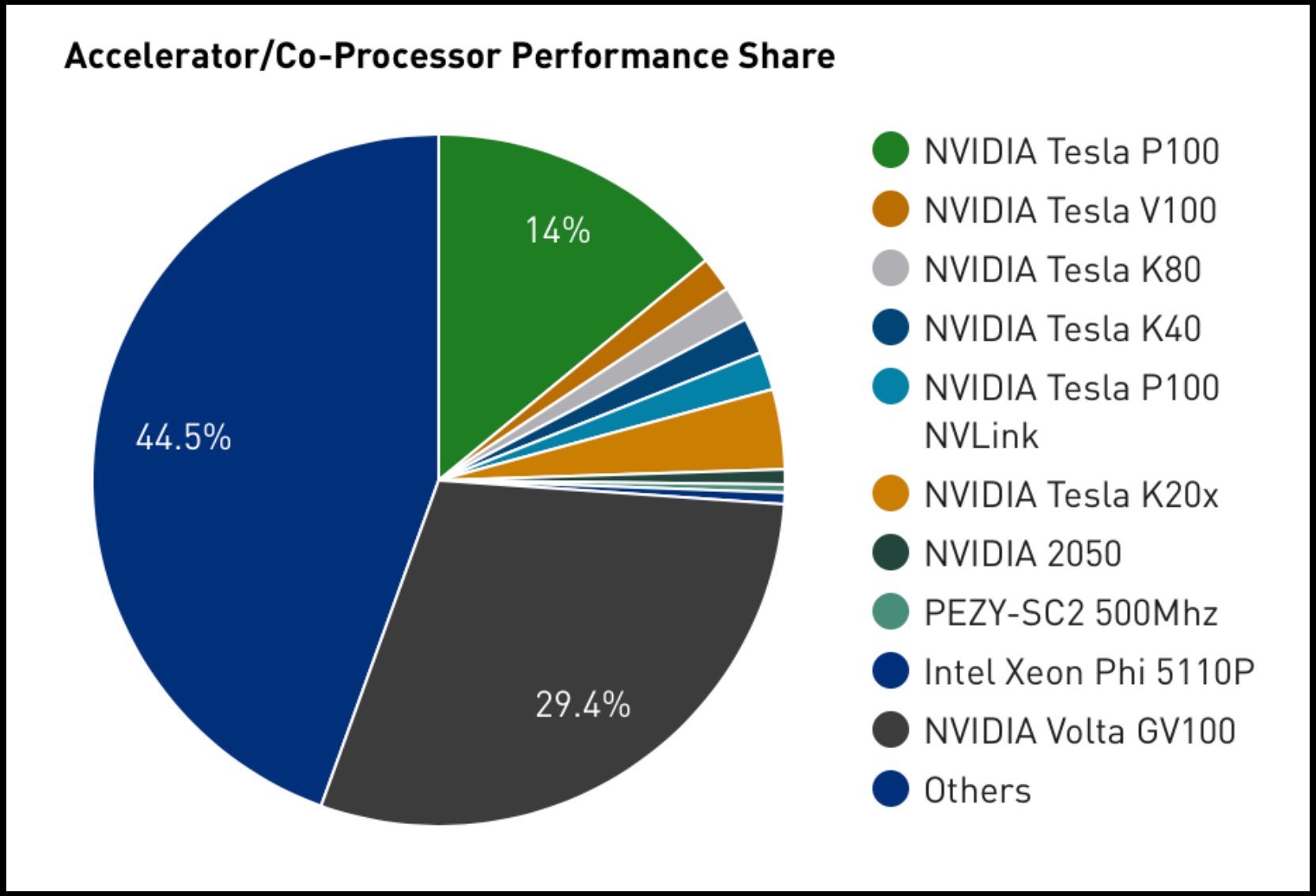
Introduction to CUDA Programming (Compute Unified Device Architecture)

Jongsoo Kim
Korea Astronomy and Space Science Institute
@COMAC 2018 Workshop

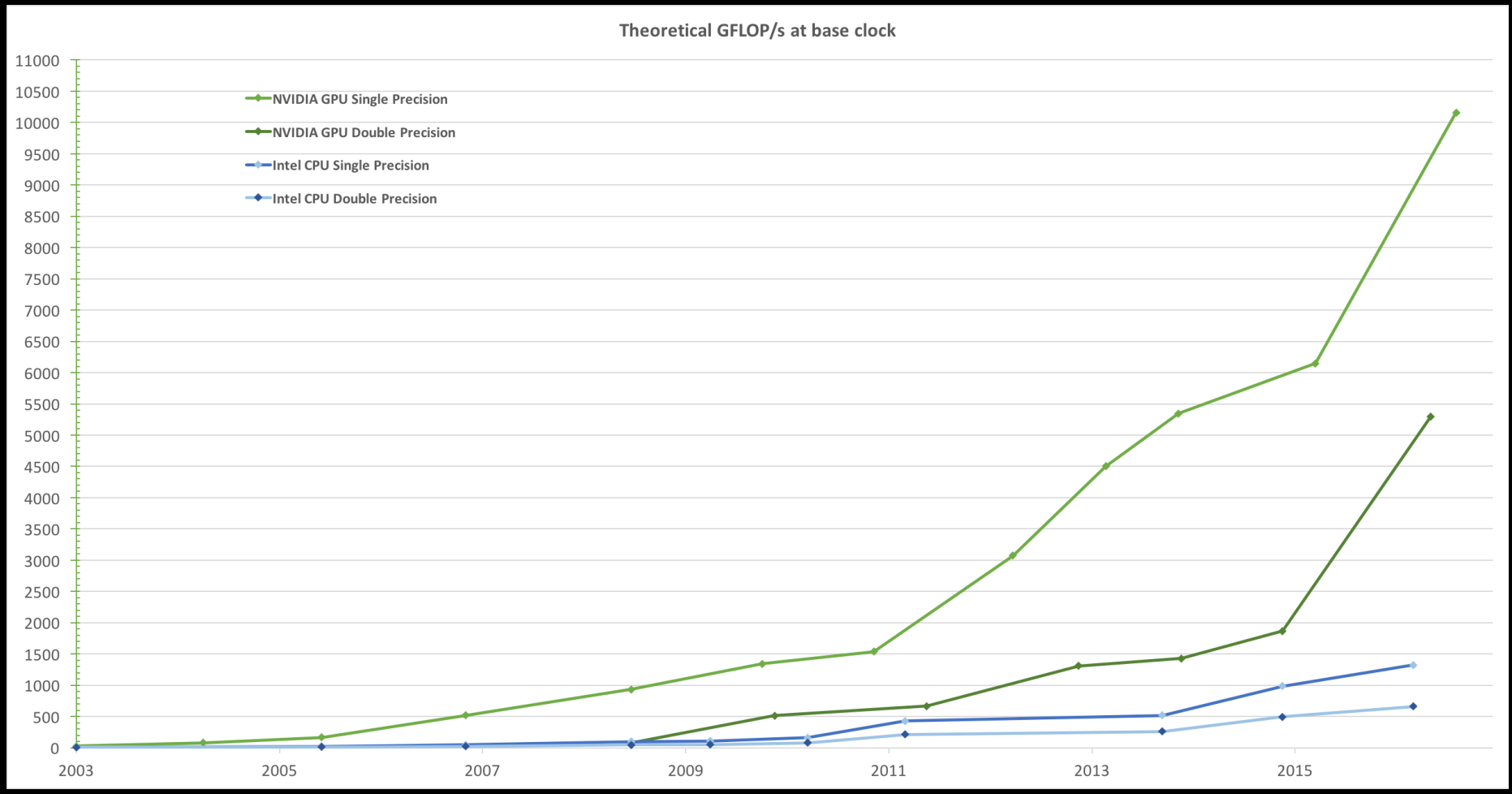
www.top500.org

Summit #1, Linpack: 122.3 Pflos/s
4356 nodes, 8.8MW
two 22-core Power 9 CPU+six NVIDA Tesla V100 GPUs

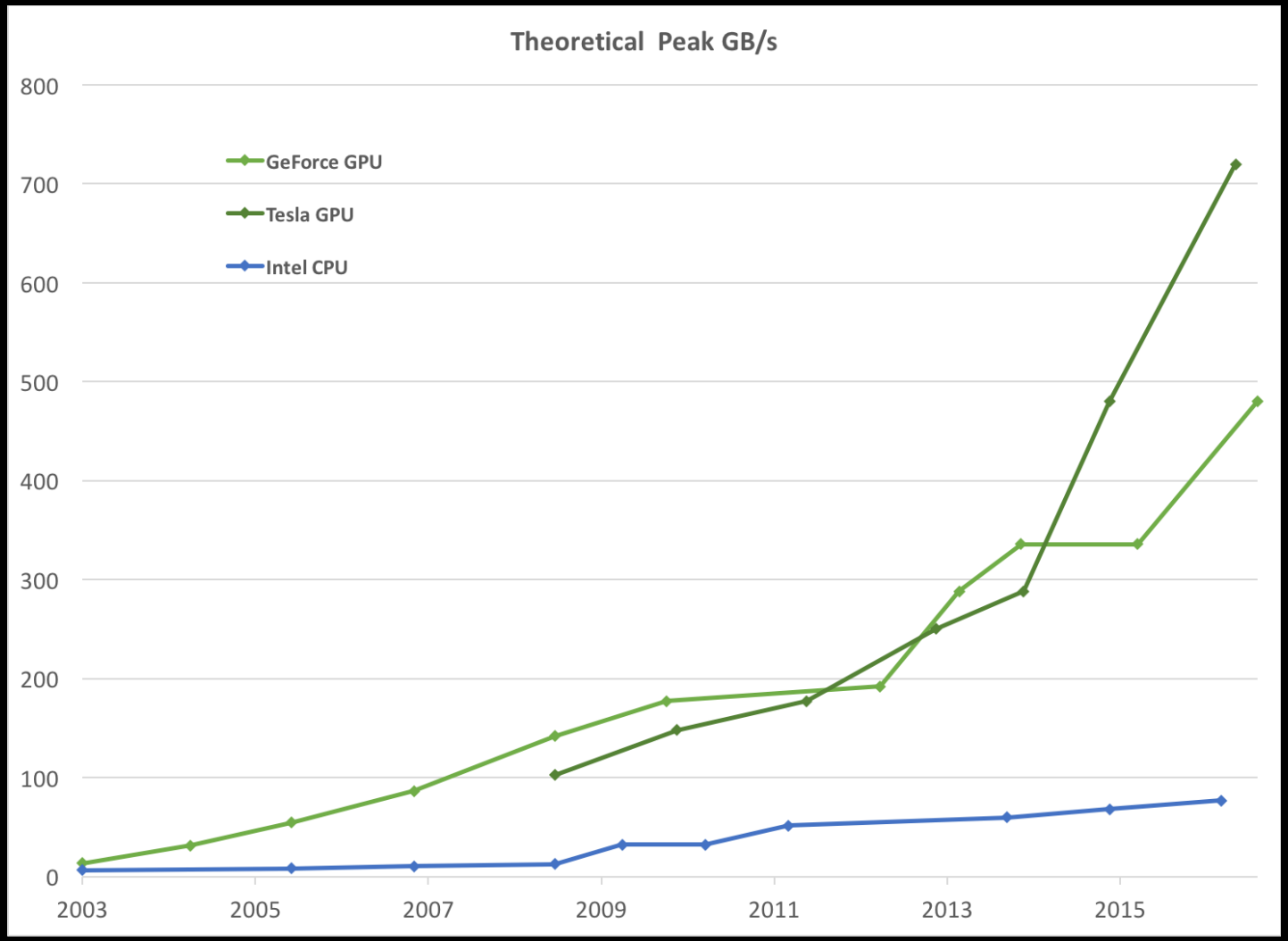
www.top500.org, Jun 20158



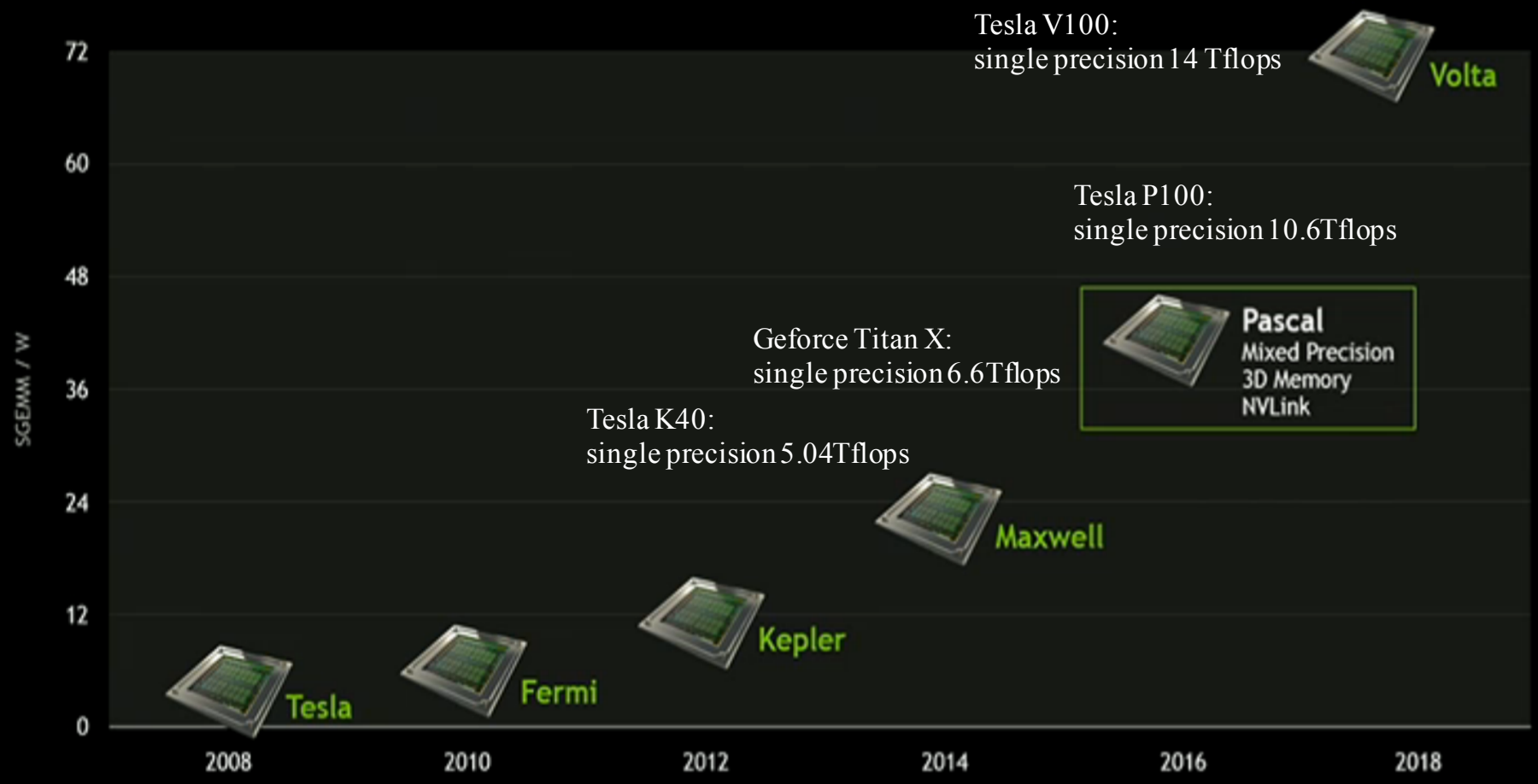
Performance, GFLOPS/s



Memory bandwidth, GB/s

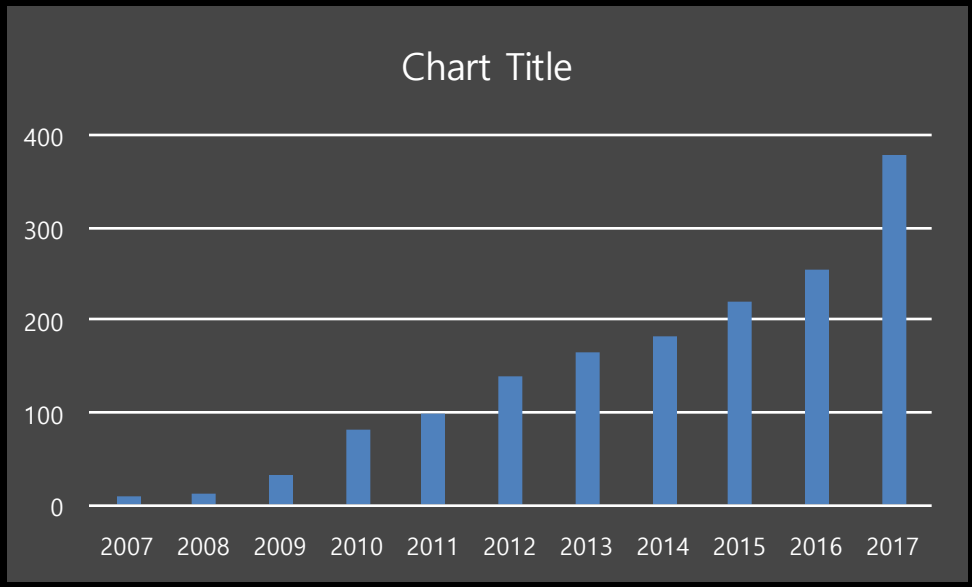


NVIDIA GPU Roadmap

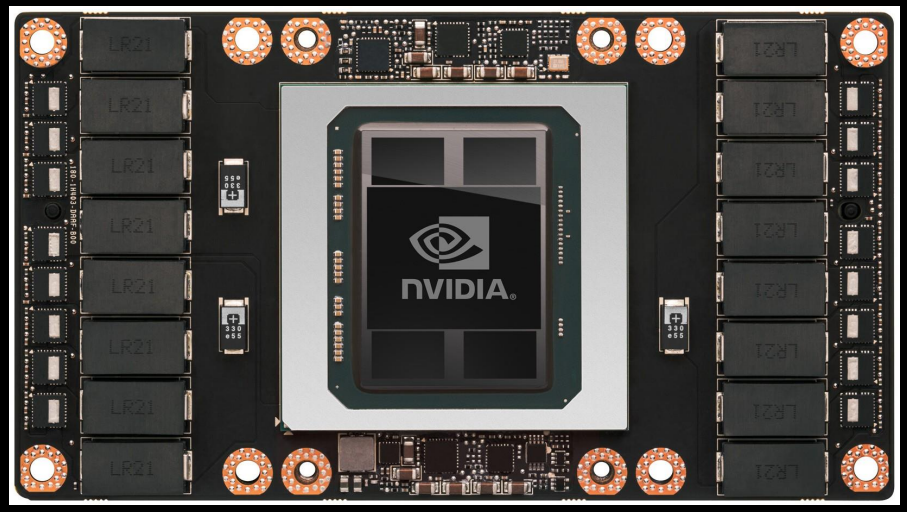


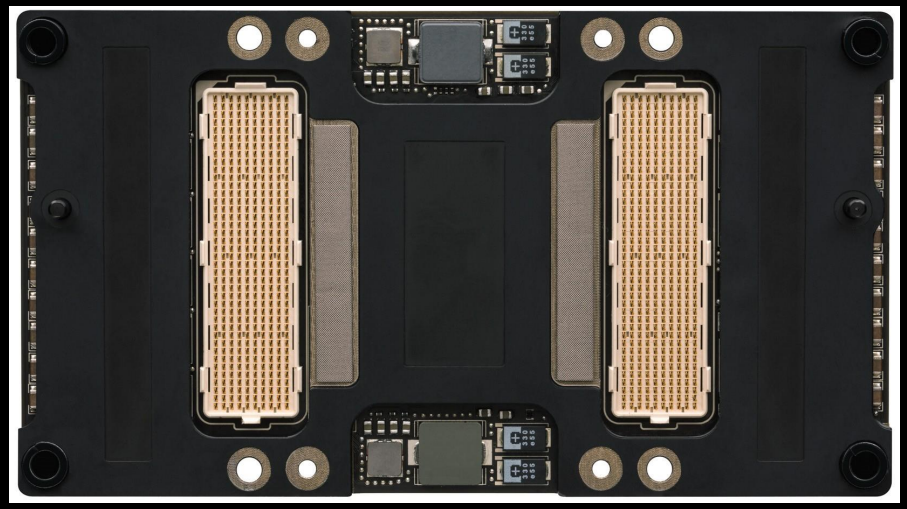
Astronomy Applications of GPU

- N-body simulations
- Fluid HD and MHD simulations
- Radiative Transfer
- Data processing, e.g., radio astronomy
- etc…
- ADS (search "GPU" in abstract)
  - 10 papers in 2007
  - 13 papers in 2008
  - 33 papers in 2009
  - 81 papers in 2010
  - 99 papers in 2011
  - 140 papers in 2012
  - 164 papers in 2013
  - 182 papers in 2014
  - 221 papers in 2015
  - 254 papers in 2016
  - 379 papers in 2017



NVIDA TESLA V100 SXM2



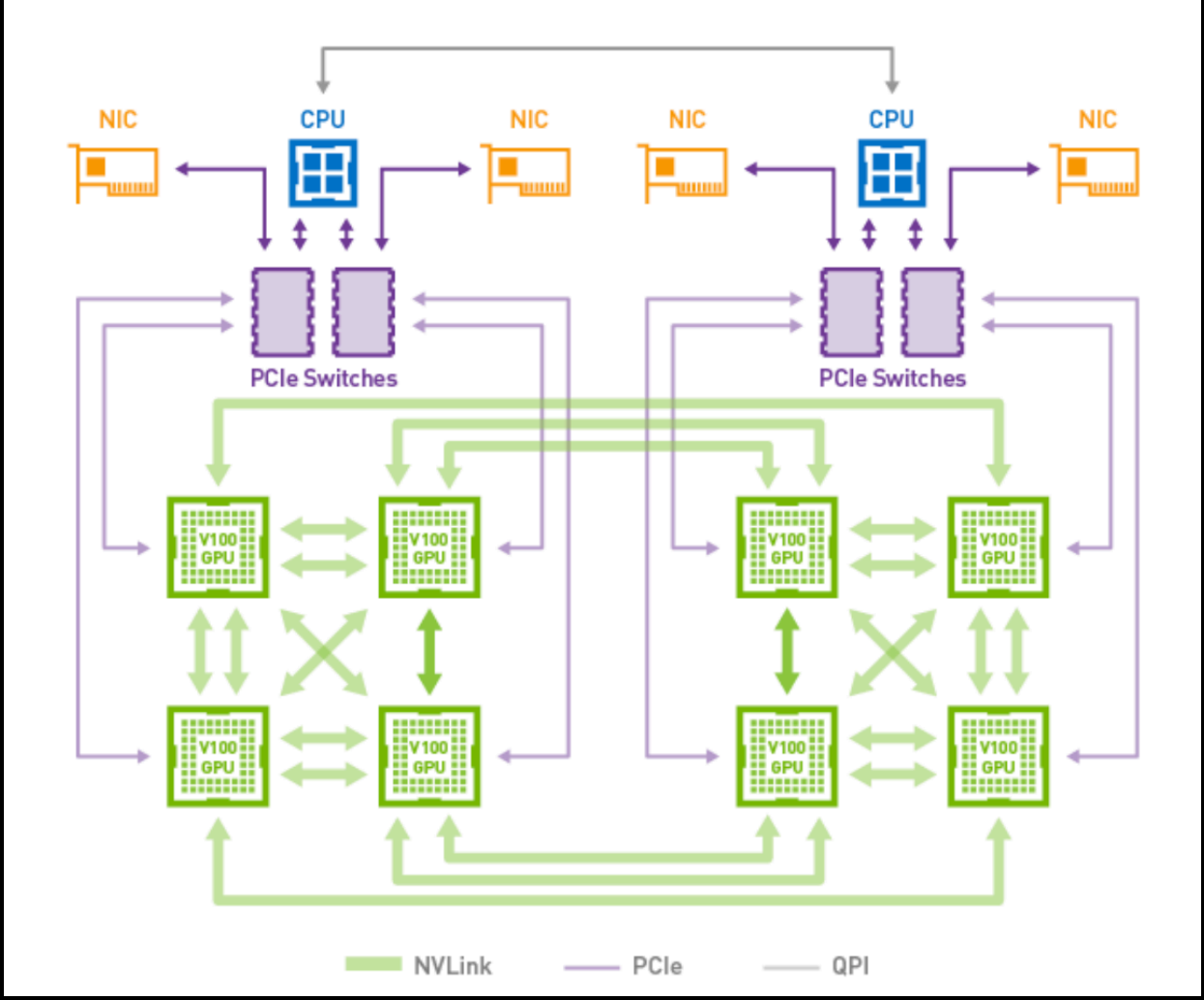


- CUD A, Tensor Cores: 5140+640
- NVLink: 300 GB/s
- HBM2: 900GB/s
- Unified Memory

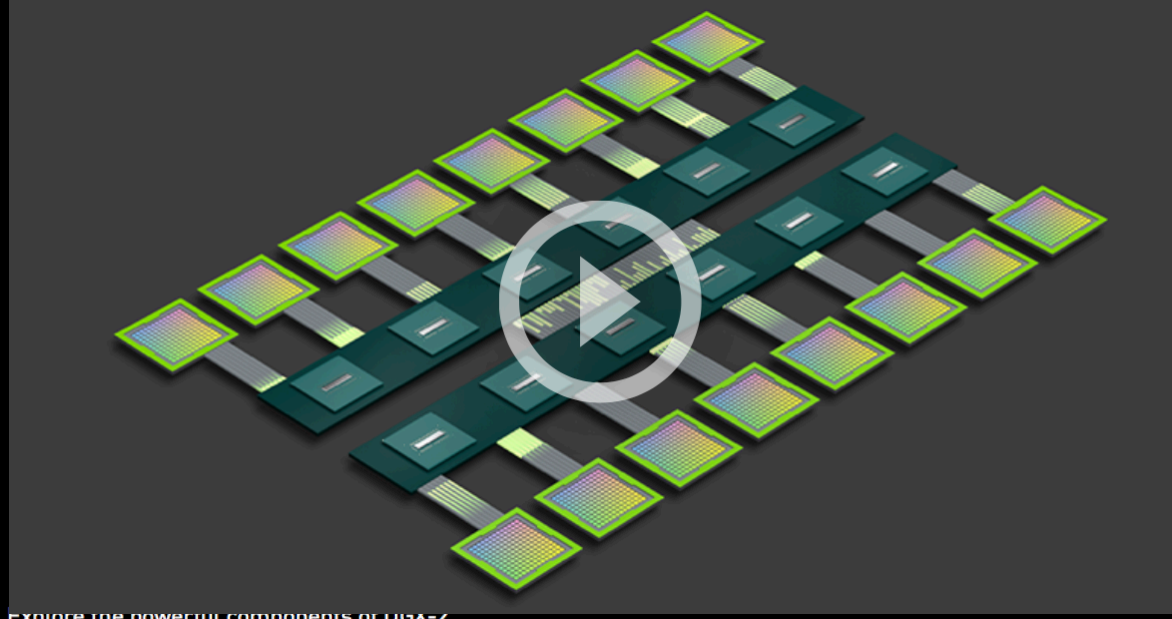
- 7.8 TFLOPS of FP64
- 15.7 TFLOPS of FP32
- 125 TFLOPS of Tensor

NVLink:

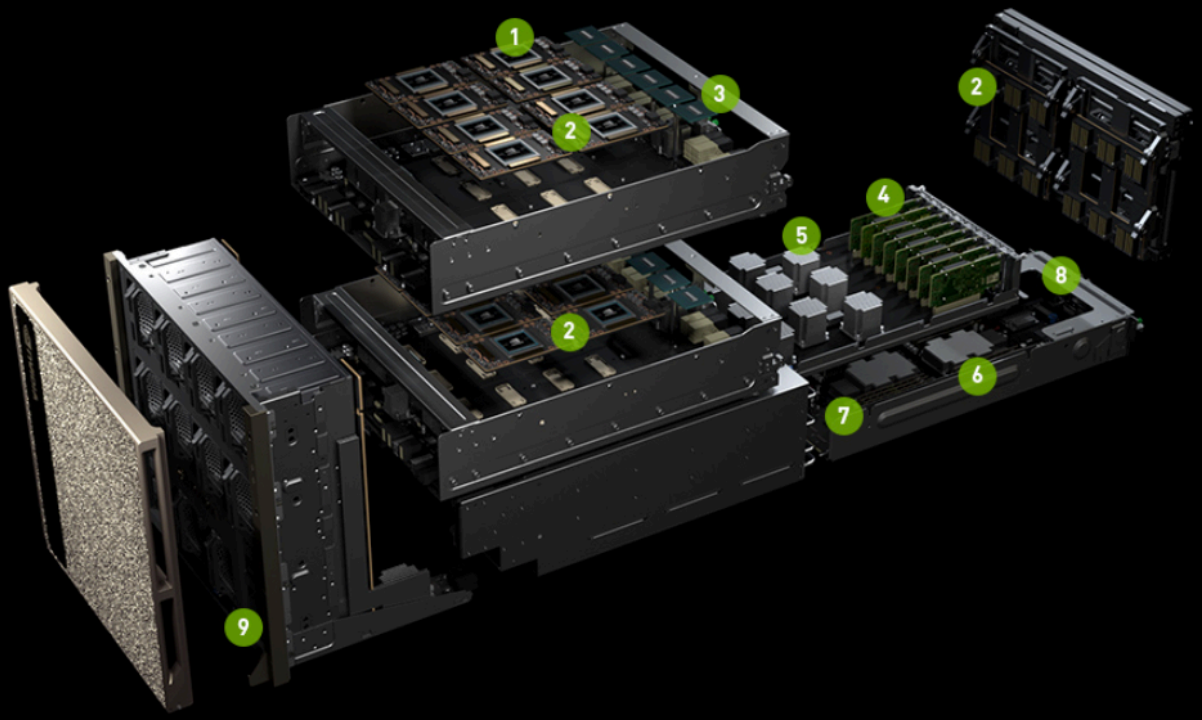
50 GB/sec bi-direction per connection
300GB/sec for six connections per GPU



NVIDIA D GX-2

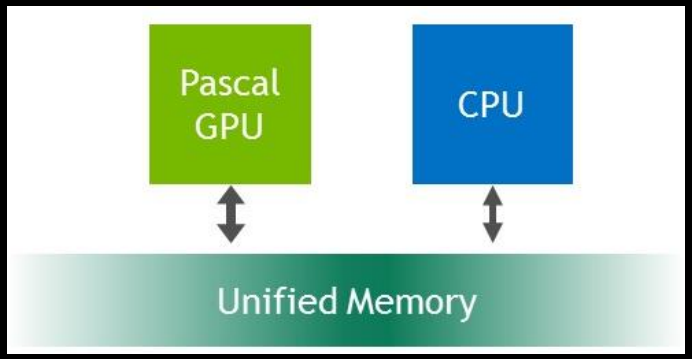


Explore the powerful components of DGX-2.



1. NVIDIA TESLA V100 32GB, SXM3
2. 16 TOTAL GPUS FOR BOTH BOARDS, 512GB TOTAL HBM2 MEMORY
   Each GPU board with 8 NVIDIA Tesla V100.
3. 12 TOTAL NVSWITCHES
   High Speed Interconnect, 2.4 TB/sec bisection bandwidth.
4. 8 EDR INFINIBAND/100 GbE ETHERNET
   1600 Gb/sec Bi-directional Bandwidth and Low-Latency.
5. PCIE SWITCH COMPLEX
6. TWO INTEL XEON PLATINUM CPUS
7. 1.5 TB SYSTEM MEMORY
8. DUAL 10/25 GbE ETHERNET
9. 30 TB NVME SSDS INTERNAL STORAGE

Pascal Unified Memory



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);
}
```

Pascal Unified Memory*

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

  fread(data, 1, N, fp);

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

  use_data(data);

  free(data);
}
```

*with operating system support

Contents

- GPGPU and Tesla GPU cards
- CUDA exercises
  - Execution configuration (hello world)
  - Global Memory (vector sum)
  - Shared Memory (dot product, matrix multiplication)
  - Texture Memory (heat equation)
  - Constant Memory (??)
  - Unified Virtual Memory (cuda6)
  - cufft library

Languages for GPGPU

- NVIDIA CUDA C/C++
- OpenCL
- PGI CUDA Fortran
- OpenACC Directives
- PyCUDA

Libraries

- cuFFT (Fast Fourier Transforms)
- cuBLAS (Basic Linear Algebra Subroutines)
- cuSPARSE (Sparse Matrix Routines)
- cuSOLVER (Dense and Sparse Direct Solvers)
- NPP (Image & Video Processing Primitivies)
- CUDA Math Library
- Thrust (Templated Parallel Algorithms & Data Structures)

Processing flow

Bottlenecks

- PCIe bus: 16GB/s
- Memory BW:

148 GB/s for Tesla S2050

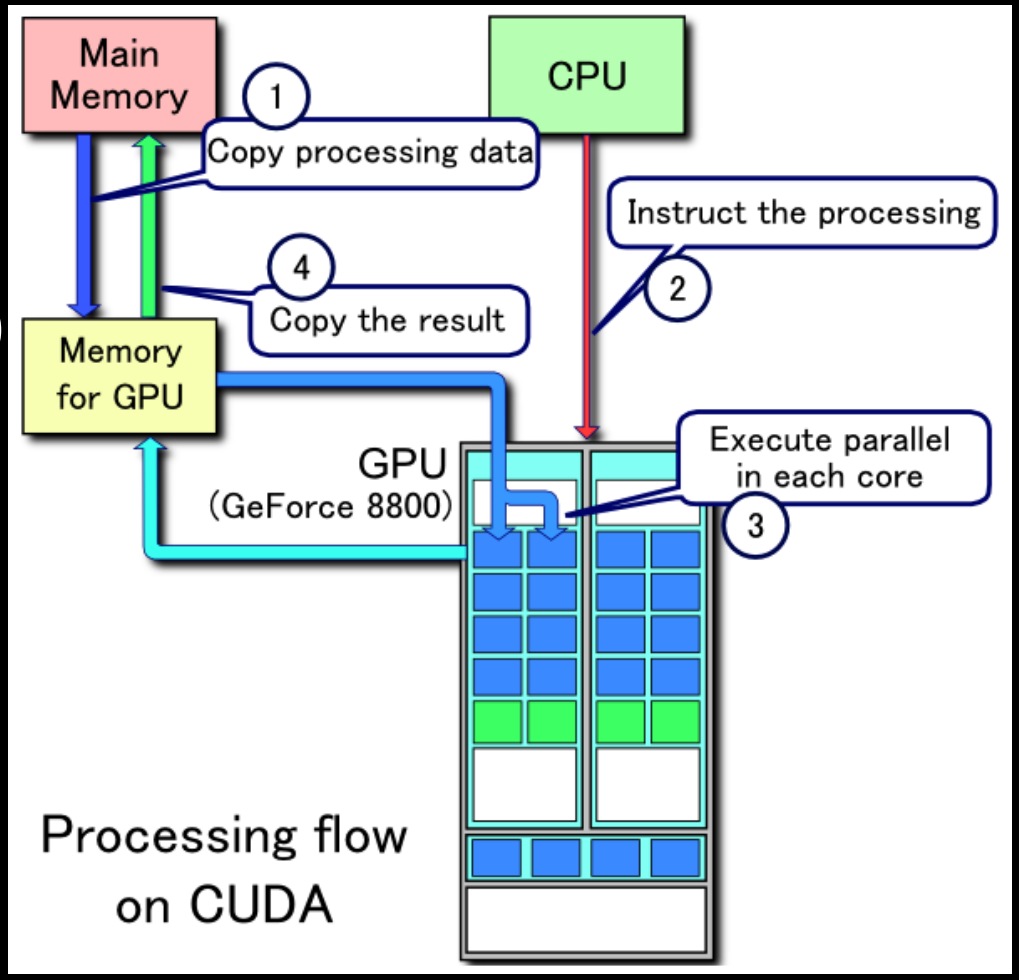
288 GB/s for Tesla K 40

336.5 GB/s for Titan X

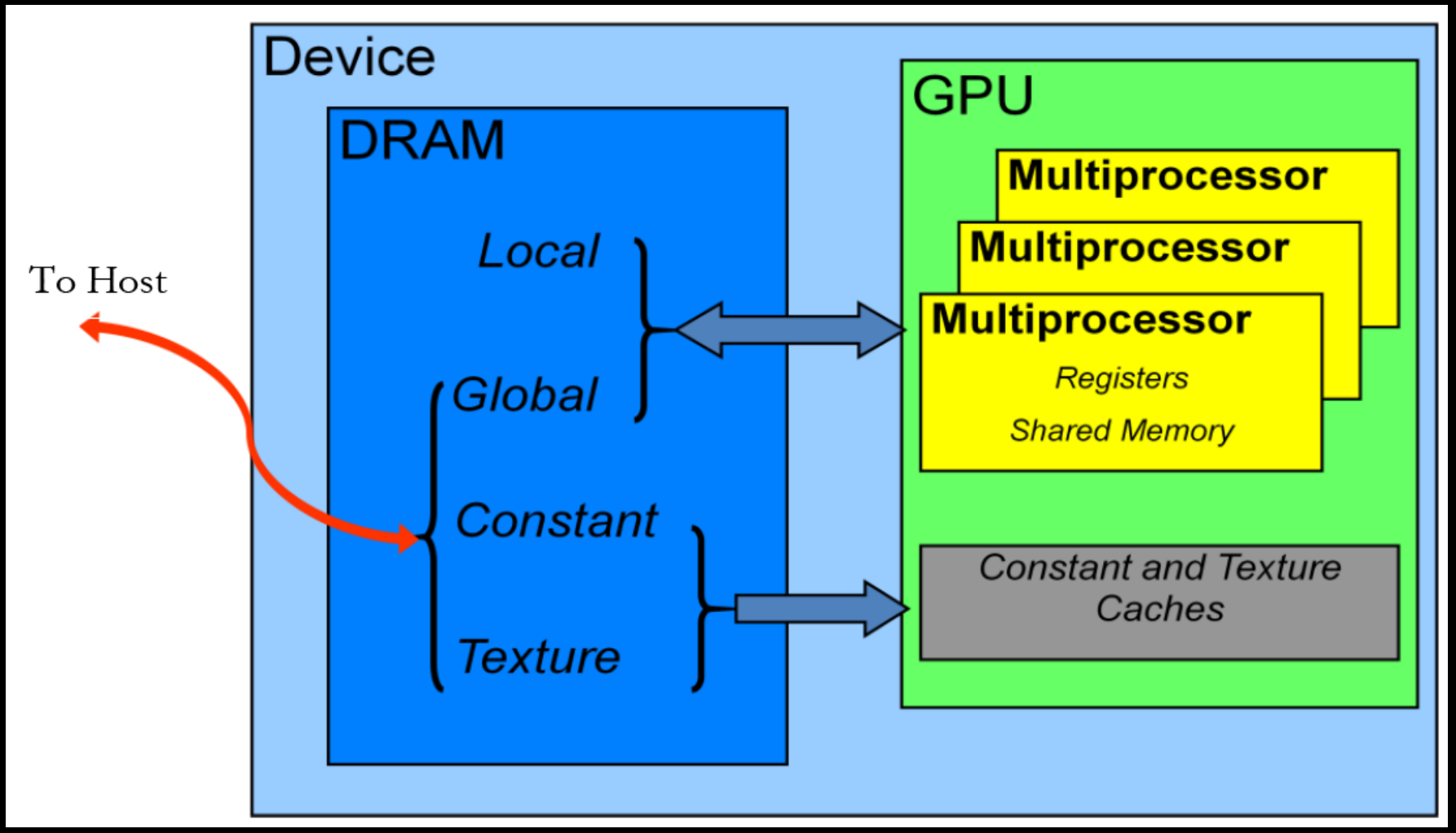
900GB/s for V100

- Memory latency:

400-800 clock cycles



Device Memory Space



Access to a NARIT GPU server

- From outside of the NARIT

  ssh –p 22240 –Y guest@stargate.narit.space
- From inside of the NARIT

  ssh –Y pollux

  mkdir –p userid/cuda_tutorial
- cd userid/cuda_tutorial
- cp ~guest/jskim/cuda_tutorial/* .
-

cuda environment variables

- Include the following two lines in .bashrc
  - export PATH=/usr/local/cuda/bin:$PATH
  - export LD_LIBRARY_PATH=/usr/local/cuda/lib64

deviceQuery

- cd /home/guest
- ./usr/local/cuda/bin/cuda-install-samples-9.2.sh .
- /home/guest/NVIDIA_CUDA-9.2_Samples/1_Utilities/deviceQuery/deviceQuery

Demo of nbody

- cd /home/guest/NVIDIA_CUDA-9.2_Samples/5_Simulations/nbody
- ./nbody –benchmark -cpu
- ./nbody –benchmark –numdevices=1
- ./nbody –benchmark –numdevices=2

Exercise 1: Hello world

```
#include <stdio.h>

void hello_world(void) {
      printf(“Hello World₩n”);
}

int main (void) {
     hello_world();
     return 0;
}
```

Exercise 1: Hello world

```
#include <stdio.h>

__global__ void hello_world(void) {
    printf("Hello World₩n");
}

int main (void) {
    hello_world<<<1,5>>>();
    cudaDeviceReset();
    return 0;
}
```

Compile and Run

- nvcc hello.cu
- ./a.out

  Hello World
  Hello World
  Hello World
  Hello World
  Hello World

C Language Extensions

- Function Type Qualifiers

  __global__

  executed on the device (GPU)

  callable from the host (CPU) only

  functions should have void return type

  any call to a __global__ function must specify the execution configuration for that call

  __device__

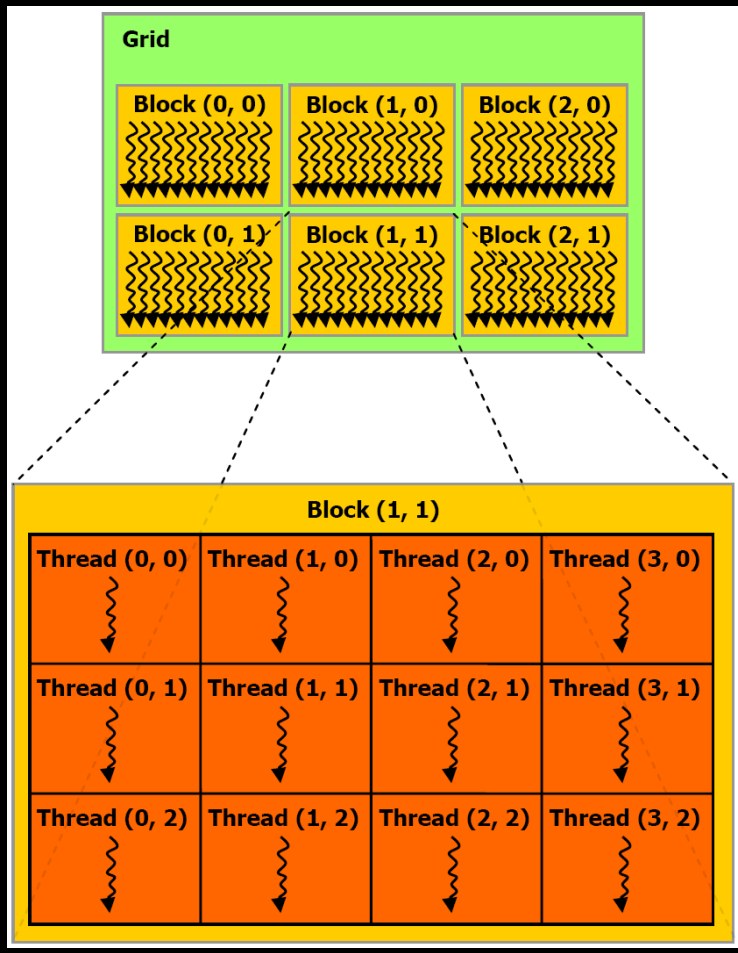
  executed on the device

  callable from the device only

  …

Grid, Block, Thread

- Tesla V100-SXM2-16GB
  - Maximum dimension size of a grid (2147483647, 65535, 65535)
  - Maximum dimension size of a block 1024x1024x64
  - max. # of threads per block 1024
  - Warp size: 32



- ~guest/NVIDIA_CUDA-9.2_Samples/1_Utilities/deviceQuery/deviceQuery
- ~guest/NVIDIA_CUDA-9.2_Samples/1_Utilities/bandwidthTest/bandwidthTest

C Language Extensions

- Execution configuration

```
<<<blocksPerGrid,threadsPerBlock>>>
<<<1,1>>>
<<<1024,1024>
```

```
dim3 blocksPerGrid(16,16,1)
dim3 threadsPerBlock(1024,1,1)
<<<blocksPerGrid,threadsPerBlock>>>
```

C Language Extensions

- Built-in Variables
  - blockIdx = (blockIdx.x, blockIdx.y, blockIdx.y)
    - three unsigned integers, uint3
  - threadIdx = (threadIdx.x, threadIdx.y, threadIdx.y)
    - three unsigned integers, uint3

- Built-in Vector types
  - dim3:
    - Integer vector type based on unit3
    - used to specify dimensions

Ex2: Execution Configuration

- nvcc exec_conf_1d.cu
- nvcc exec_conf_2d.cu
- ./a.out

Exercise 3: Vector sum

gcc vector_sum.c
nvcc vector_sum_block.cu
nvcc vector_sum_thread.cu
nvcc vector_sum_tb.cu
nvcc vector_sum_block.unified.cu

Exercise 3: Vector sum

```
#include <stdio.h>

const int N=128;

void add(int *a, int *b, int *c) {
    int i = 0;
    while (i < N) {
        c[i] = a[i] + b[i];
        i += 1;
    }
}
```

Exercise 3: Vector sum

```
int main (void) {
    int a[N], b[N], c[N];

    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }

    add (a, b, c);

    for (int i=0; i<N; i++) {
        printf("%d + %d = %d₩n", a[i],b[i],c[i]);
    }

    return 0;
}
```

Exercise 3: Vector sum

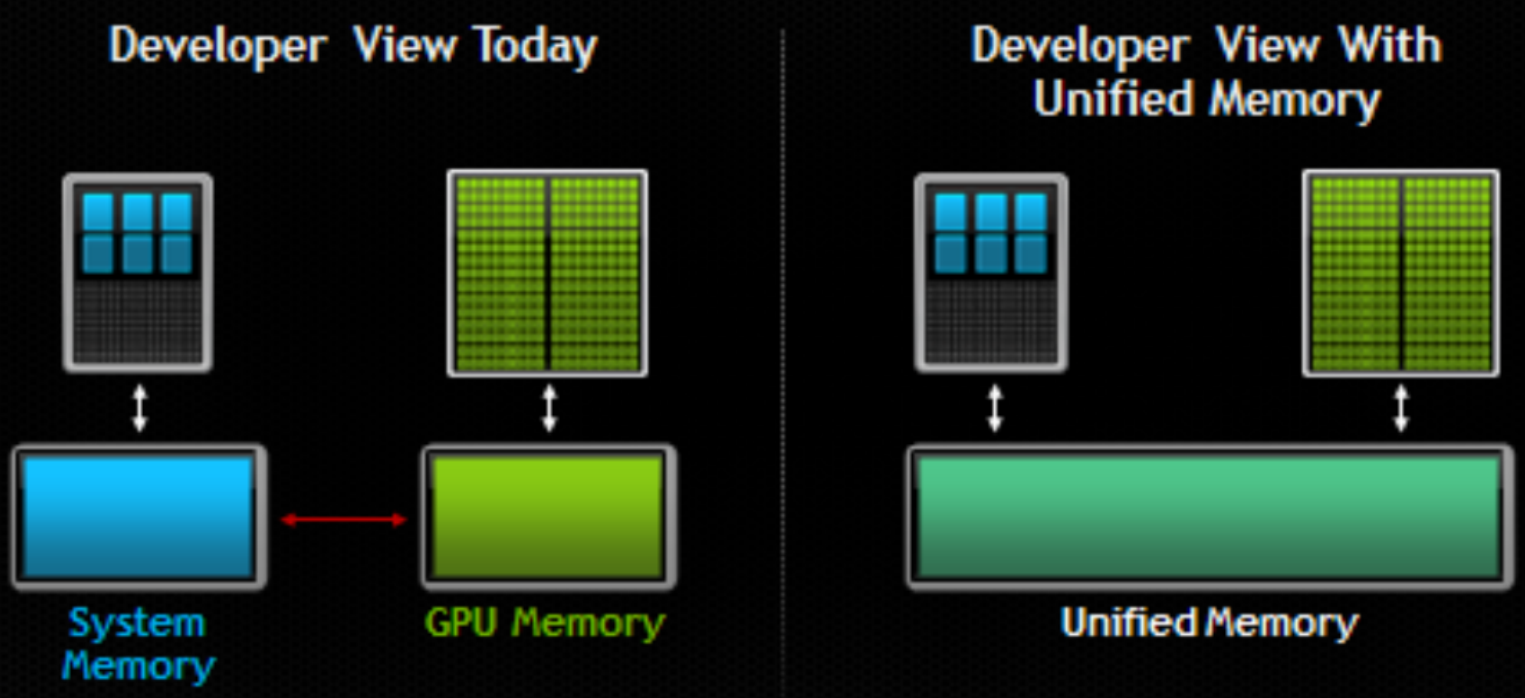
```
const int N=128;

void add(int *a, int *b, int *c) {
    int i = 0;
    while (i < N) {
        c[i] = a[i] + b[i];
        i += 1;
    }
}
```

```
const int N = 128;

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

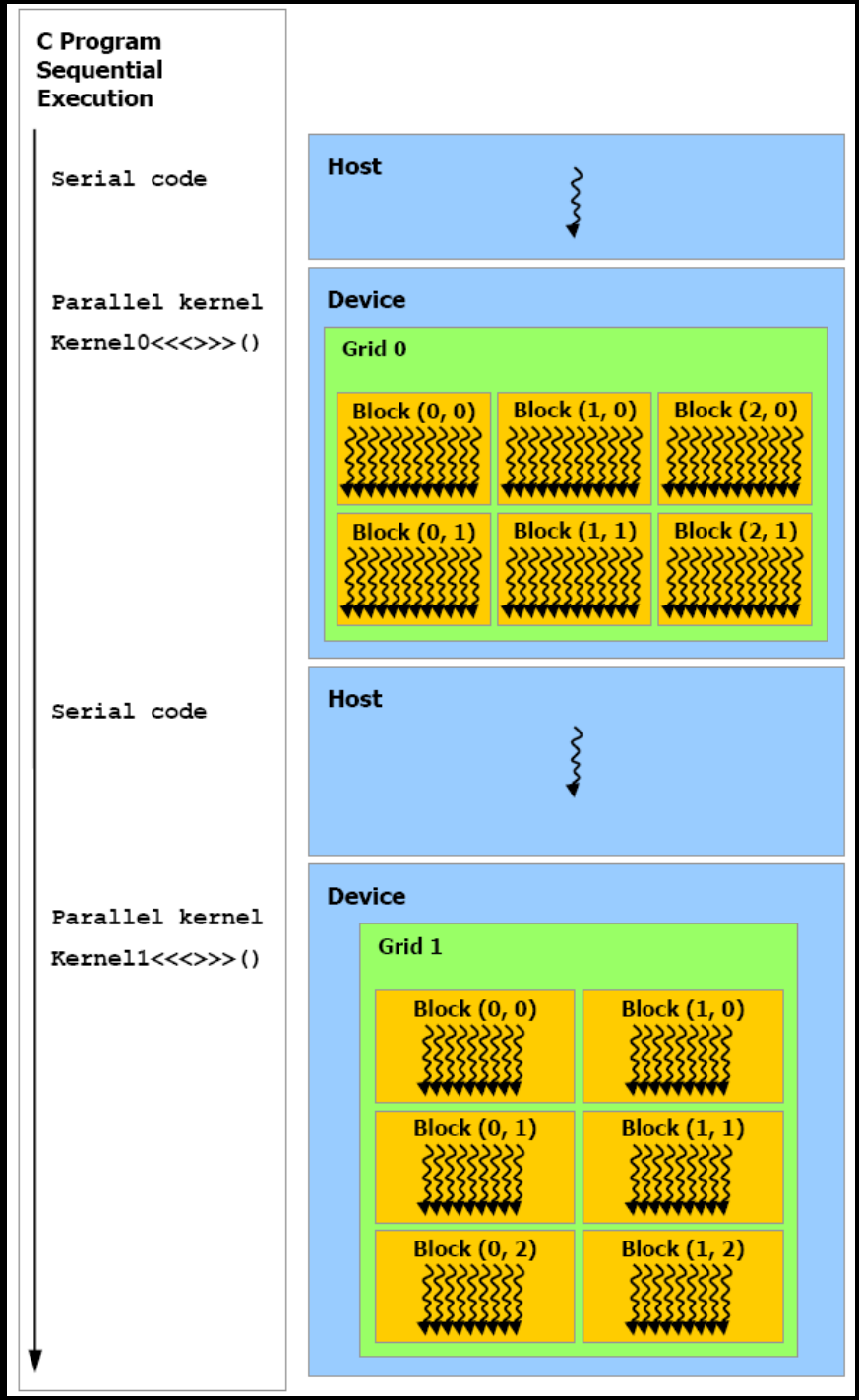
```
int main (void) {
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;

    // allocate the memory on the GPU
    cudaMalloc( (void**)&dev_a, N * sizeof(int) );
    cudaMalloc( (void**)&dev_b, N * sizeof(int) );
    cudaMalloc( (void**)&dev_c, N * sizeof(int) );

    // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = -i;          b[i] = i * i;
    }    // copy the arrays 'a' and 'b' to the GPU

    cudaMemcpy ( dev_a, a, N * sizeof(int), cudaMemcpyHostToDevice );
    cudaMemcpy ( dev_b, b, N * sizeof(int), cudaMemcpyHostToDevice );

    add<<<N,1>>>(dev_a, dev_b, dev_c);

    // copy the array 'c' back from the GPU to the CPU
    cudaMemcpy ( c, dev_c, N * sizeof(int), cudaMemcpyDeviceToHost );

    // display the results
    for (int i=0; i<N; i++) {
        printf("%d + %d = %d₩n", a[i],b[i],c[i]);
    }
    // free the memory allocated on the CPU
   cudaFree (dev_a);    cudaFree (dev_b);    cudaFree (dev_c);
    return 0;
}
```

Unified Memory

Dramatically Lower Developer Effort



```
int main (void) {
    int *a, *b, *c;
    cudaMallocManaged (&a, N);
    cudaMallocManaged (&b, N);
    cudaMallocManaged (&c, N);
   // fill the arrays 'a' and 'b' on the CPU
    for (int i=0; i<N; i++) {
        a[i] = -i;          b[i] = i * i;     }
   add<<<N,1>>>(a, b, c);
   cudaDeviceSynchronize();
   for (int i=0; i<N; i++) {
       printf("%d + %d = %d₩n", a[i],b[i],c[i]);     }
   cudaFree (a); cudaFree (b); cudaFree (c);
   return 0;
}
```


C Program
Sequential
Execution
Serial code
Parallel kernel
Kernel0<<<>>>()
Serial code
Parallel kernel
Kernel1<<<>>>()
Host
Device
Grid 0
Block (0, 0)
Block (1, 0)
Block (2, 0)
Block (0, 1)
Block (1, 1)
Block (2, 1)
Host
Device
Grid 1
Block (0, 0)
Block (1, 0)
Block (0, 1)
Block (1, 1)
Block (0, 2)
Block (1, 2)

AI (Arithmetic Intensity)

- Definition: number of operations per byte
- AI for a[N]+b[N]

  N ops / 8N bytes = 1/8
- AI for A[N][N] * B[N][N]

  $2N^3$ ops/ $8N^2$ bytes = N/4
- If AI ~ 1, performance = AI x memory BW
- If AI >~ 40(2.x),15(1.x), can hide latency of global memory → better performance

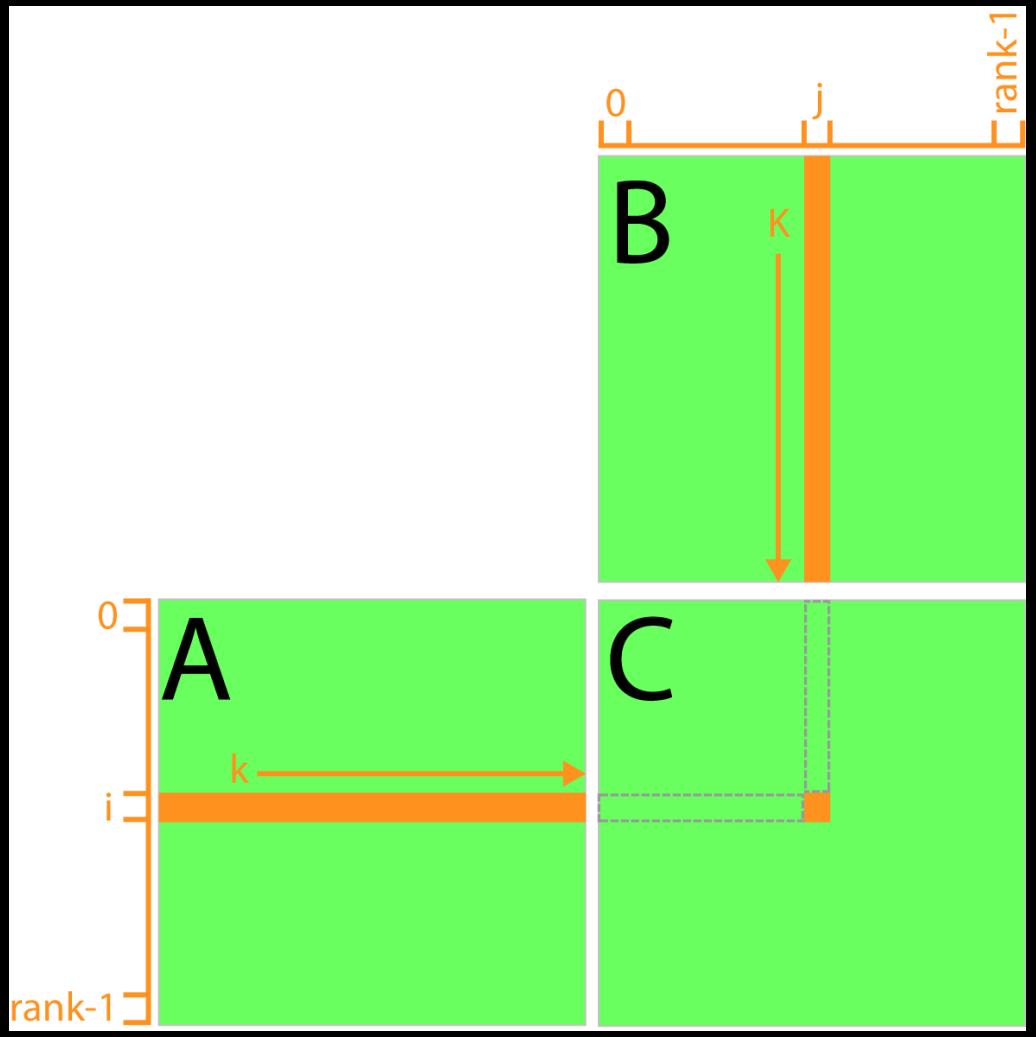
Exercise 5: Matrix Multiplication

- gcc –std=c99 matmul.c
- nvcc –Xptxas –v –arch sm_20 matmul_global.cu
- nvcc –Xptxas –v –arch sm_20 matmul_shared.cu

Exercise 5: Matrix Multiplication

```
void MatMul(const float * A, const float * B, float * C) {

    for (int row=0; row<N; ++row)
    for (int col=0; col<N; ++col) {
      float Cvalue = 0;
        for (int k = 0; k < N; ++k) {
          Cvalue += A[row*N+k]*B[k*N+col];
        }
        C[row*N+col] = Cvalue;
     }
}
```

M-M Multiplication: Global



Compute Capability

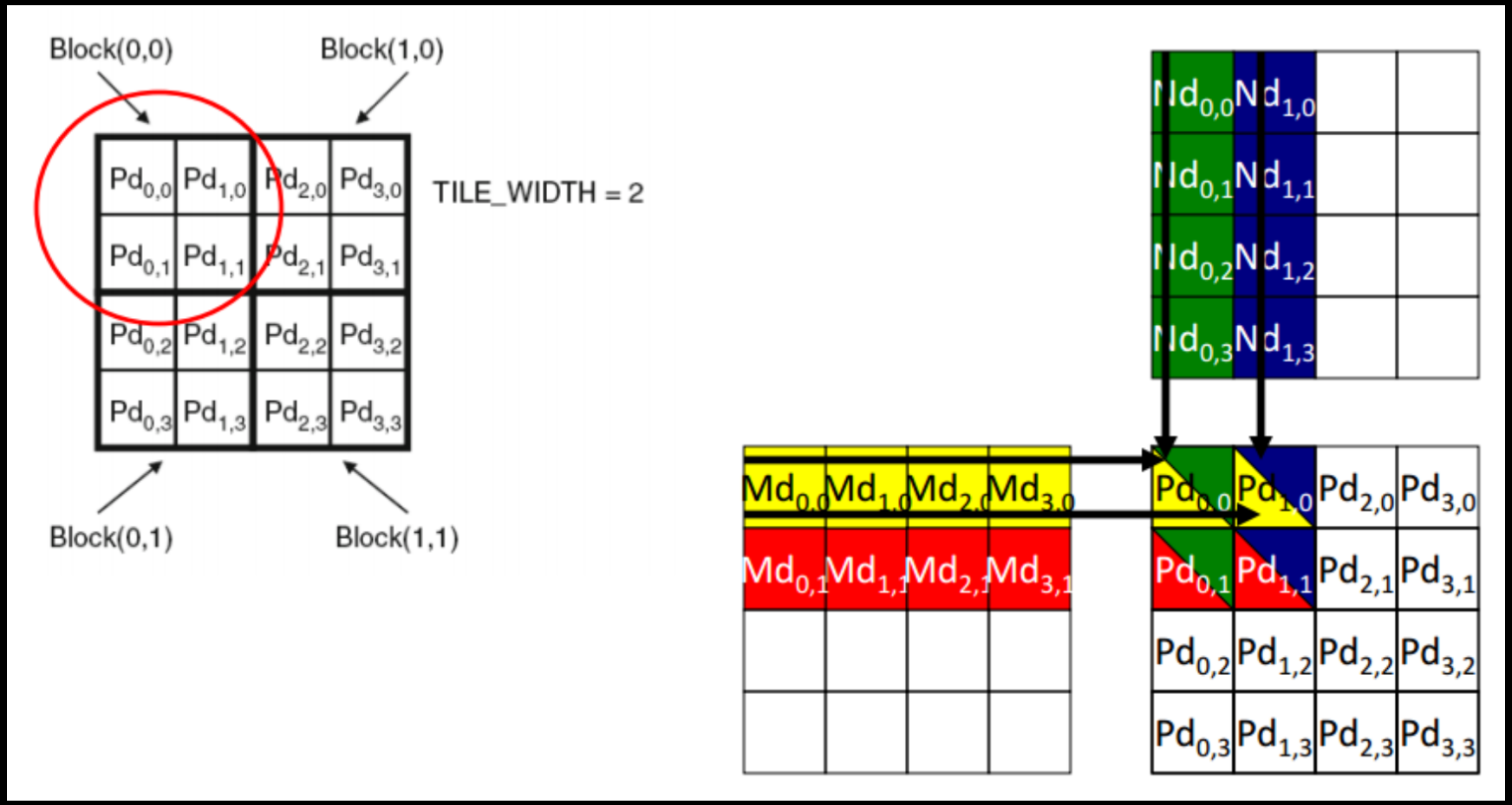
Table 14. Technical Specifications per Compute Capability

	Compute Capability										
Technical Specifications	**3.0**	**3.2**	**3.5**	**3.7**	**5.0**	**5.2**	**5.3**	**6.0**	**6.1**	**6.2**	**7.0**
Maximum number of resident grids per device (Concurrent Kernel Execution)	16	4	32				16	128	32	16	128
Maximum dimensionality of grid of thread blocks	3										
Maximum x-dimension of a grid of thread blocks	$2^{31}-1$										
Maximum y- or z-dimension of a grid of thread blocks	65535										
Maximum dimensionality of thread block	3										
Maximum x- or y-dimension of a block	1024										
Maximum z-dimension of a block	64										
Maximum number of threads per block	1024										
Warp size	32										
Maximum number of resident blocks per multiprocessor	16				32						
Maximum number of resident warps per multiprocessor	64										
Maximum number of resident threads per multiprocessor	2048										
Number of 32-bit registers per multiprocessor	64 K			128 K	64 K						

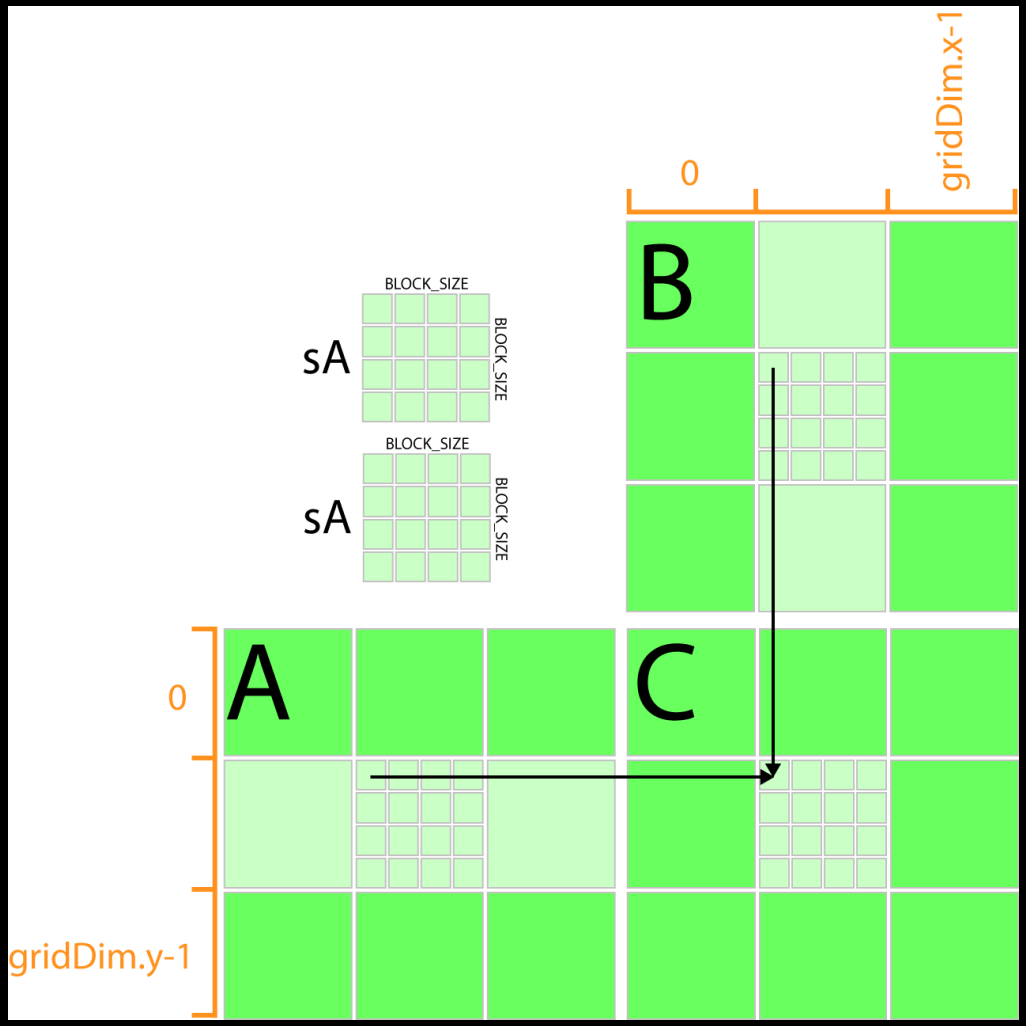
Reducing Global Memory Traffic

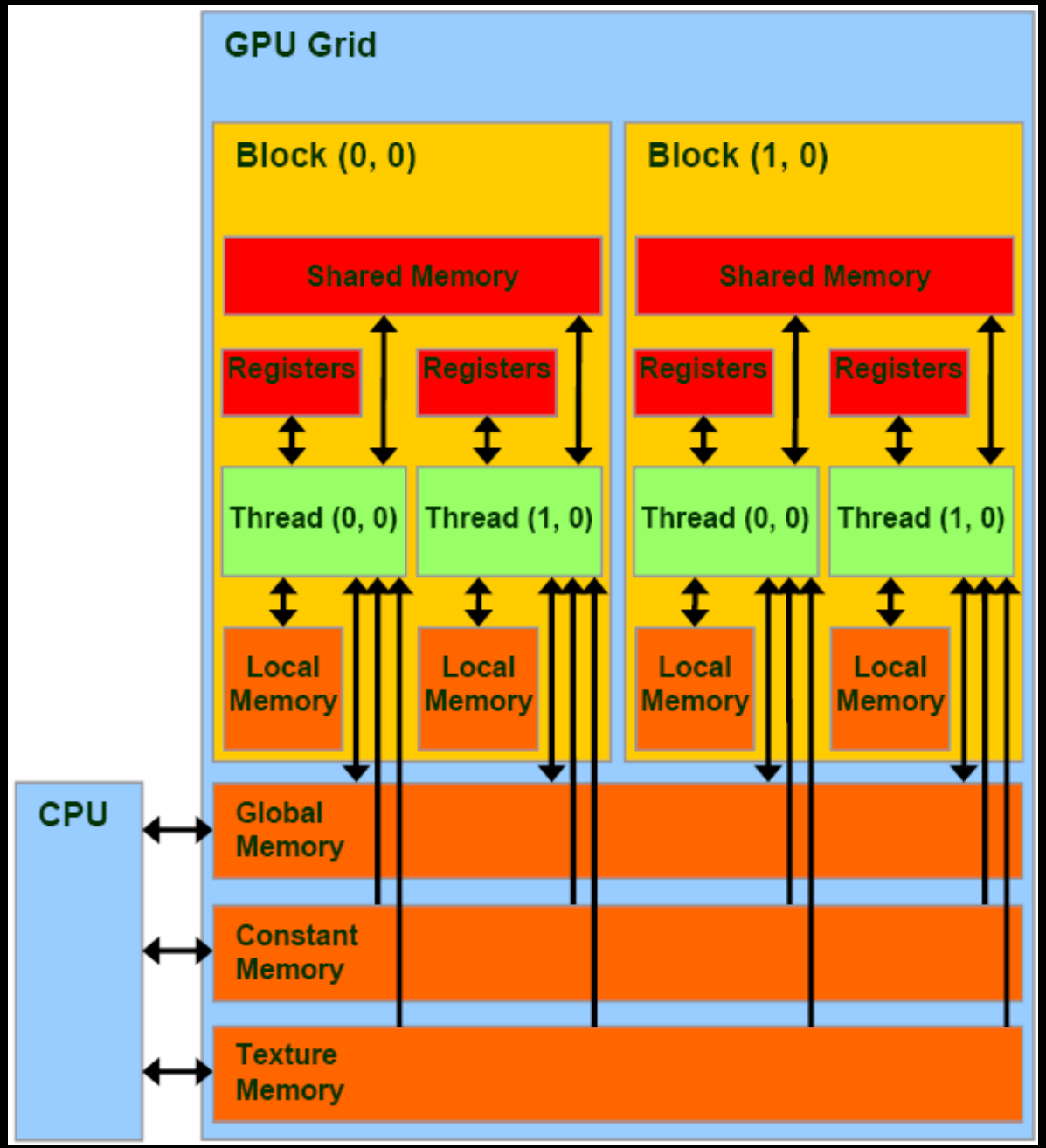
- Reducing global memory access enhances performance
- tiling: partition of data into subsets called tiles, such that each tile fits into fast (shared) memory

Matrix Mul: tiling



Matrix-Multiplication: shared

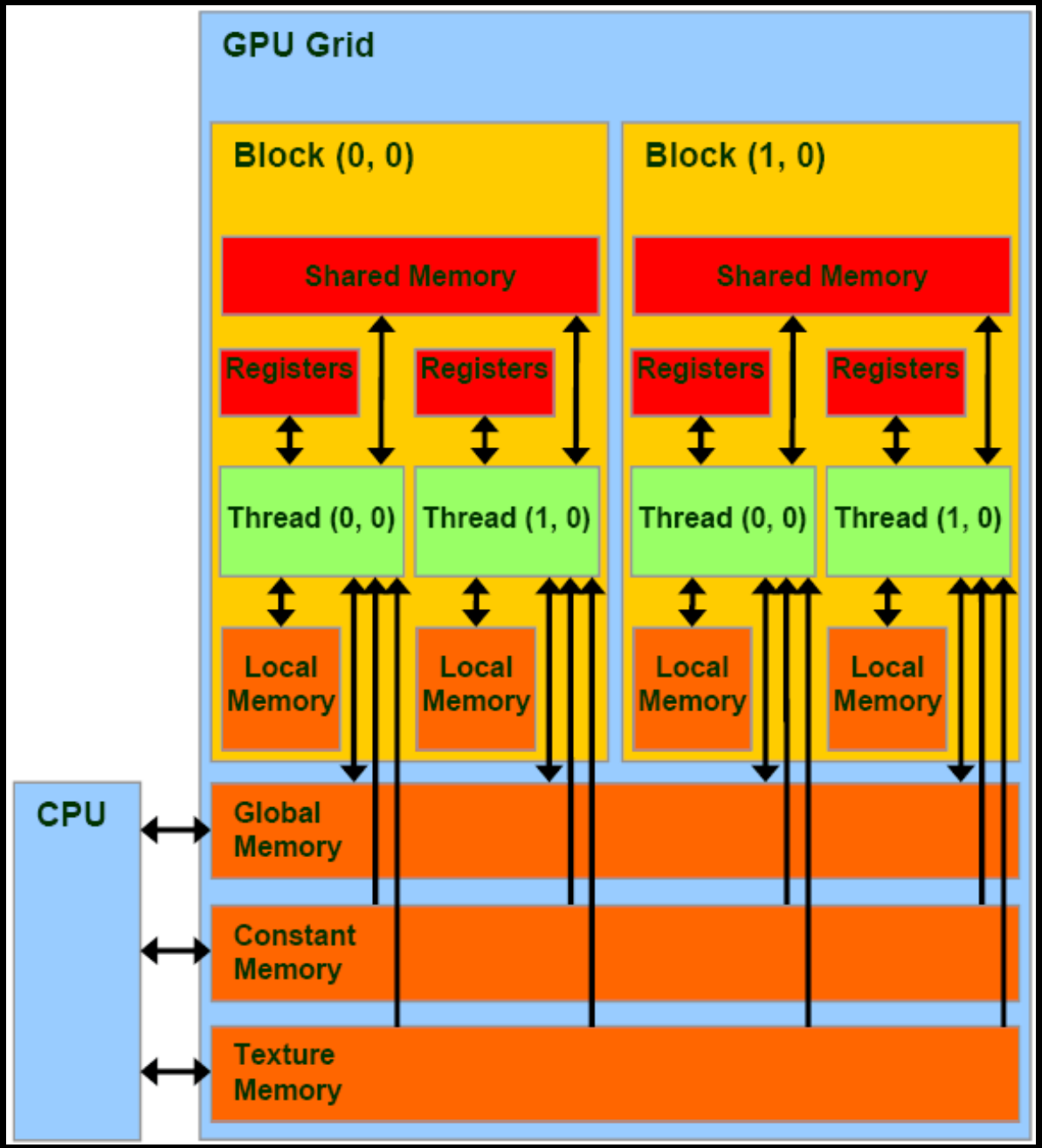




Register
- Fastest
- Thread scope, thread lifetime

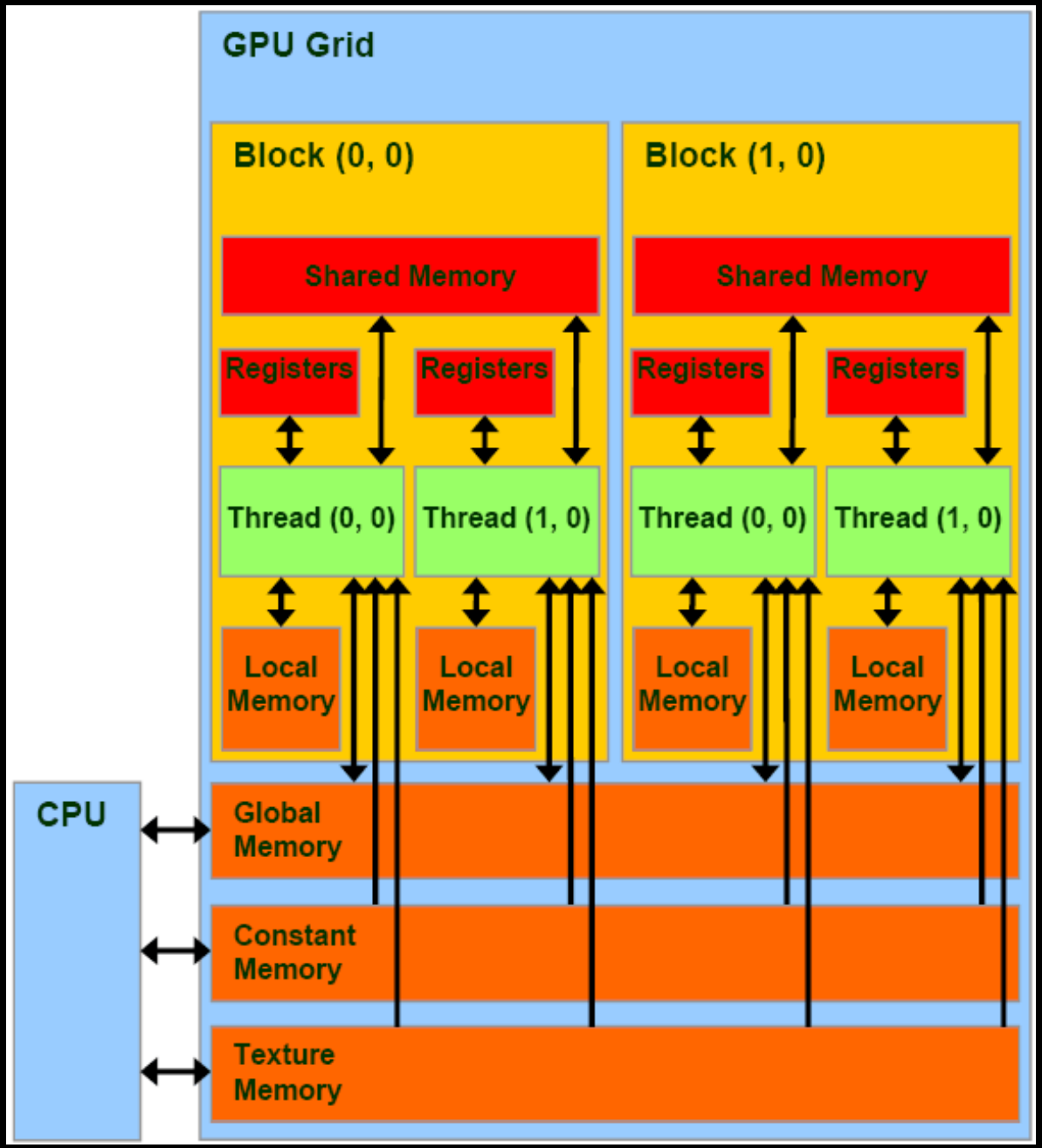
Local
- Does not exist
- Abstraction of thread scope

Global memory
- Implemented in DRAM
- High-access latency: 400-800 cycles
- Finite bandwidth: 148GB/sec for S2050
- Potential of traffic congestion
- Grid scope, application lifetime



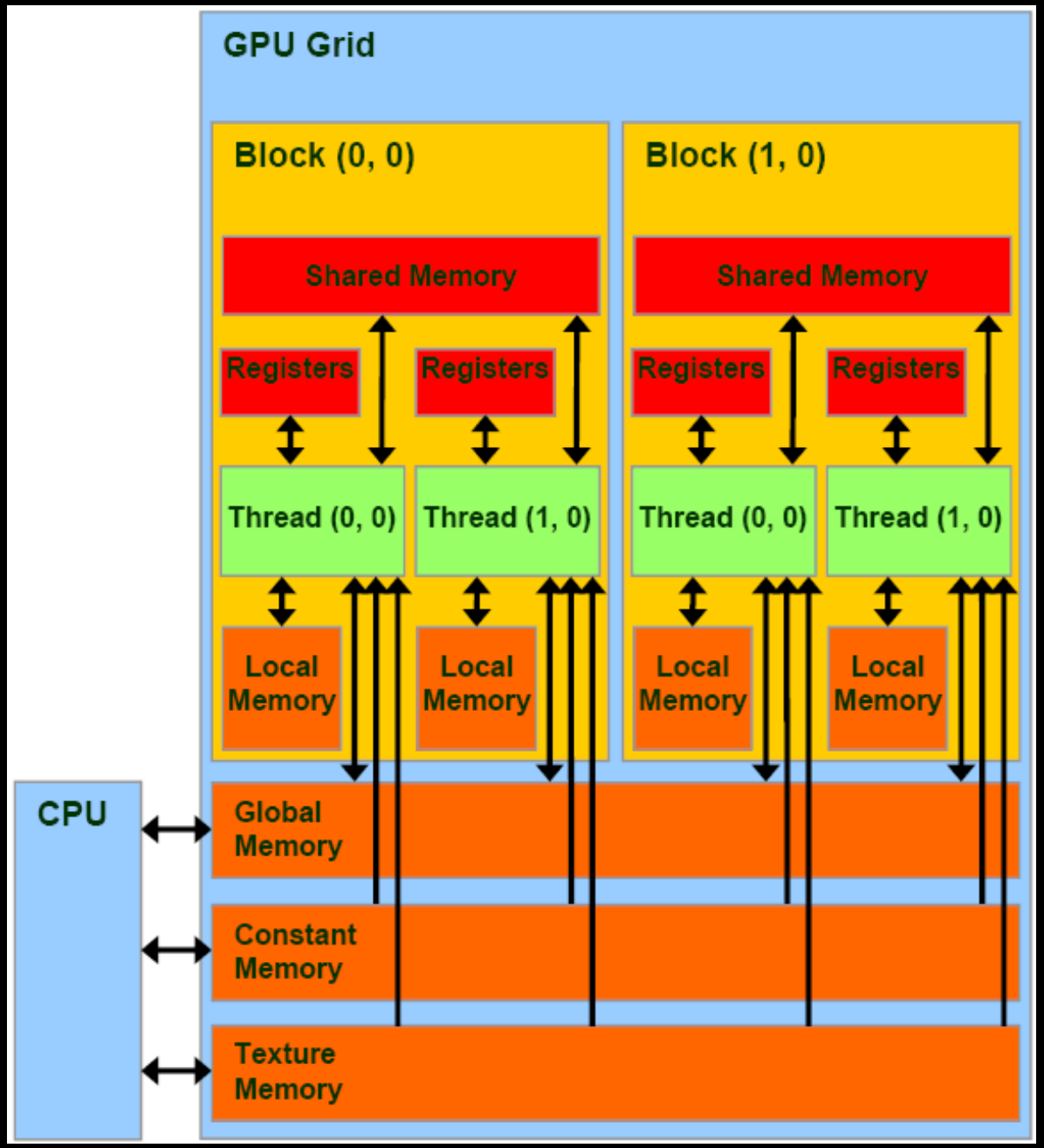
Shared memory

- Extreme fast, highly parallel
- Block scope, kernel lifetime



Constant memory

- Space for "constant" data during a kernel execution
- 64kB per a GPU board
- Cached on chip
- Fast if threads of a half warp reads the same address



Texture memory

- cached on chip
- read only
- designed for graphics appllications
- Optimized 2D “spatial locality”

Exercise 4: Dot-Product

```
gcc std=c99 dot_product.c
nvcc dot_product.cu
nvcc –arch sm_13 dot_product_double.cu
nvcc -l cublas dot_product_cublas.cu
```

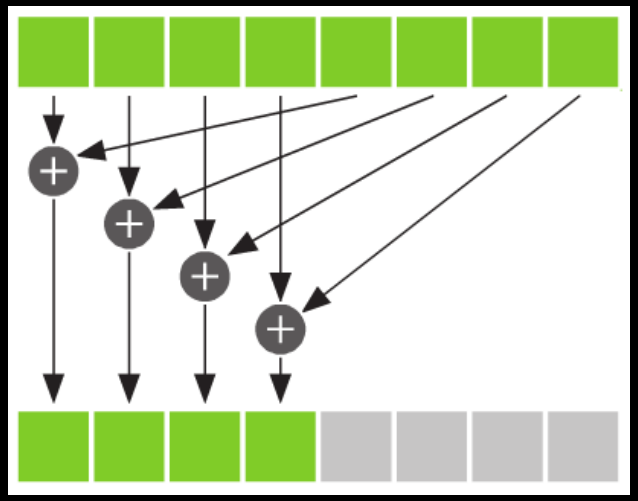
Exercise 4: Dot Product

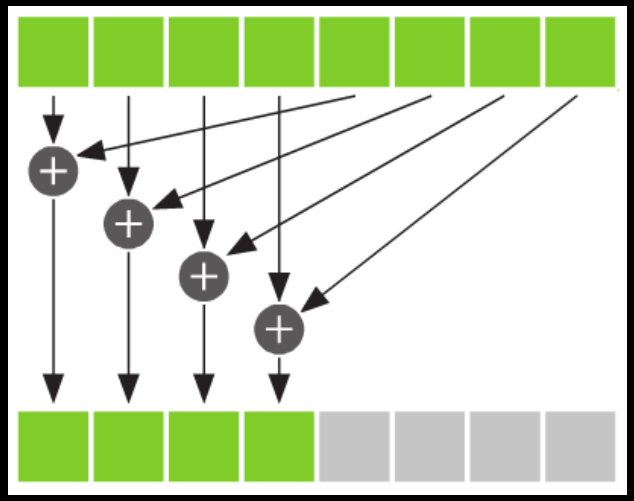
```
#include <stdio.h>
#define sum_squares(x) (x*(x+1)*(2*x+1)/6)

const int N = 2048;

float dot_product(float *a, float *b) {

    float c = 0.0f;
    for (int i=0; i<N; i++)
        c += a[i]*b[i];
    return c;
}
```

```
int main (void) {

    float a[N], b[N];

    for (int i=0; i<N; i++) {
        a[i] = (float) i;
        b[i] = (float) i;
    }

    float c = dot_product(a,b);

    printf("Dot prodoct of a and b = %f₩n", c);
    printf("sum_squares of (N-1) = %f₩n",
                sum_squares((float)(N-1)) );

    return 0;
}
```

Reduction

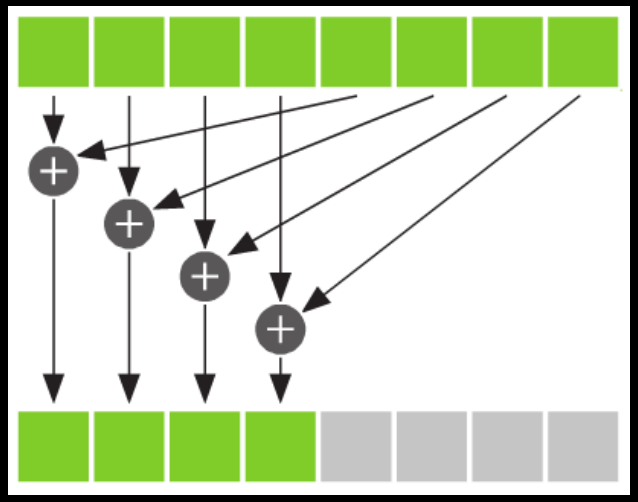
blockIdx 0

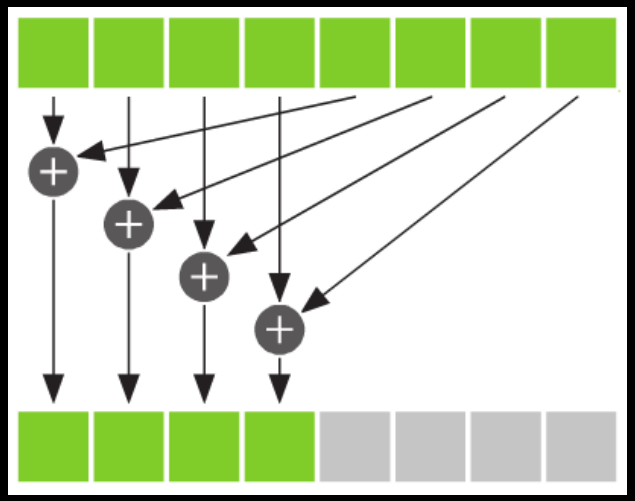




blockIdx 1

blockIdx 2





blockIdx 3

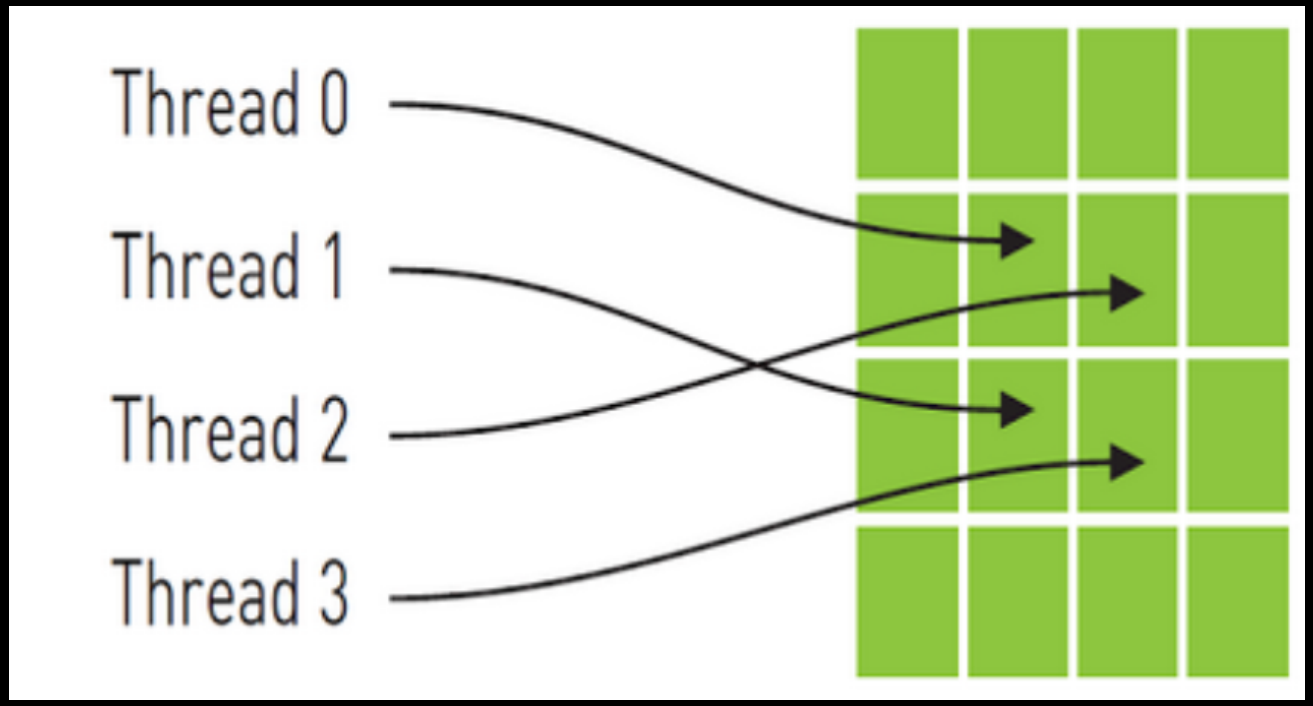
cuBLAS

- Basic Linear Algebra Subprograms on CUDA
  - cublasHandle_t handle=0;
  - cublasCreate(&handle)
  - cublasSdot(handle,N,dev_a,1,dev_b,1,&c);
  - cublasDestroy(handle);

Exercise 6: constant memory

- nvcc –arch sm_70 const_mem.cu

Spatial Locality



Heat equation

$$\frac{\partial T}{\partial t} = \kappa\left(\frac{\partial}{\partial x^2} + \frac{\partial}{\partial y^2}\right)T$$

$$\frac{T_{i,j}^{n+1} - T_{i,j}^{n}}{\Delta t} = \kappa\left(\frac{T_{i+1,j}^{n} - 2T_{i,j}^{n} + T_{i-1,j}^{n}}{\Delta x^2} + \frac{T_{i,j+1}^{n} - 2T_{i,j}^{n} + T_{i,j-1}^{n}}{\Delta y^2}\right)$$

$$T_{i,j}^{n+1} = \left(1 - 4\frac{\kappa\Delta t}{h^2}\right)T_{i,j}^{n} + \frac{\kappa\Delta t}{h^2}\left(T_{i+1,j}^{n} + T_{i-1,j}^{n} + T_{i,j+1}^{n} + T_{i,j-1}^{n}\right)$$

where $h = \Delta x = \Delta y$

$$T_{i,j}^{n+1} = \frac{1}{4}\left(T_{i+1,j}^{n} + T_{i-1,j}^{n} + T_{i,j+1}^{n} + T_{i,j-1}^{n}\right) \text{ when } \Delta t = \frac{h^2}{4\kappa}$$

Exercise 7: heat equation

- gcc –std=c99 heat.c
- nvcc –arch sm_20 heat_global.cu
- nvcc –arch sm_20 heat_texture_1d.cu
- nvcc –arch sm_20 heat_texture_2d.cu

Continuous vs discrete Fourier Transform

$$H(f) = \int_{-\infty}^{\infty} h(t) e^{-j2\pi f t} dt$$

$$H = \sum_{k=0}^{N-1} h(k) e^{-j2\pi nk/N} \quad n = 0,1, \ldots N-1$$

Fourier Transform of a rectangular pulse

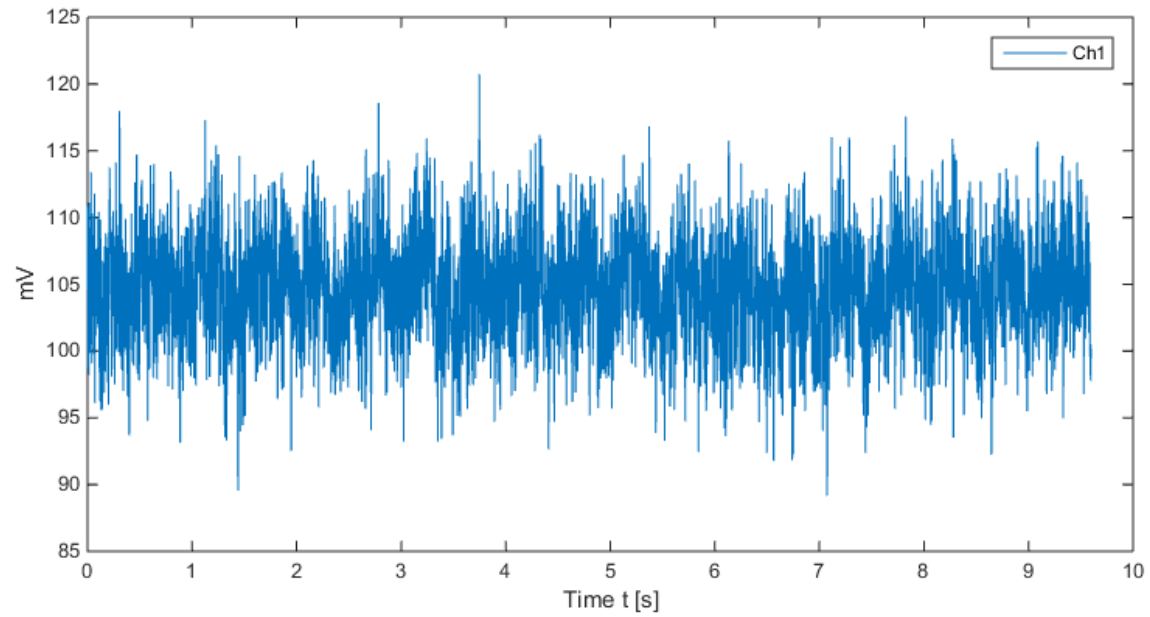
$$h(t) = A \quad |t| < T_0$$
$$= 0 \quad |t| > T_0$$

$$H(f) = \int_{-T_0}^{T_0} Ae^{-j2\pi ft} dt$$

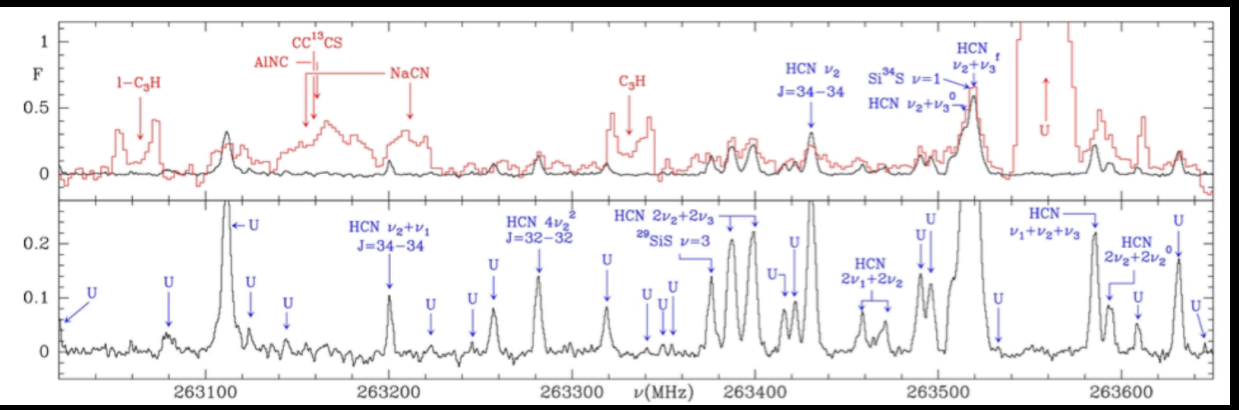
$$= A\int_{-T_0}^{T_0} \cos(2\pi ft)\, dt - jA\int_{-T_0}^{T_0} \sin(2\pi ft)\, dt = \frac{A}{2\pi f}\sin(2\pi ft)\,|_{-T_0}^{T_0}$$

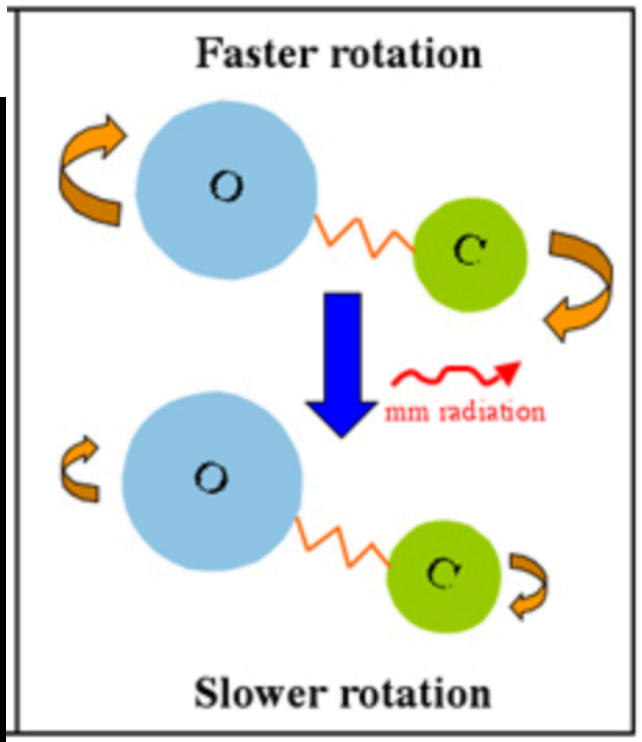
$$= 2AT_0\frac{\sin(2\pi T_0 f)}{2\pi T_0 f}$$

Radio astronom
v 101



Fourier Transform

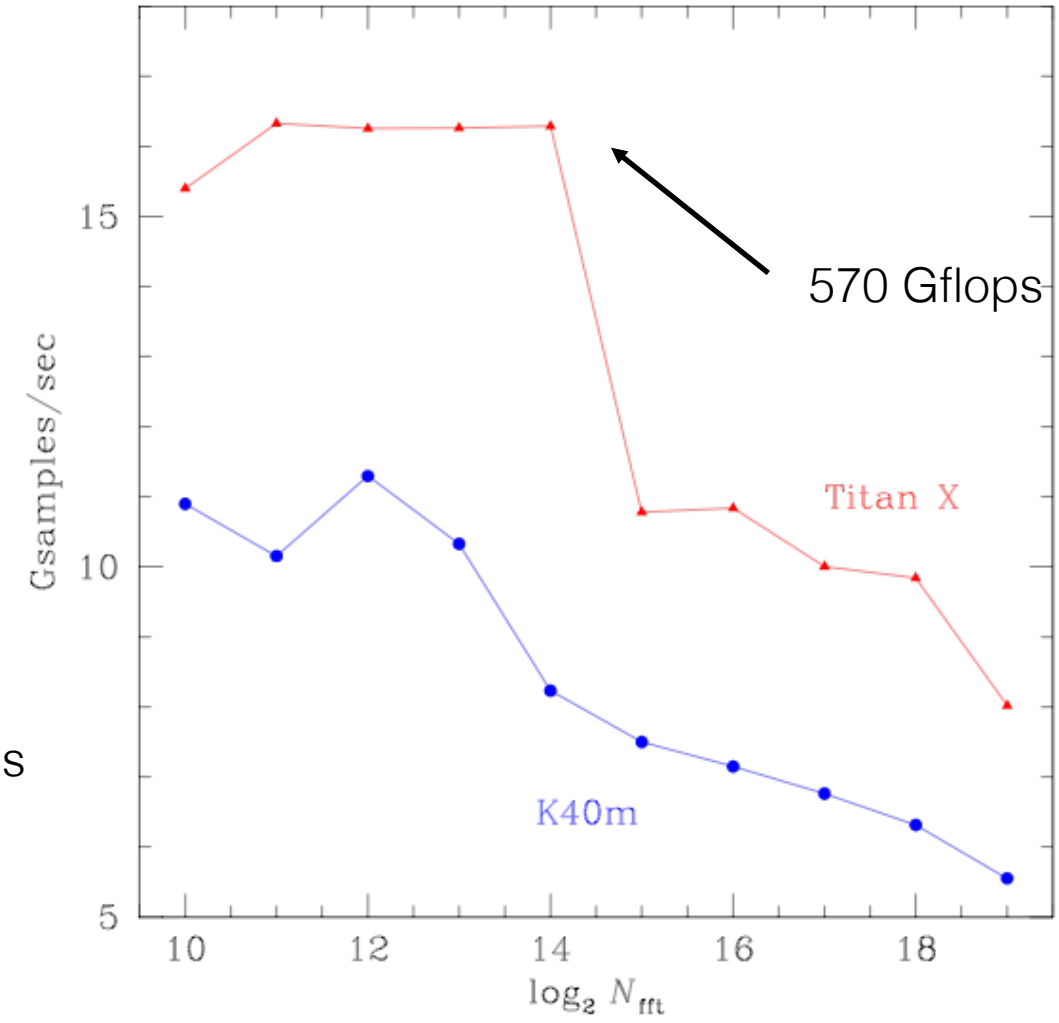




cuFFT

```
cufftPlanMany(….);
cufftExecR2C(plan, idata, odata)
```

- total number of samples: 250*2^19=131,072,000
- 7~16 Gsample/sec (3.5~8 GHz)
- peak performance: ~ 570Gflops (flops: 2.5 N $\log_2$(N) for R2C FFT)
- for a given number of samples, FFT performance is higher at small fft points

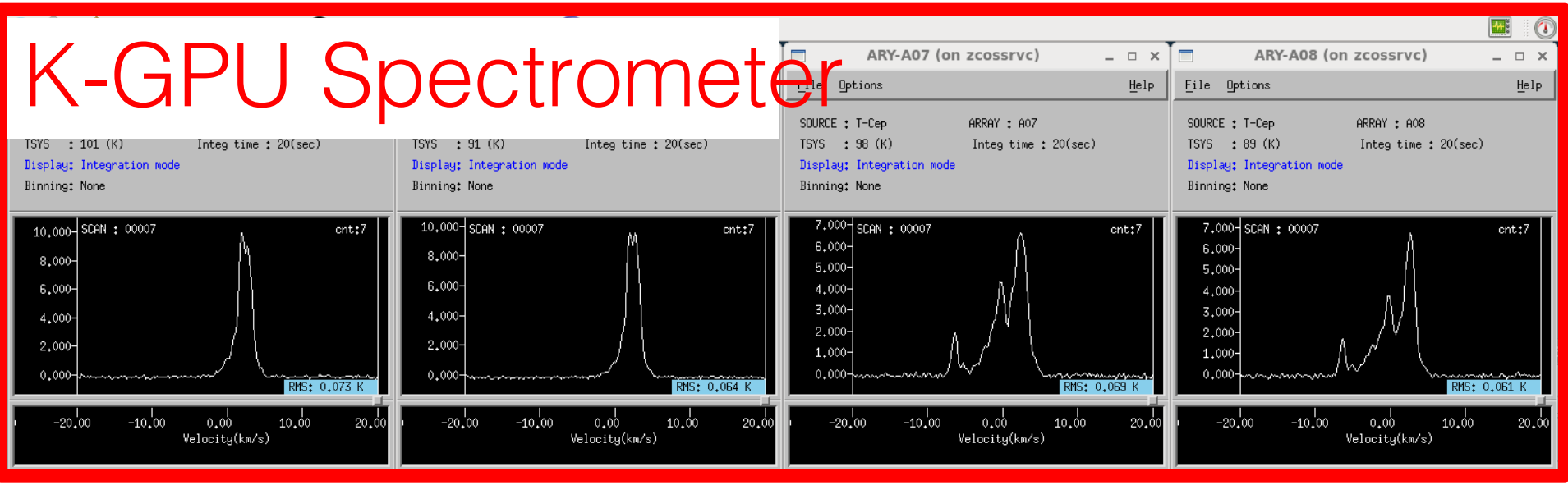


First Spectra with 45-m Telescope (Dec. 25, 2017)

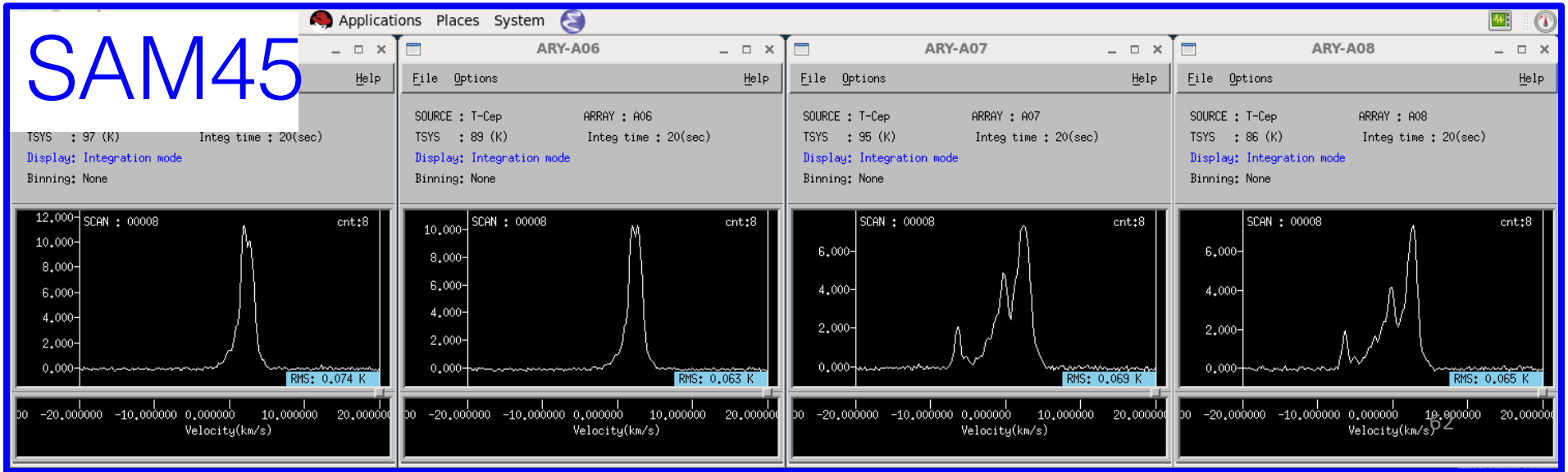
SiO (v=2, J=1-0) @ 42.8GHz

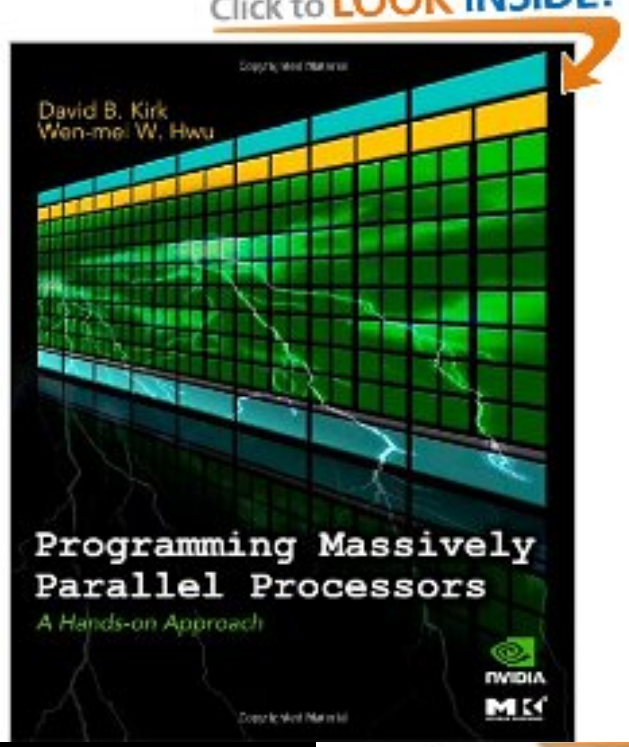
SiO (v=1, J=1-0) @ 43.1GH

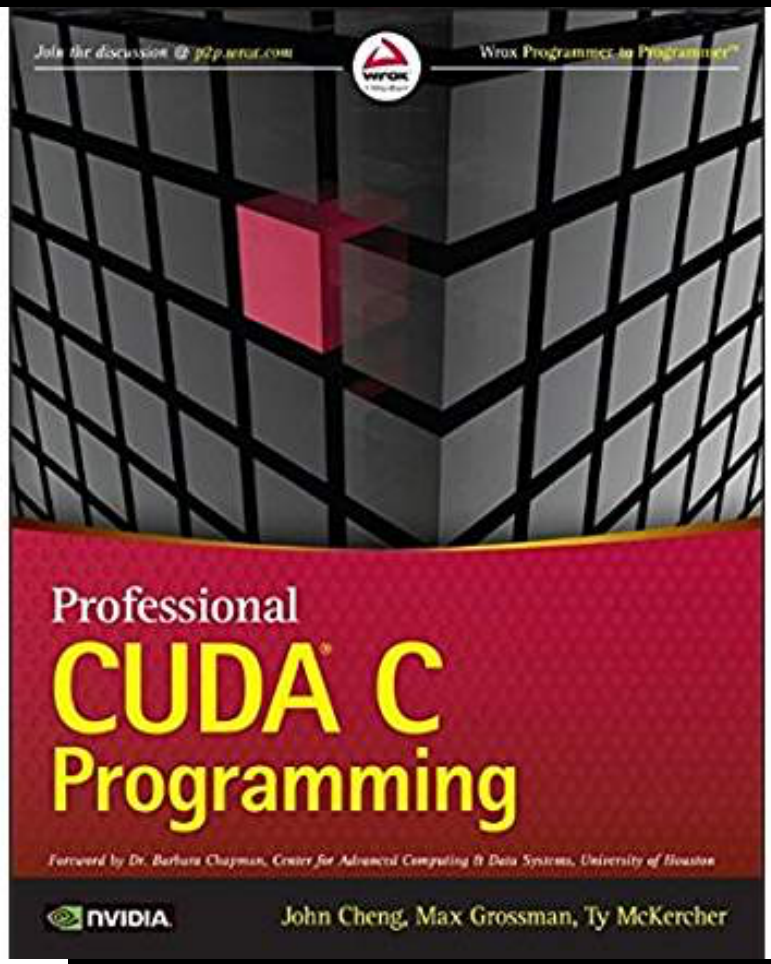
K-GPU Spectrometer

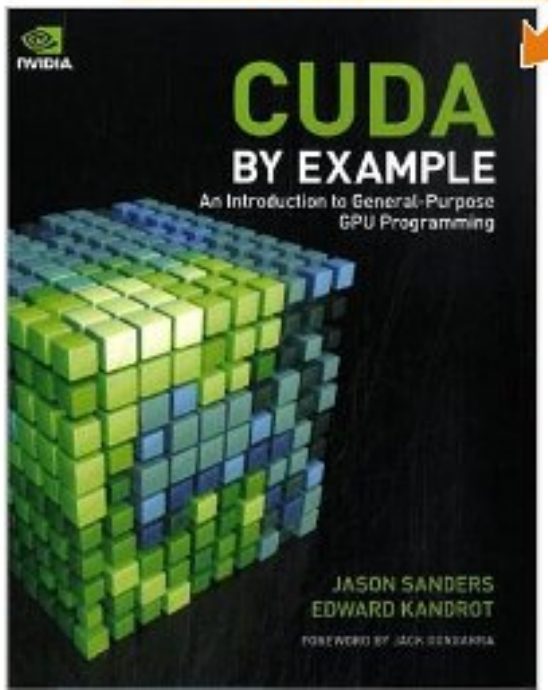


SAM45




Click to LOOK INSIDE!
David B. Kirk
Wen-mei W. Hwu
Programming Massively Parallel Processors
A Hands-on Approach
NVIDIA


Join the discussion @ p2p.wrox.com
Wrox Programmer to Programmer
Professional
CUDA C
Programming
Foreword by Dr. Barbara Chapman, Center for Advanced Computing & Data Systems, University of Houston
NVIDIA
John Cheng, Max Grossman, Ty McKercher


OOK INSIDE
NVIDIA
CUDA
BY EXAMPLE
An Introduction to General-Purpose GPU Programming
JASON SANDERS
EDWARD KANDROT

Useful information

- http://developer.nvidia.com
- CUDA toolkit 9.2
- Getting Started Guide (installation)
- CUDA C programming Guide