https://tinyurl.com/cudafordummies/i/l1/single_thread_block_vector_addition.cu
→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

Single Thread Block Vector Addition

only 3 basic extensions

1) kernel declaration specifier

```
// kernel definition;
__global__ void VecAdd(float *A, float *B, float *C)
{
    int i;
    i = threadIdx.x;
    C[i] = A[i] + B[i];
}


int main()
{
    ...
    // kernel invocation with N threads
    N = 100;
    VecAdd <<< 1, N >>> (A, B, C);
    ...
}
```

→ https://tinyurl.com/cudafordummies/i/l1/single_thread_block_vector_addition.cu

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

Single Thread Block Vector Addition

only 3 basic extensions

1) kernel declaration specifier

```
// kernel definition;
__global__ void VecAdd(float *A, float *B, float *C)
{
    int i;
    i = threadIdx.x;
    C[i] = A[i] + B[i];
}


int main()
{
    ...
    // kernel invocation with N threads
    N = 100;
    VecAdd <<< 1, N >>> (A, B, C);
    ...
}
```

2) kernel execution configuration

→ https://tinyurl.com/cudafordummies/i/l1/single_thread_block_vector_addition.cu

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

CUDA — Basic Design Principles cont.

CUDA C-Programming Guide, Programming Model

Single Thread Block Vector Addition

only 3 basic extensions

1) kernel declaration specifier

```
// kernel definition;
__global__ void VecAdd(float *A, float *B, float *C)
{
    int i;
    i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // kernel invocation with N threads
    N = 100;
    VecAdd <<< 1, N >>> (A, B, C);
    ...
}
```

3) built-in variables, e.g. threadIdx.x=0,1,2...

2) kernel execution configuration

→ https://tinyurl.com/cudafordummies/i/l1/single_thread_block_vector_addition.cu

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

CUDA — Basic Design Principles cont.

Compile and Run and Monitor

```
cuda-zen sh@n3073-004:~$  nvcc single_thread_block_vector_addition.cu
cuda-zen sh@n3073-004:~$  ./a.out

0 100.000000
1 100.000000
2 100.000000
3 100.000000
4 100.000000
5 100.000000
6 100.000000
7 100.000000
8 100.000000
9 100.000000
10 100.000000
11 100.000000
...
99 100.000000
```

CUDA — Basic Design Principles cont.

Compile and Run and Monitor cont.

```
cuda-zen sh@n3073-004:~$  watch -n 0.1 nvidia-smi

+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 545.23.08              Driver Version: 545.23.08    CUDA Version: 12.3     |
|-----------------------------------------+----------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  Compute M. |
|                                         |                      |               MIG M. |
|=========================================+======================+======================|
|   0  NVIDIA A100-PCIE-40GB          Off | 00000000:01:00.0 Off |                  Off |
| N/A   45C    P0              41W / 250W |      4MiB / 40960MiB |      0%      Default |
|                                         |                      |             Disabled |
+-----------------------------------------+----------------------+----------------------+
|   1  NVIDIA A100-PCIE-40GB          Off | 00000000:81:00.0 Off |                  Off |
| N/A   41C    P0              39W / 250W |      4MiB / 40960MiB |      4%      Default |
|                                         |                      |             Disabled |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|    0   N/A  N/A    807767      C   ./a.out                                     12MiB |
+---------------------------------------------------------------------------------------+
```

CUDA — Basic Design Principles cont.

CUDA C-Programming Guide, Programming Model cont.

- __global__ declares a function as being a GPU-kernel
  1. executed on the device
  2. callable from the host
  3. also callable from the device for devices of compute capability ≥ 3.2

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

CUDA — Basic Design Principles cont.

CUDA C-Programming Guide, Programming Model cont.

- __global__ declares a function as being a GPU-kernel
  1. executed on the device
  2. callable from the host
  3. also callable from the device for devices of compute capability ≥ 3.2
- __host__ declares a function as being a host-function
  1. executed on the host
  2. callable from the host only
  3. default when omitted

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

CUDA — Basic Design Principles cont.

CUDA C-Programming Guide, Programming Model cont.

- __global__ declares a function as being a GPU-kernel
  1. executed on the device
  2. callable from the host
  3. also callable from the device for devices of compute capability $\geq$ 3.2
- __host__ declares a function as being a host-function
  1. executed on the host
  2. callable from the host only
  3. default when omitted
- __device__ declares a function as being a GPU-only-function
  1. executed on the device
  2. callable from the device only

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

CUDA — Basic Design Principles cont.

CUDA C-Programming Guide, Programming Model cont.

- threadIdx.[x,y,z] may be one-dimensional, two-dimensional or three-dimensional referring to a one-dimensional, two-dimensional or three-dimensional thread block

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

CUDA — Basic Design Principles cont.

CUDA C-Programming Guide, Programming Model cont.

- threadIdx.[x,y,z] may be one-dimensional, two-dimensional or three-dimensional referring to a one-dimensional, two-dimensional or three-dimensional thread block
- threadIdx.[x,y,z] provides a direct formal abstraction of domains, ie facilitates straightforward reference to the elements of a vector, matrix, or a volume

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

CUDA — Basic Design Principles cont.

CUDA C-Programming Guide, Programming Model cont.

- threadIdx.[x,y,z] may be one-dimensional, two-dimensional or three-dimensional referring to a one-dimensional, two-dimensional or three-dimensional thread block
- threadIdx.[x,y,z] provides a direct formal abstraction of domains, ie facilitates straightforward reference to the elements of a vector, matrix, or a volume
- threadIdx.[x,y,z] for N threads goes from 0 to N-1

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

CUDA — Basic Design Principles cont.

CUDA C-Programming Guide, Programming Model cont.

- threadIdx.[x,y,z] may be one-dimensional, two-dimensional or three-dimensional referring to a one-dimensional, two-dimensional or three-dimensional thread block
- threadIdx.[x,y,z] provides a direct formal abstraction of domains, ie facilitates straightforward reference to the elements of a vector, matrix, or a volume
- threadIdx.[x,y,z] for N threads goes from 0 to N-1
- when working with thread blocks of two/three-dimensional shapes, the declaration switches from int N; to
  dim3 threadsPerBlock(N, N); (structure with 3 members, x, y, z, of type int)

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

Single Thread Block Matrix Addition

```
#define N 30

// kernel definition;
__global__ void MatAdd(float **A, float **B, float **C)
{
    int i, j;
    i = threadIdx.x;
    j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    int numBlocks;
    dim3 threadsPerBlock;
    // kernel invocation with one block of N * N threads
    numBlocks = 1;
    threadsPerBlock.x = N;
    threadsPerBlock.y = N;
    MatAdd <<< numBlocks, threadsPerBlock >>> (A, B, C);
    ...
}
```

→ https://tinyurl.com/cudafordummies/i/l1/single_thread_block_matrix_addition.cu
→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

CUDA — Basic Design Principles cont.

CUDA C-Programming Guide, Programming Model cont.

Single Thread Block Matrix Addition

2D thread block initialization

```
#define N 30

// kernel definition;
__global__ void MatAdd(float **A, float **B, float **C)
{
    int i, j;
    i = threadIdx.x;
    j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    int numBlocks;
    dim3 threadsPerBlock;
    // kernel invocation with one block of N * N threads
    numBlocks = 1;
    threadsPerBlock.x = N;
    threadsPerBlock.y = N;
    MatAdd <<< numBlocks, threadsPerBlock >>> (A, B, C);
    ...
}
```

→ https://tinyurl.com/cudafordummies/i/l1/single_thread_block_matrix_addition.cu
→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

CUDA — Basic Design Principles cont.

CUDA C-Programming Guide, Programming Model cont.

Single Thread Block Matrix Addition

```
#define N 30

// kernel definition;
__global__ void MatAdd(float **A, float **B, float **C)
{
    int i, j;
    i = threadIdx.x;
    j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    int numBlocks;
    dim3 threadsPerBlock;
    // kernel invocation with one block of N * N threads
    numBlocks = 1;
    threadsPerBlock.x = N;
    threadsPerBlock.y = N;
    MatAdd <<< numBlocks, threadsPerBlock >>> (A, B, C);
    ...
}
```

2D thread block initialization

goes directly into kernel execution configuration

→ https://tinyurl.com/cudafordummies/i/l1/single_thread_block_matrix_addition.cu

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

CUDA — Basic Design Principles cont.

CUDA C-Programming Guide, Programming Model cont.

Single Thread Block Matrix Addition

```
#define N 30

// kernel definition;
__global__ void MatAdd(float **A, float **B, float **C)
{
    int i, j;
    i = threadIdx.x;
    j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    int numBlocks;
    dim3 threadsPerBlock;
    // kernel invocation with one block of N * N threads
    numBlocks = 1;
    threadsPerBlock.x = N;
    threadsPerBlock.y = N;
    MatAdd <<< numBlocks, threadsPerBlock >>> (A, B, C);
    ...
}
```

built-in 2D thread indices 0,1,2...

2D thread block initialization

goes directly into kernel execution configuration

→ https://tinyurl.com/cudafordummies/i/l1/single_thread_block_matrix_addition.cu

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

CUDA — Basic Design Principles cont.

CUDA C-Programming Guide, Programming Model cont.

- There is an upper limit to the number of threads in a thread block, e.g. ≈1024 for current GPUs, because all threads of a thread block are supposed to run on the same SM (streaming multiprocessor)

→ https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth

CUDA — Basic Design Principles cont.

CUDA C-Programming Guide, Programming Model cont.

- There is an upper limit to the number of threads in a thread block, e.g. $\approx$1024 for current GPUs, because all threads of a thread block are supposed to run on the same SM (streaming multiprocessor)
- A single SM typically contains 64-128 CUDA cores (INT32, FP32, FP64, TC)

→ https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth

CUDA — Basic Design Principles cont.

CUDA C-Programming Guide, Programming Model cont.

- There is an upper limit to the number of threads in a thread block, e.g. $\approx$1024 for current GPUs, because all threads of a thread block are supposed to run on the same SM (streaming multiprocessor)
- A single SM typically contains 64-128 CUDA cores (INT32, FP32, FP64, TC)
- Different GPU architectures vary in terms of numbers of SMs, e.g. V100 has 80 SMs, A40 has 84 SMs, A100 has 108 SMs, H100 has 128 SMs;

→ https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth

CUDA — Basic Design Principles cont.

CUDA C-Programming Guide, Programming Model cont.

- There is an upper limit to the number of threads in a thread block, e.g. $\approx$1024 for current GPUs, because all threads of a thread block are supposed to run on the same SM (streaming multiprocessor)
- A single SM typically contains 64-128 CUDA cores (INT32, FP32, FP64, TC)
- Different GPU architectures vary in terms of numbers of SMs, e.g. V100 has 80 SMs, A40 has 84 SMs, A100 has 108 SMs, H100 has 128 SMs;
- However, multiple thread blocks can be launched in parallel as defined by the initial parameter numBlocks used in the kernel execution configuration <<<numBlocks,threadsPerBlock>>>

→ https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth

CUDA — Basic Design Principles cont.

How to Determine Max #Threads and Related

```
cuda-zen sh@n3073-004:~$  which nvcc
cuda-zen sh@n3073-004:~$  /gpfs/opt/sw/cuda-zen/spack-0.19.0/opt/spack/linux-almalinux8-zen/gcc-9.5.0/cuda-11.8.0-ananl33ltrpp33xetc
oltkbbbfuxoeez/extras/demo_suite/deviceQuery


./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 2 CUDA Capable device(s)

Device 0: "NVIDIA A100-PCIE-40GB"
  CUDA Driver Version / Runtime Version          12.3 / 11.8
  CUDA Capability Major/Minor version number:    8.0
  Total amount of global memory:                 40338 MBytes (42297786368 bytes)
  (108) Multiprocessors, ( 64) CUDA Cores/MP:     6912 CUDA Cores
  GPU Max Clock rate:                            1410 MHz (1.41 GHz)
  Memory Clock rate:                             1215 Mhz
  Memory Bus Width:                              5120-bit
  L2 Cache Size:                                 41943040 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
```

CUDA — Basic Design Principles cont.

How to Determine Max #Threads and Related

```
cuda–zen sh@n3073–004:~$  which nvcc
cuda–zen sh@n3073–004:~$  /gpfs/opt/sw/cuda-zen/spack-0.19.0/opt/spack/linux-almalinux8-zen/gcc-9.5.0/cuda-11.8.0-ananl33ltrpp33xetc
oltkbbbfuxoeez/extras/demo_suite/deviceQuery


./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 2 CUDA Capable device(s)

Device 0: "NVIDIA A100-PCIE-40GB"
  CUDA Driver Version / Runtime Version          12.3 / 11.8
  CUDA Capability Major/Minor version number:    8.0
  Total amount of global memory:                 40338 MBytes (42297786368 bytes)
  (108) Multiprocessors, ( 64) CUDA Cores/MP:     6912 CUDA Cores
  GPU Max Clock rate:                            1410 MHz (1.41 GHz)
  Memory Clock rate:                             1215 Mhz
  Memory Bus Width:                              5120-bit
  L2 Cache Size:                                 41943040 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
```

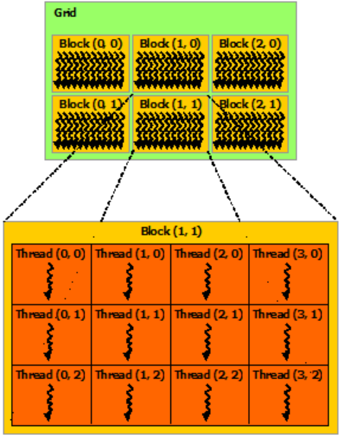
CUDA — Basic Design Principles cont.

How to Determine Max #Threads and Related cont.

```
./deviceQuery Starting...
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 3 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >
> Peer access from NVIDIA A100-PCIE-40GB (GPU0) -> NVIDIA A100-PCIE-40GB (GPU1) : Yes
> Peer access from NVIDIA A100-PCIE-40GB (GPU1) -> NVIDIA A100-PCIE-40GB (GPU0) : Yes
deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 12.3, CUDA Runtime Version = 11.8, NumDevs = 2
Result = PASS
```

CUDA — Basic Design Principles cont.

CUDA C-Programming Guide, Programming Model cont.

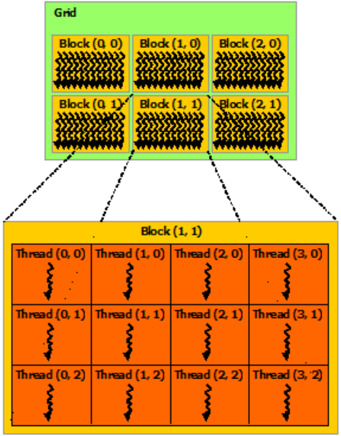


- numBlocks organization of the block grid is similar to threadsPerBlock

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

CUDA — Basic Design Principles cont.

CUDA C-Programming Guide, Programming Model cont.

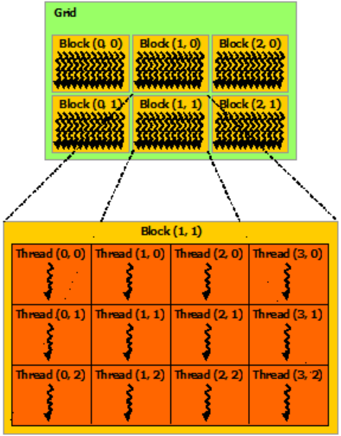


- numBlocks organization of the block grid is similar to threadsPerBlock
- Can again be one-dimensional, two-dimensional or three-dimensional

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

CUDA — Basic Design Principles cont.

CUDA C-Programming Guide, Programming Model cont.

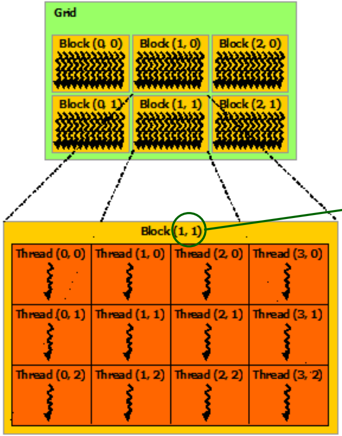


- numBlocks organization of the block grid is similar to threadsPerBlock
- Can again be one-dimensional, two-dimensional or three-dimensional
- Again dim3 declaration

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

CUDA — Basic Design Principles cont.

CUDA C-Programming Guide, Programming Model cont.

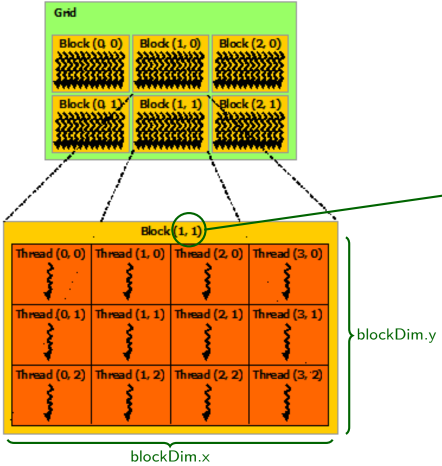


- numBlocks organization of the block grid is similar to threadsPerBlock
- Can again be one-dimensional, two-dimensional or three-dimensional
- Again dim3 declaration
- blockIdx.[x,y,z] is again a built-in variable at the kernel level to identify corresponding thread blocks for each of the parallel threads

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

CUDA — Basic Design Principles cont.

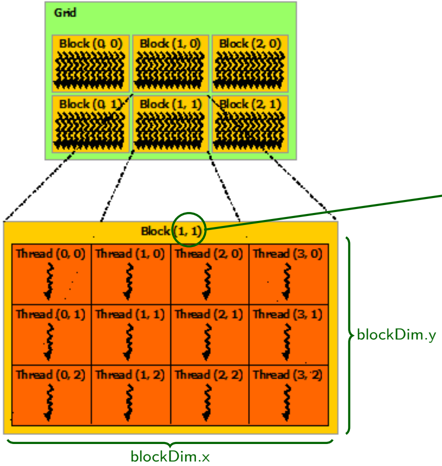
CUDA C-Programming Guide, Programming Model cont.



- numBlocks organization of the block grid is similar to threadsPerBlock
- Can again be one-dimensional, two-dimensional or three-dimensional
- Again dim3 declaration
- blockIdx.[x,y,z] is again a built-in variable at the kernel level to identify corresponding thread blocks for each of the parallel threads
- blockDim.[x,y,z] is another built-in variable for the kernel to asses thread block dimensions

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

ASC Austrian Scientific Computing



- numBlocks organization of the block grid is similar to threadsPerBlock
- Can again be one-dimensional, two-dimensional or three-dimensional
- Again dim3 declaration
- blockIdx.[x,y,z] is again a built-in variable at the kernel level to identify corresponding thread blocks for each of the parallel threads
- blockDim.[x,y,z] is another built-in variable for the kernel to asses thread block dimensions
- i = (blockIdx.x * blockDim.x) + threadIdx.x is the most common use case

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

Multiple Thread Blocks Matrix Addition

```
#define N 256;

// kernel definition;
__global__ void MatAdd(float **A, float **B, float **C)
{
    int i, j;
    i = (blockIdx.x * blockDim.x) + threadIdx.x;
    j = (blockIdx.y * blockDim.y) + threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}
int main()
{
    dim3 threadsPerBlock, numBlocks;
    ...
    // kernel invocation with blocks of 256 threads
    threadsPerBlock.x = 16;
    threadsPerBlock.y = 16;
    numBlocks.x = N / threadsPerBlock.x;
    numBlocks.y = N / threadsPerBlock.y;
    MatAdd <<< numBlocks, threadsPerBlock >>> (A, B, C);
    ...
}
```

→ https://tinyurl.com/cudafordummies/i/l1/multiple_thread_blocks_matrix_addition.cu

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

Multiple Thread Blocks Matrix Addition

```
#define N 256;

// kernel definition;
__global__ void MatAdd(float **A, float **B, float **C)
{
    int i, j;
    i = (blockIdx.x * blockDim.x) + threadIdx.x;
    j = (blockIdx.y * blockDim.y) + threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}
int main()
{
    dim3 threadsPerBlock, numBlocks;
    ...
    // kernel invocation with blocks of 256 threads
    threadsPerBlock.x = 16;
    threadsPerBlock.y = 16;
    numBlocks.x = N / threadsPerBlock.x;
    numBlocks.y = N / threadsPerBlock.y;
    MatAdd <<< numBlocks, threadsPerBlock >>> (A, B, C);
    ...
}
```

2D initial-izations

→ https://tinyurl.com/cudafordummies/i/l1/multiple_thread_blocks_matrix_addition.cu

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

Multiple Thread Blocks Matrix Addition

```
#define N 256;

// kernel definition;
__global__ void MatAdd(float **A, float **B, float **C)
{
    int i, j;
    i = (blockIdx.x * blockDim.x) + threadIdx.x;
    j = (blockIdx.y * blockDim.y) + threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}
int main()
{
    dim3 threadsPerBlock, numBlocks;
    ...
    // kernel invocation with blocks of 256 threads
    threadsPerBlock.x = 16;
    threadsPerBlock.y = 16;
    numBlocks.x = N / threadsPerBlock.x;
    numBlocks.y = N / threadsPerBlock.y;
    MatAdd <<< numBlocks, threadsPerBlock >>> (A, B, C);
    ...
}
```

2D initial-izations

general type kernel exe-cution con-figuration

→ https://tinyurl.com/cudafordummies/i/l1/multiple_thread_blocks_matrix_addition.cu

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

Multiple Thread Blocks Matrix Addition

```
#define N 256;

// kernel definition;
__global__ void MatAdd(float **A, float **B, float **C)
{
    int i, j;
    i = (blockIdx.x * blockDim.x) + threadIdx.x;
    j = (blockIdx.y * blockDim.y) + threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}
int main()
{
    dim3 threadsPerBlock, numBlocks;
    ...
    // kernel invocation with blocks of 256 threads
    threadsPerBlock.x = 16;
    threadsPerBlock.y = 16;
    numBlocks.x = N / threadsPerBlock.x;
    numBlocks.y = N / threadsPerBlock.y;
    MatAdd <<< numBlocks, threadsPerBlock >>> (A, B, C);
    ...
}
```

general usage of built-in variables in 2D

2D initializations

general type kernel execution configuration

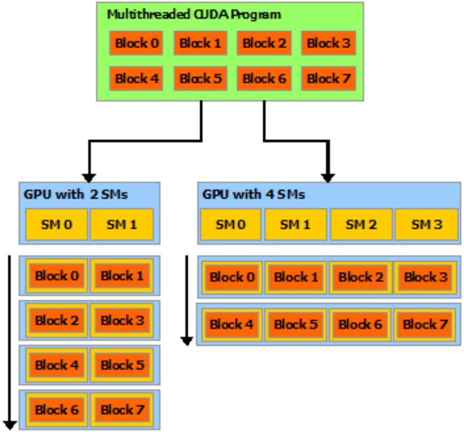
→ https://tinyurl.com/cudafordummies/i/l1/multiple_thread_blocks_matrix_addition.cu
→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

ASC Austrian Scientific Computing

CUDA — Basic Design Principles cont.

CUDA C-Programming Guide, Programming Model cont.

- 256 threads per thread block is arbitrary but a frequent choice

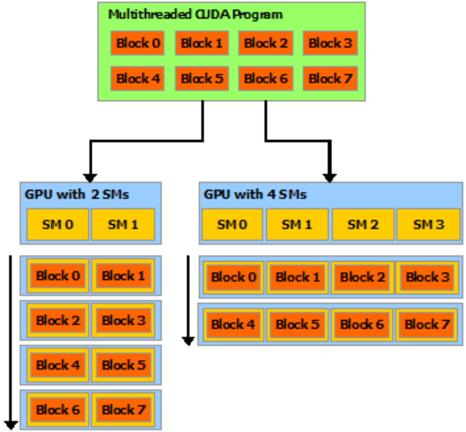


→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

CUDA — Basic Design Principles cont.

CUDA C-Programming Guide, Programming Model cont.

- 256 threads per thread block is arbitrary but a frequent choice
- Thread blocks are required to execute independently in any order

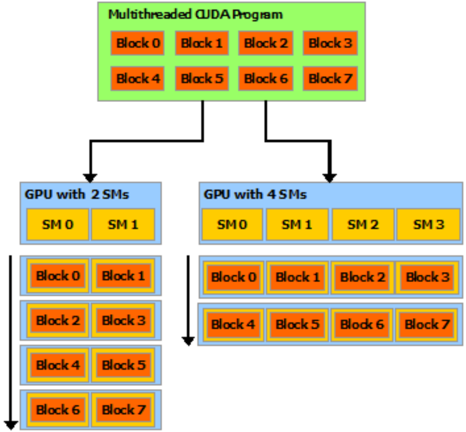


→ https://docs.nvidia.com/cuda/cuda-c-programming-guide

CUDA — Basic Design Principles cont.

CUDA C-Programming Guide, Programming Model cont.

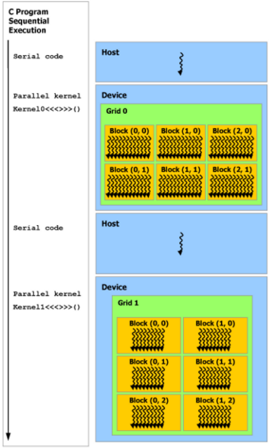
- 256 threads per thread block is arbitrary but a frequent choice
- Thread blocks are required to execute independently in any order
- Scalability results from this requirement



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

CUDA — Basic Design Principles cont.

CUDA C-Programming Guide, Programming Model cont.

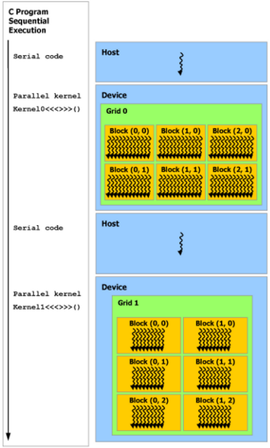


- Workflow is devided between host and device

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

CUDA — Basic Design Principles cont.

CUDA C-Programming Guide, Programming Model cont.

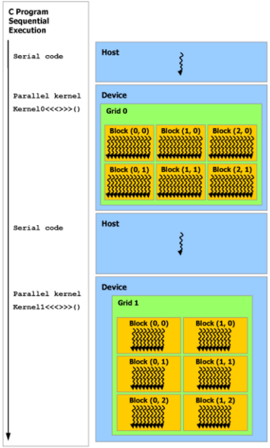


- Workflow is devided between host and device
- CUDA threads execute on the GPU the rest of the program on the host CPU

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

CUDA — Basic Design Principles cont.

CUDA C-Programming Guide, Programming Model cont.

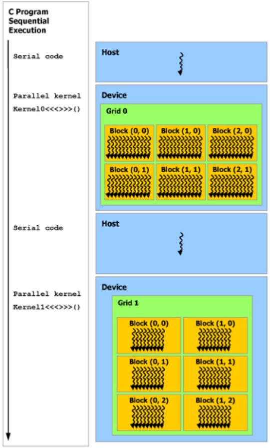


- Workflow is devided between host and device
- CUDA threads execute on the GPU the rest of the program on the host CPU
- Thread blocks are all parallel

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

CUDA — Basic Design Principles cont.

CUDA C-Programming Guide, Programming Model cont.

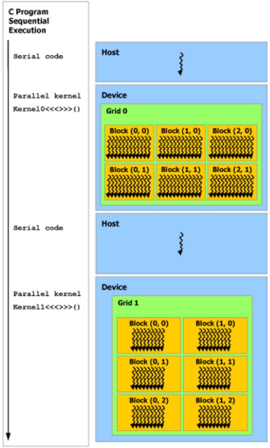


- Workflow is devided between host and device
- CUDA threads execute on the GPU the rest of the program on the host CPU
- Thread blocks are all parallel
- Host code is usually serial, both sections may execute concurrently

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

CUDA — Basic Design Principles cont.

CUDA C-Programming Guide, Programming Model cont.



- Workflow is devided between host and device
- CUDA threads execute on the GPU the rest of the program on the host CPU
- Thread blocks are all parallel
- Host code is usually serial, both sections may execute concurrently
- If subsequent host sections are dependent on kernel results, we need to insert cudaDeviceSynchronize(); after kernel call

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

Take Home Messages

- GPU kernels need to account for proper logic

Take Home Messages

- GPU kernels need to account for proper logic
- Kernel execution configuration facilitates efficient operation of GPU resources

Take Home Messages

- GPU kernels need to account for proper logic
- Kernel execution configuration facilitates efficient operation of GPU resources
- Massive parallelism at the level of GPU threads replacing conventional loops over array elements with many individual threads directly acting on thread-specific data elements in parallel

Take Home Messages

- GPU kernels need to account for proper logic
- Kernel execution configuration facilitates efficient operation of GPU resources
- Massive parallelism at the level of GPU threads replacing conventional loops over array elements with many individual threads directly acting on thread-specific data elements in parallel
- Built-in variables to quasi-automatize various workloads, e.g. threadIdx, blockIdx 0,1,2,3...

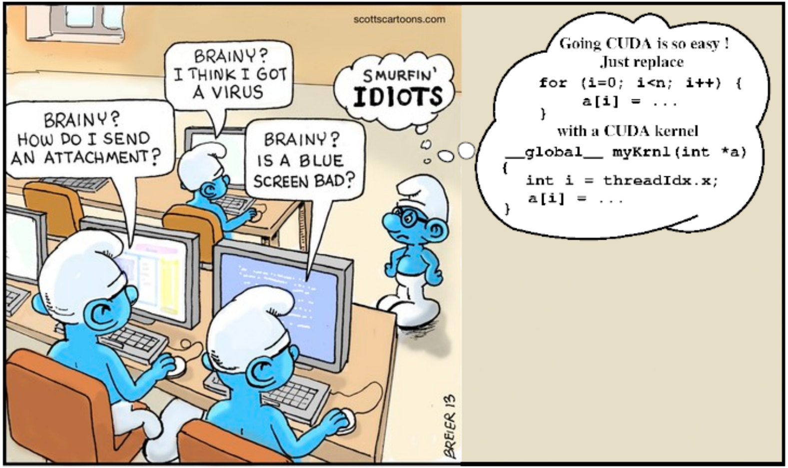
Take Home Messages

- GPU kernels need to account for proper logic
- Kernel execution configuration facilitates efficient operation of GPU resources
- Massive parallelism at the level of GPU threads replacing conventional loops over array elements with many individual threads directly acting on thread-specific data elements in parallel
- Built-in variables to quasi-automatize various workloads, e.g. threadIdx, blockIdx 0,1,2,3...
- Shapes of thread blocks go hand in hand with domain decomposition (vector, matrix, volume)

Take Home Messages

- GPU kernels need to account for proper logic
- Kernel execution configuration facilitates efficient operation of GPU resources
- Massive parallelism at the level of GPU threads replacing conventional loops over array elements with many individual threads directly acting on thread-specific data elements in parallel
- Built-in variables to quasi-automatize various workloads, e.g. threadIdx, blockIdx 0,1,2,3...
- Shapes of thread blocks go hand in hand with domain decomposition (vector, matrix, volume)
- GPU computing means eco-friendly computing !

Take Home Messages cont.



IT Smurf