Introduction to CUDA Programming (Compute Unified Device Architecture)

Jongsoo Kim (KASI)

7th KIAS CAC Summer School

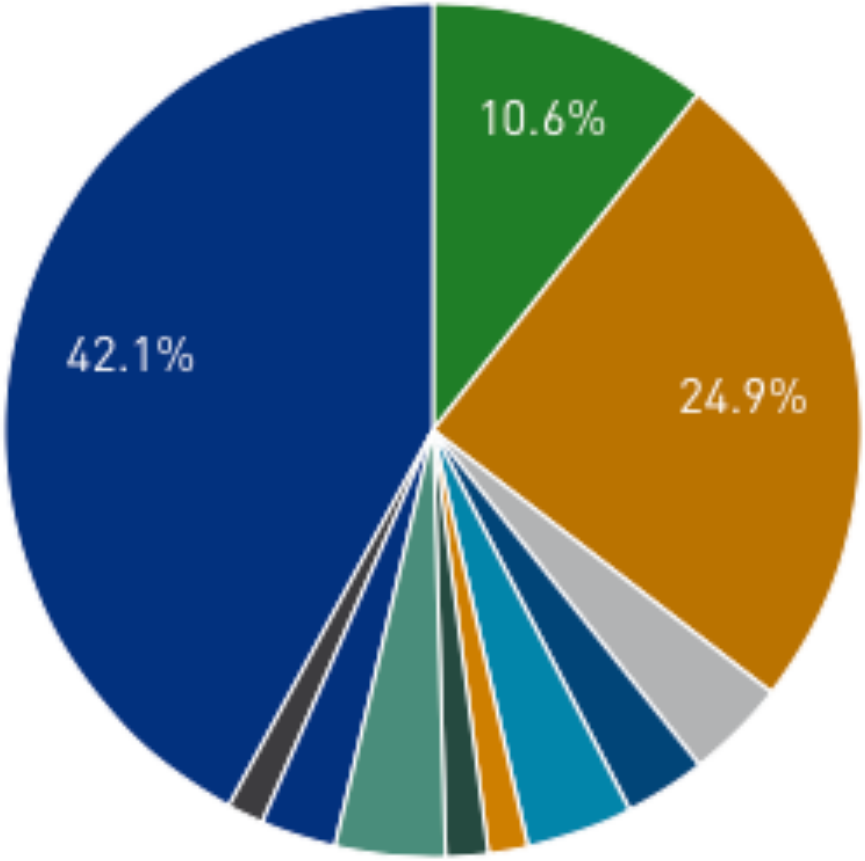
www.top500.org

Titan #2,
Linpack: 17.6 Pflos/s
560,640 cores, 8.2MW
NVIDIA K20x

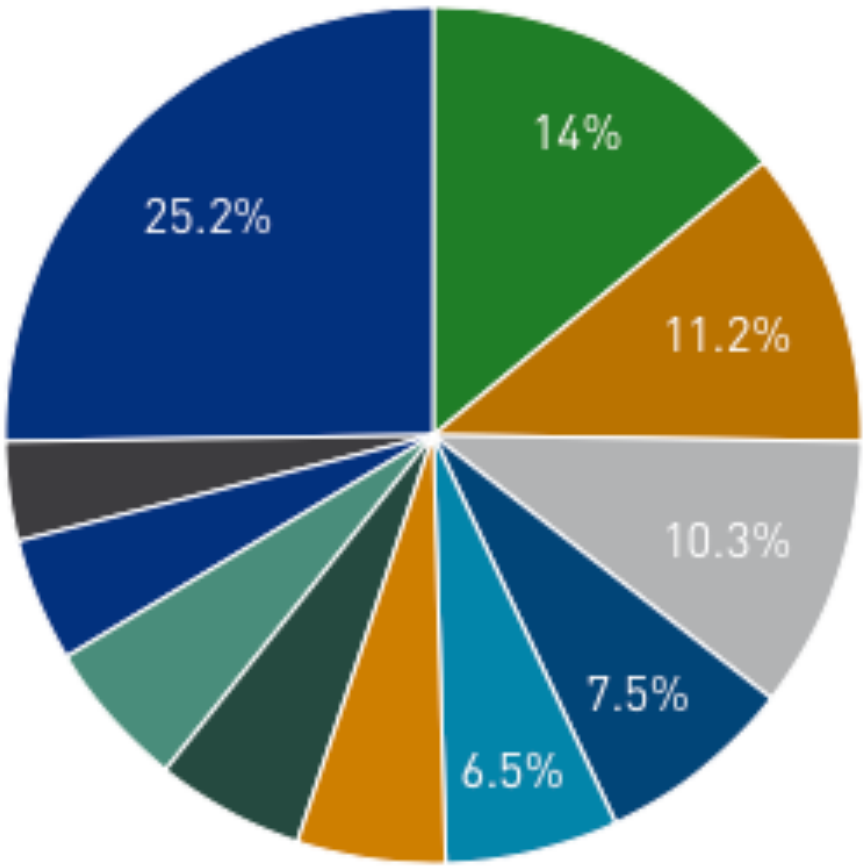
www.top500.org, November 2015

accelerator/co-processor

system

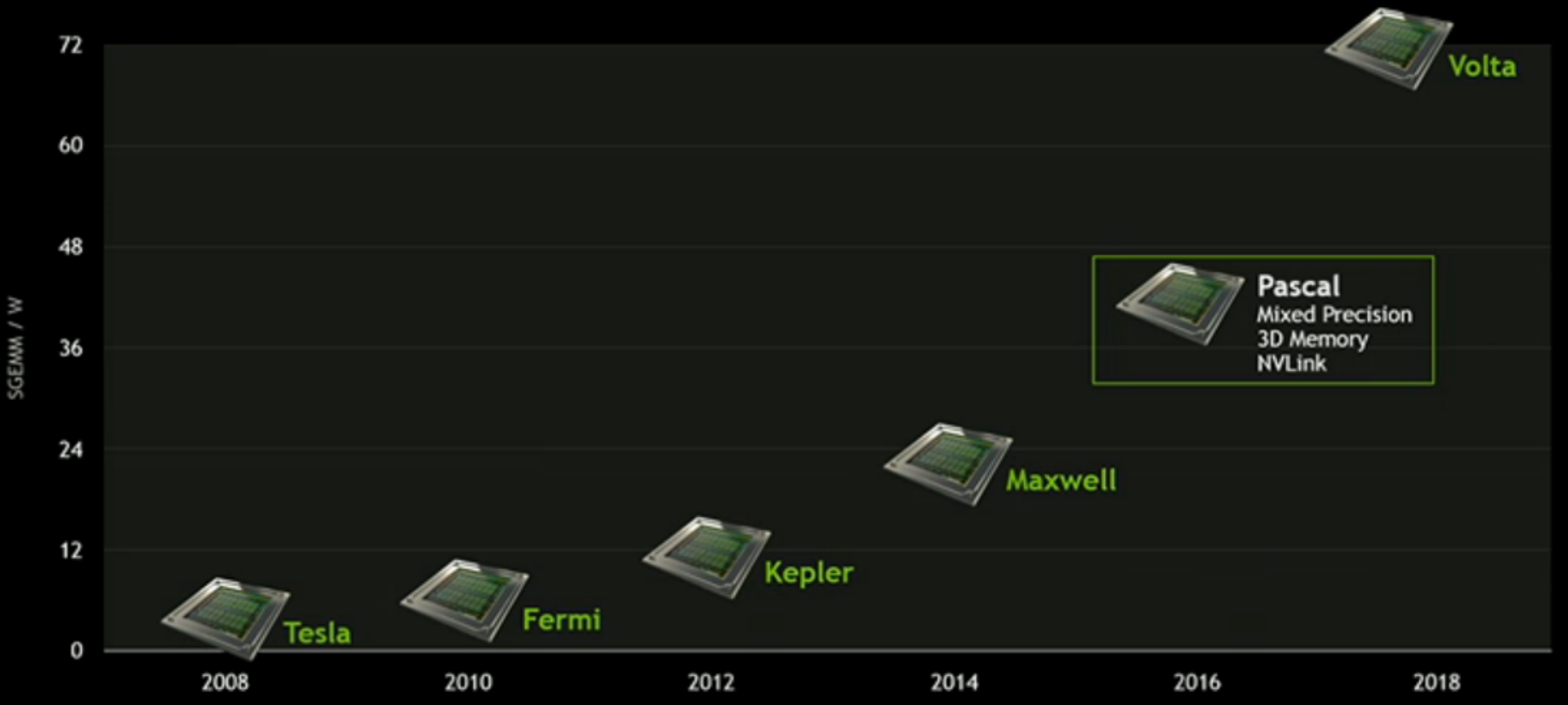


performance

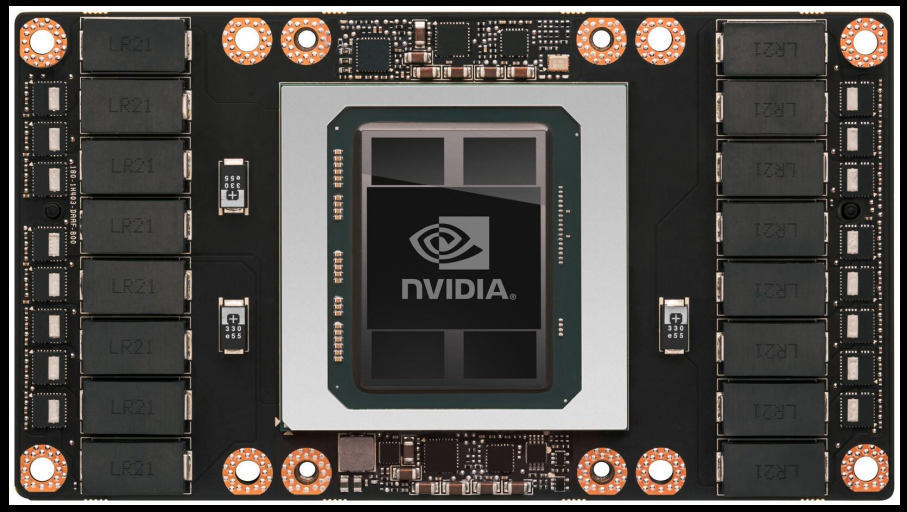


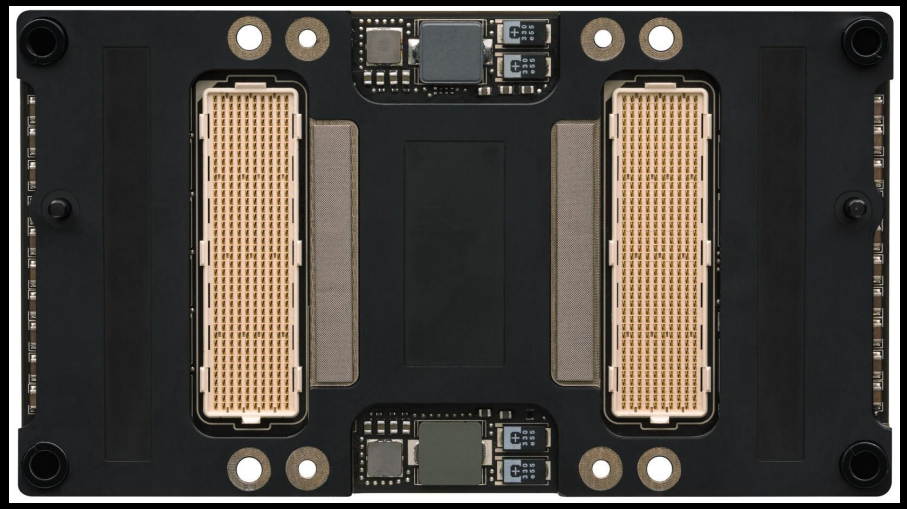
- NVIDIA Tesla K40
- NVIDIA Tesla K20x
- NVIDIA Tesla K80
- Intel Xeon Phi 7120P
- Intel Xeon Phi 5110P
- NVIDIA 2090
- NVIDIA Tesla K20
- NVIDIA 2050
- Intel Xeon Phi 5120D
- NVIDIA Tesla K20m
- Others

NVIDIA GPU Roadmap



NVIDA TESLA P100

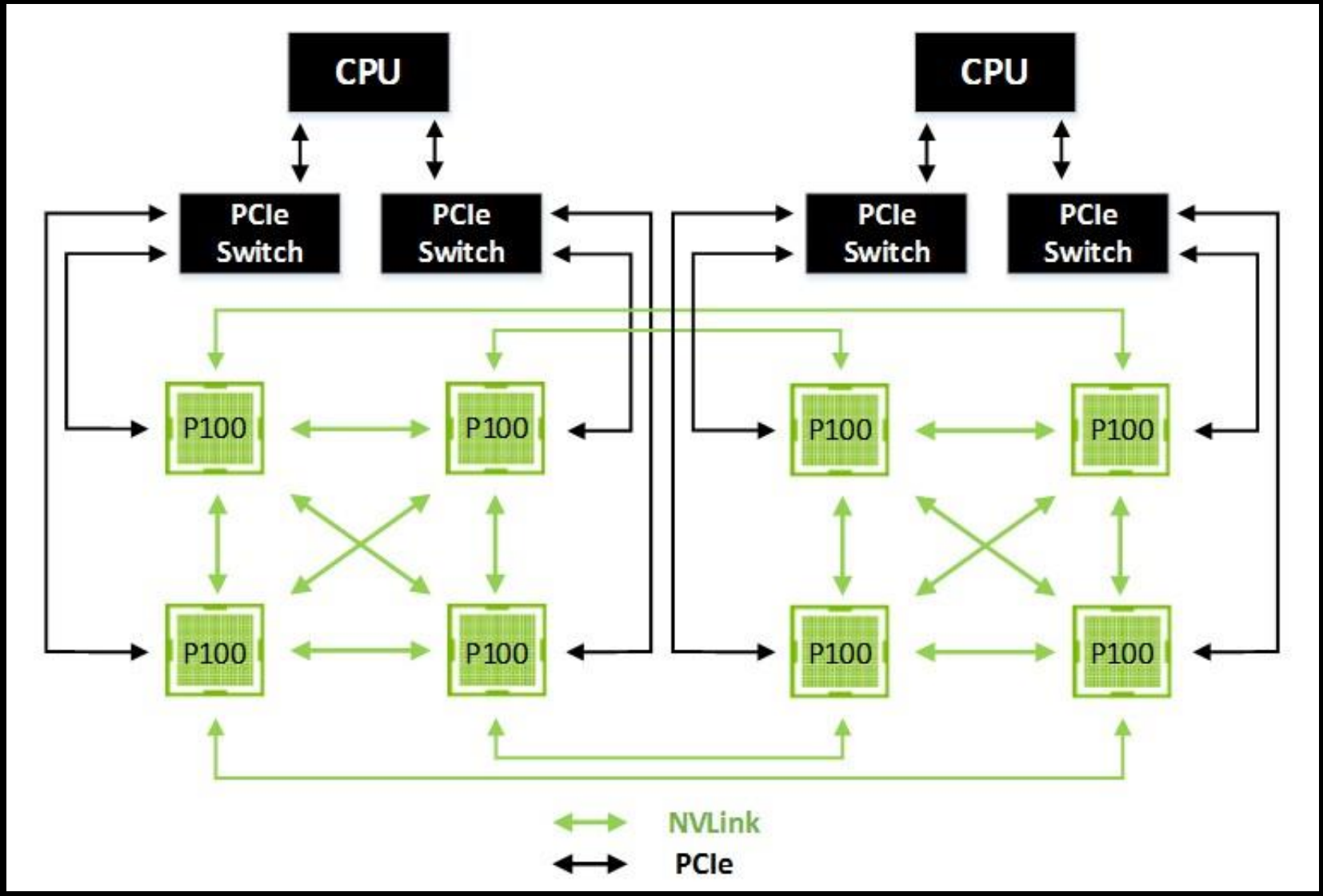




- NVLink
- HBM2
- Unified Memory
- …

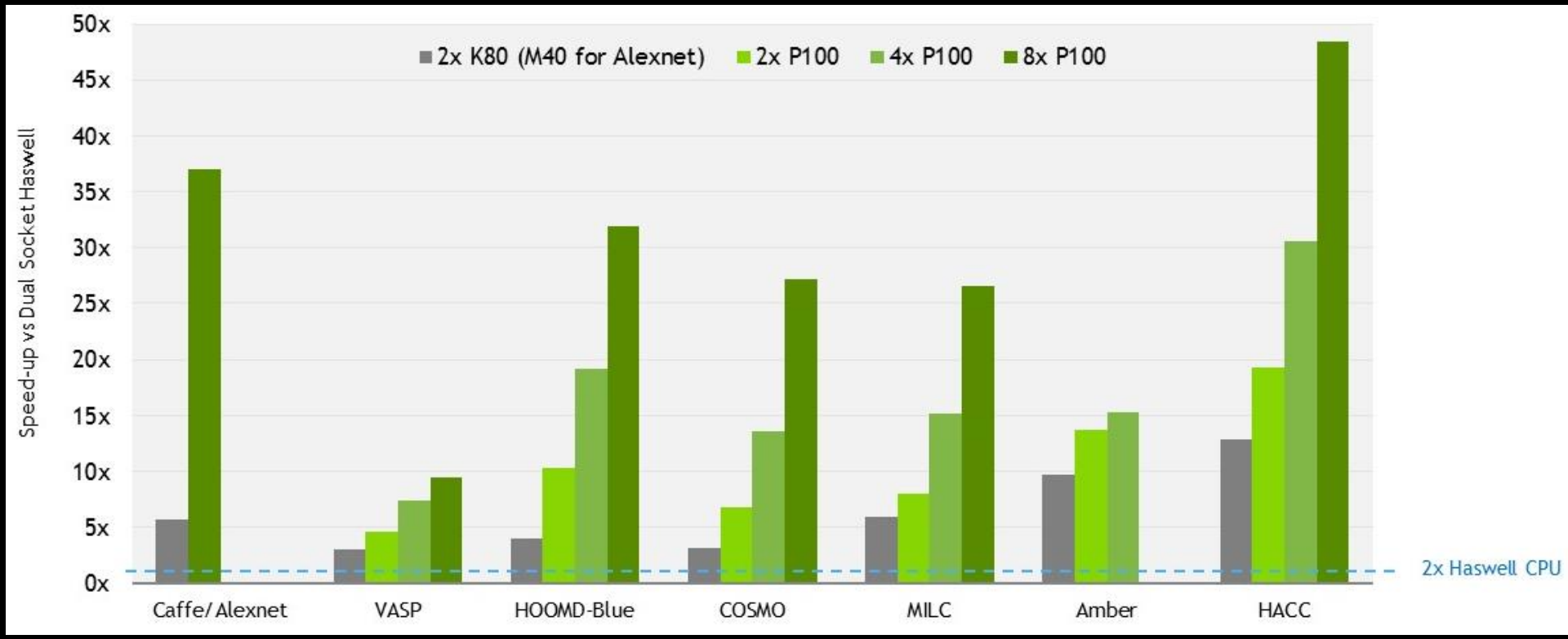
- 5.3 TFLOPS of FP64
- 10.6 TFLOPS of FP32
- 21.2 TFLOPS of FP16

NVLink: 160GB/sec (5x PCIe 3x16)

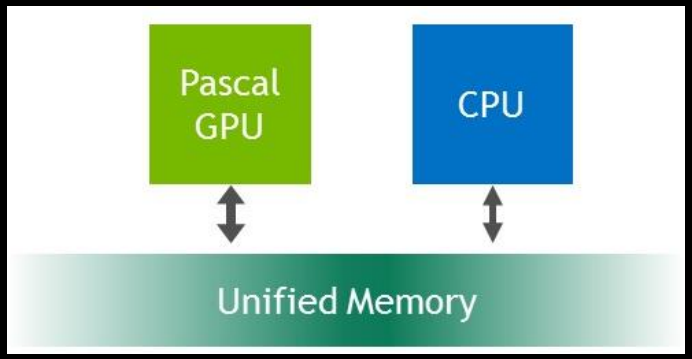


Performance

NVIDIA DGX1



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);
}
```

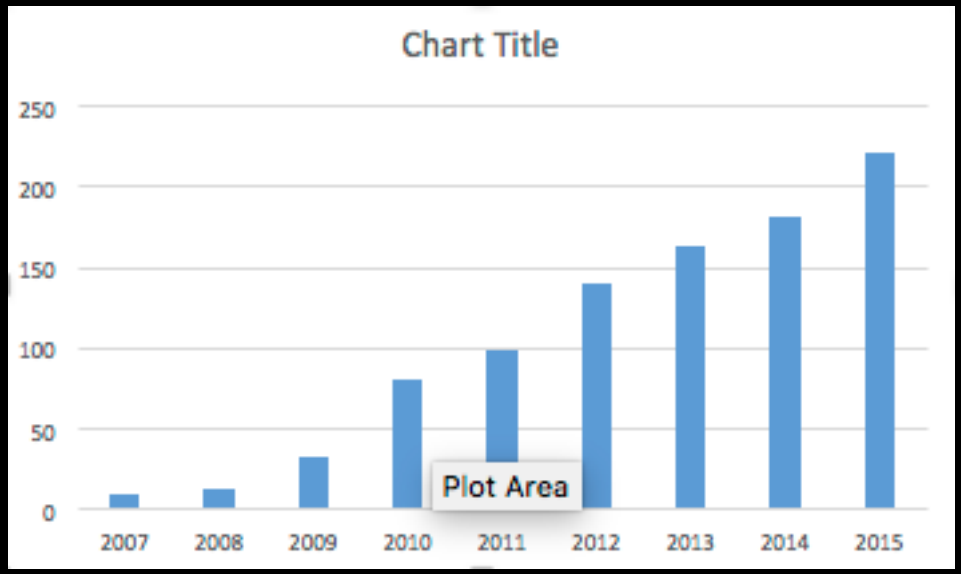
Pascal Unified Memory*

```
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

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



Contents

- Serial, Parallel, Hybrid Programming
- 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)

Serial Programming

- Languages
  - Fortran, C/C++, python, …
- Optimization of Serial Codes
  - Compilers,
  - opts flags,
  - math kernel, IPP (integrated Performance Primitives), ….
- Intel AVX (Advanced Vector Extensions)
  - 256-bit SIMD vector extension

Parallel Programming

- OpenMP
  - SMP, incremental parallelization
- CUDA (openCL)
  - Coprocessor, incremental parallelization
- MPI
  - Cluster, needs sometimes many changes of serial codes

Hybrid Programming

- OpenMP (MPI) + CUDA
  - SMP, multiple GPUs
- MPI + OpenMP
  - Cluster of SMP nodes
- MPI + CUDA
  - Cluster, multiple GPUs
- MPI + OpenMP + CUDA

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: 8-16GB/s
- Memory BW:

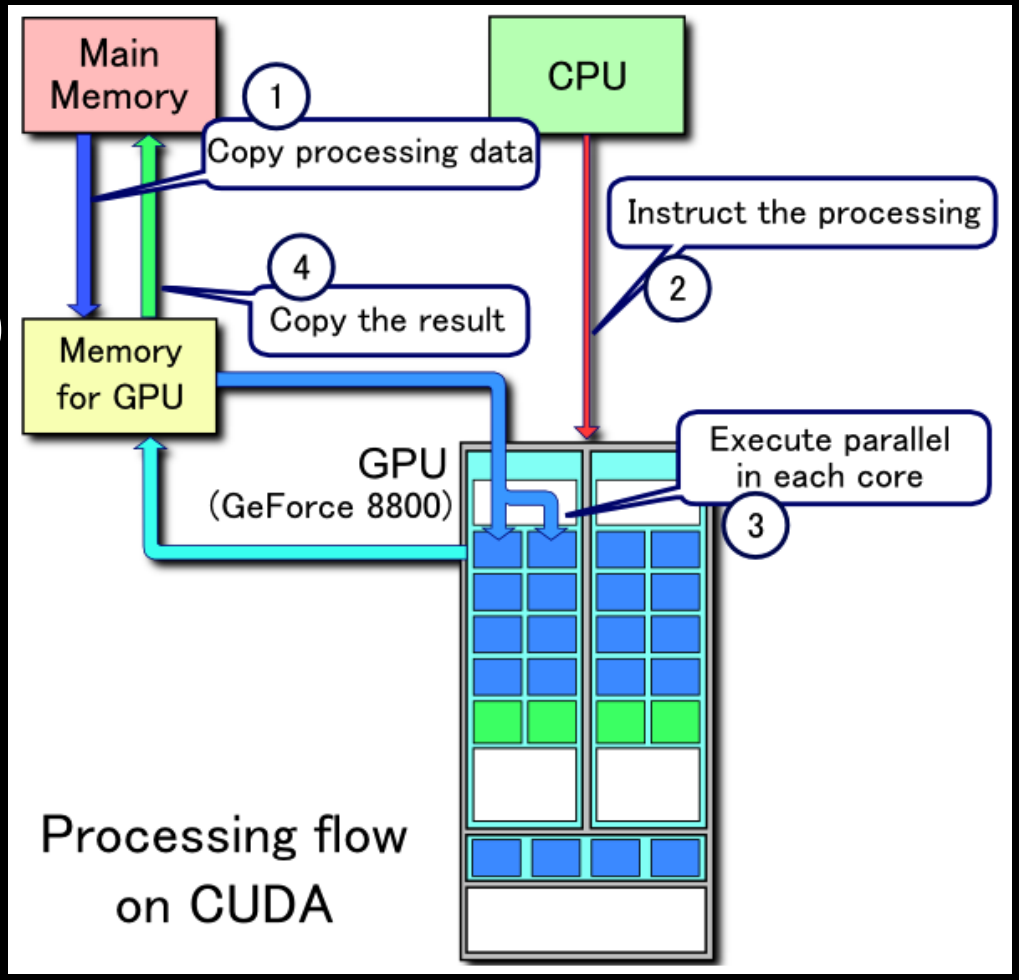
148 GB/s for Tesla S2050

288 GB/s for Tesla K 40

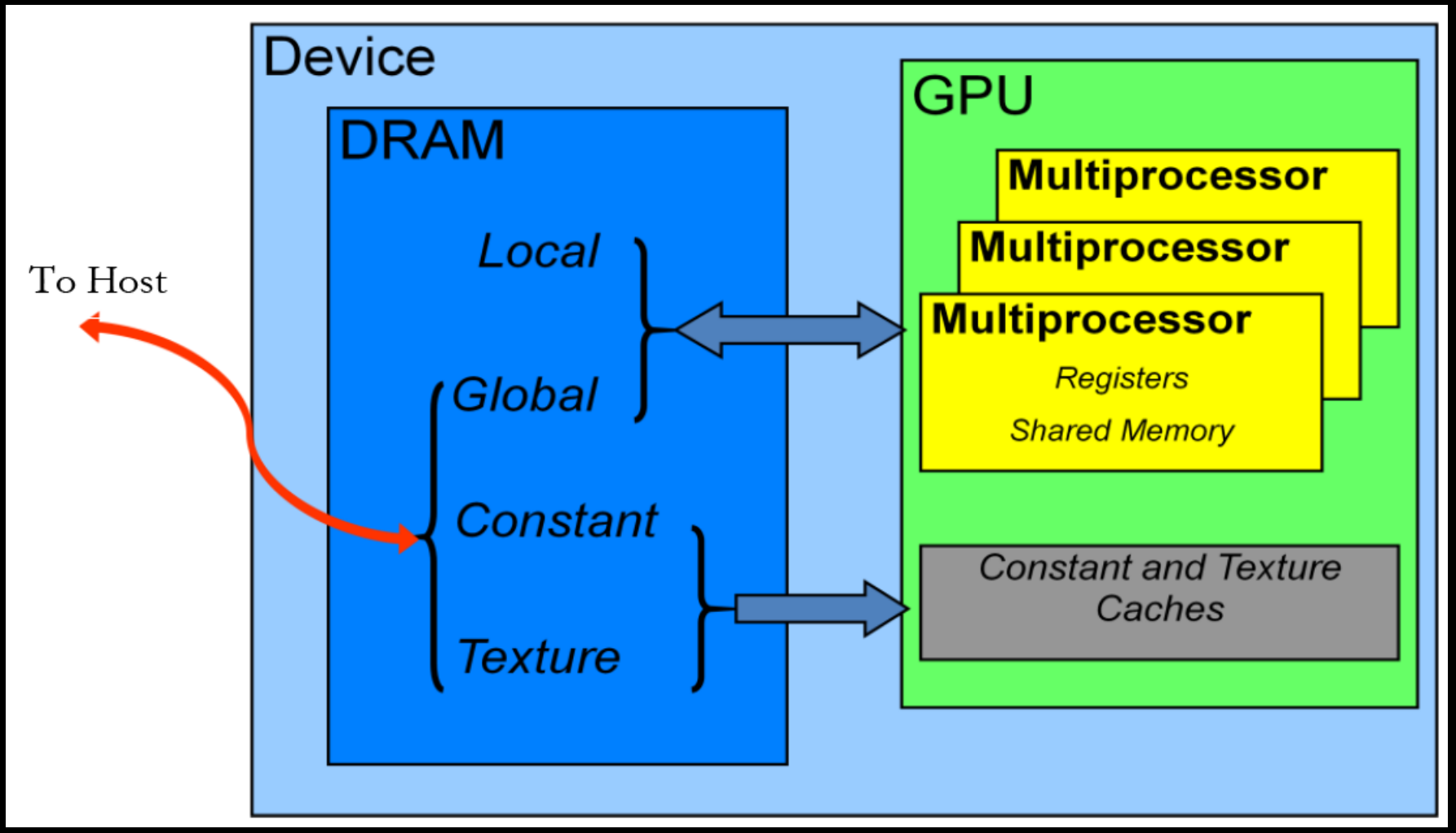
336.5 GB/s for Titan X

- Memory latency:

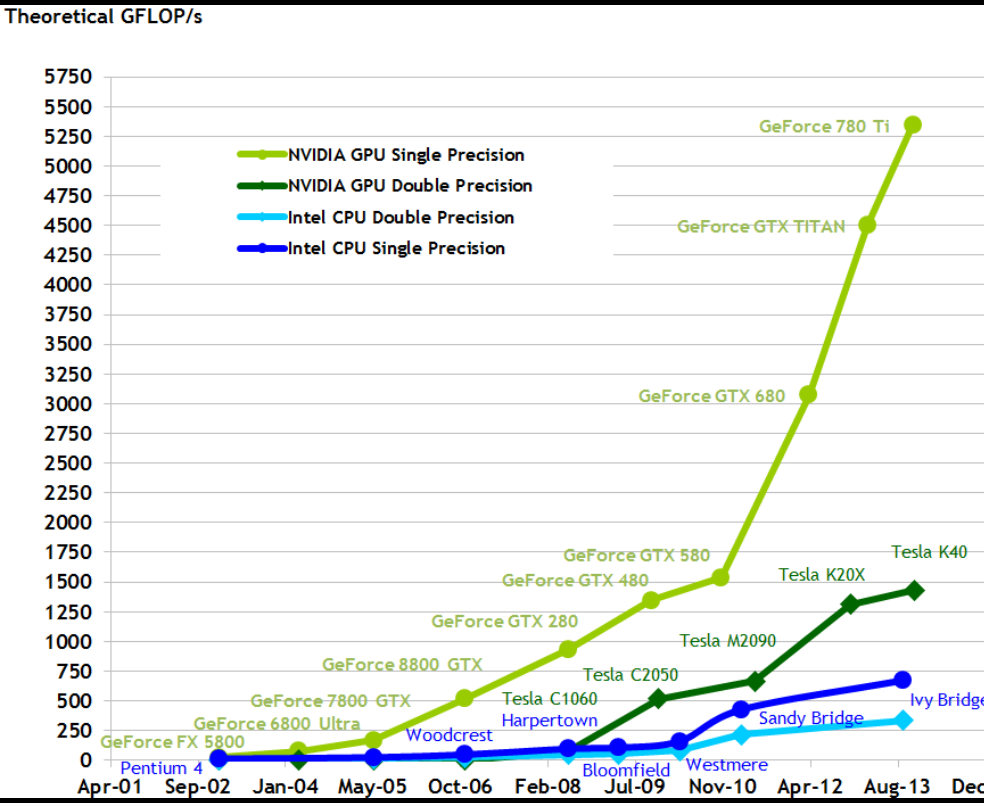
400-800 clock cycles

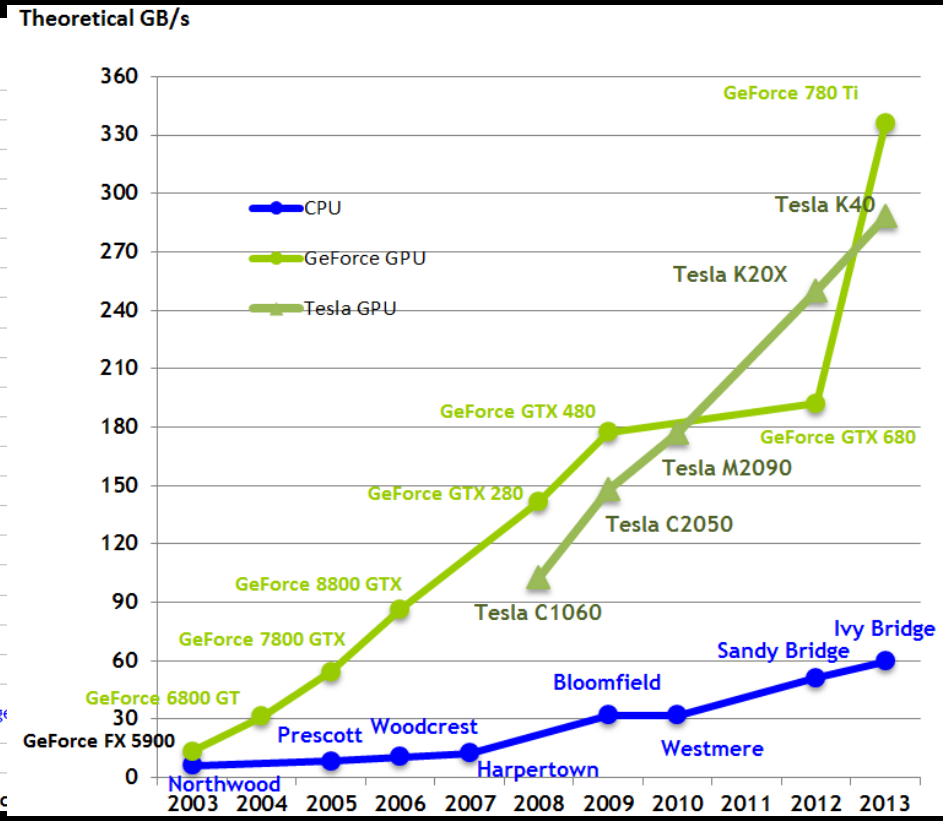


Device Memory Space



Performance and Bandwidth





Tesla S2050

- Tesla architecture
- cuda cores: 448
- Max GPU clock rate: 1.15 GHz
- Peak performance: ~ 1 TFLops
- Memory bandwidth: 148 GB/sec
- Memory: 2.6 GB

GPU and Xeon Phi servers in KASI

- Supermicro SYS-1028GQ-TRT
  - 2 Intel Xeon E5-2667 v3 3.2 GHz (16 cores)
  - 128 GB memory
  - 4 Gforce TiTan X GPUs
- Dell PE T630
  - 2 Intel Xeon E5-2667 v3 3.2 GHz (16 cores)
  - 128 GB memory
  - 1 Tesla K40m
  - 1 Xeon Phi 3120

Titan X

- Maxwell architecture
- CUDA cores: 3072
- base and boost clocks: 1000, 1075MHz
- performance: 6.14~6.6 Tflops single precision
- memory Bandwidth: 336.5 GB/ sec
- memory: 12GB

K40m

- Kepler architecture
- cores: 2880
- base, boost clocks: 745 MHz, 810 MHz and 875 MHz
- performance: 4.29~5.04 Tflops single precision
- memory bandwidth: 288 GB/ sec
- memory: 12 GB

Access to a KIAS GPU cluster

- From outside of the KIAS
  ssh cac@newton.kias.re.kr
- From inside of the KIAS or newton
  ssh guest@gpu.kias.re.kr-p 2020 passwd: kias#1503
  ssh gpu[01-24]
- mkdir –p userid/examples
- cd userid/examples
- cp ~guest/jskim/examples/* .

•ssh –p 2020 guest@gpu.kias.re.kr

Demo of nbody

- cd /home/jskim/NVIDIA_CUDA-7.5_Samples/5_Simulations/nbody
- ./nbody –benchmark –numdevices=4

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>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

Compile and Run

- nvcc –arch sm_20 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__

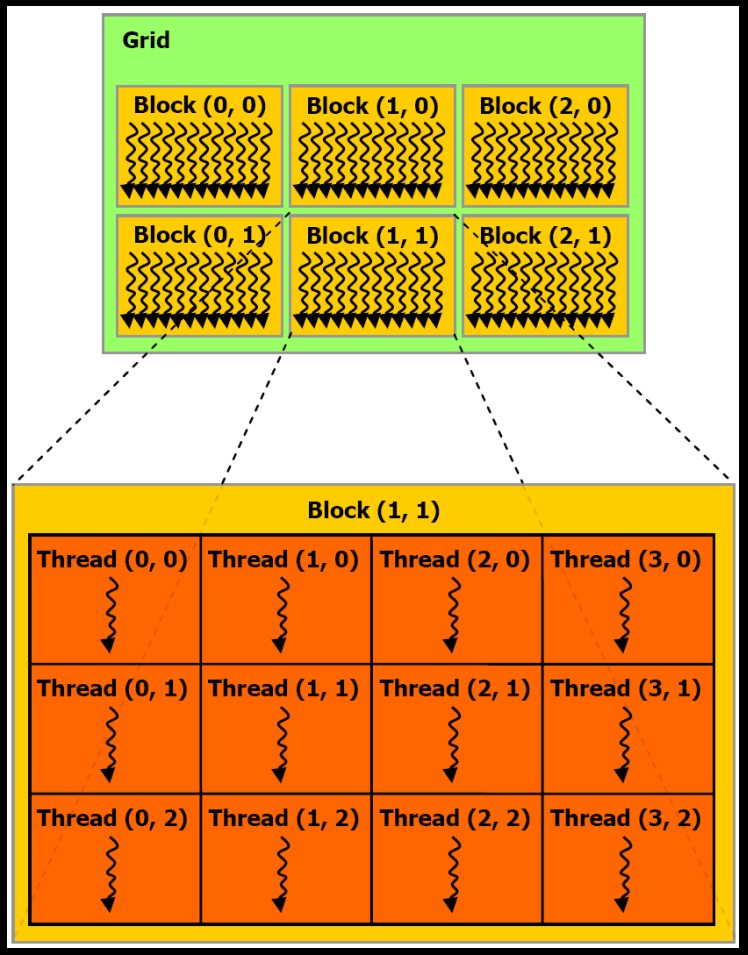
  executed on the device

  callable from the device only

  …

Grid, Block, Thread

- Tesla S2050
  - max. block size of each Dim per grid 65535x65535x1
  - max. thread size of each Dim per block 1024x1024x64
  - max. # of threads per block 1024



C Language Extensions

- Execution configuration

```
<<<blocksPerGrid,threadsPerBlock>>>
<<<1,1>>>
<<<65535,1024>

dim3 blocksPerGrid(65535,65535,1)
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 –arch sm_20 exec_conf_1d.cu
- nvcc –arch sm_20 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 –arch sm_30 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

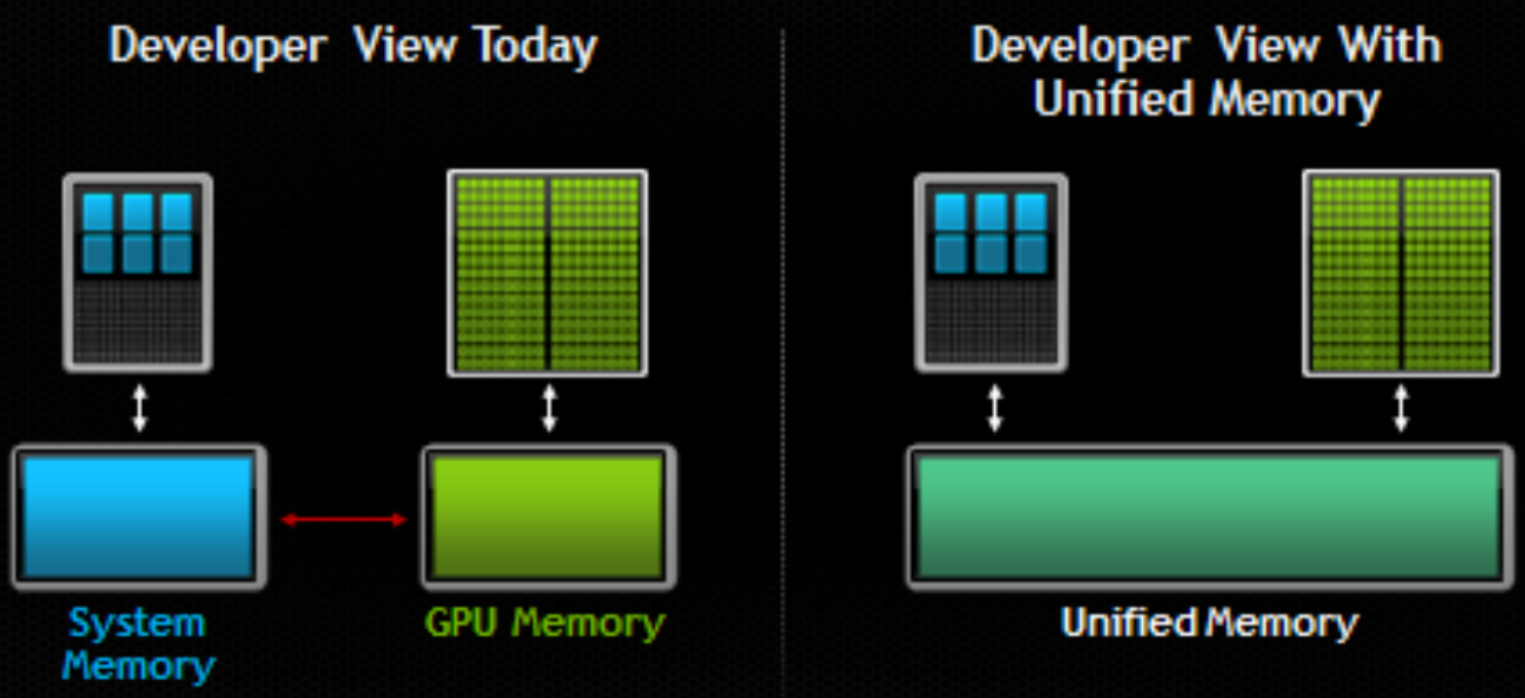
```
const int N=128;

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;
    }
}
```

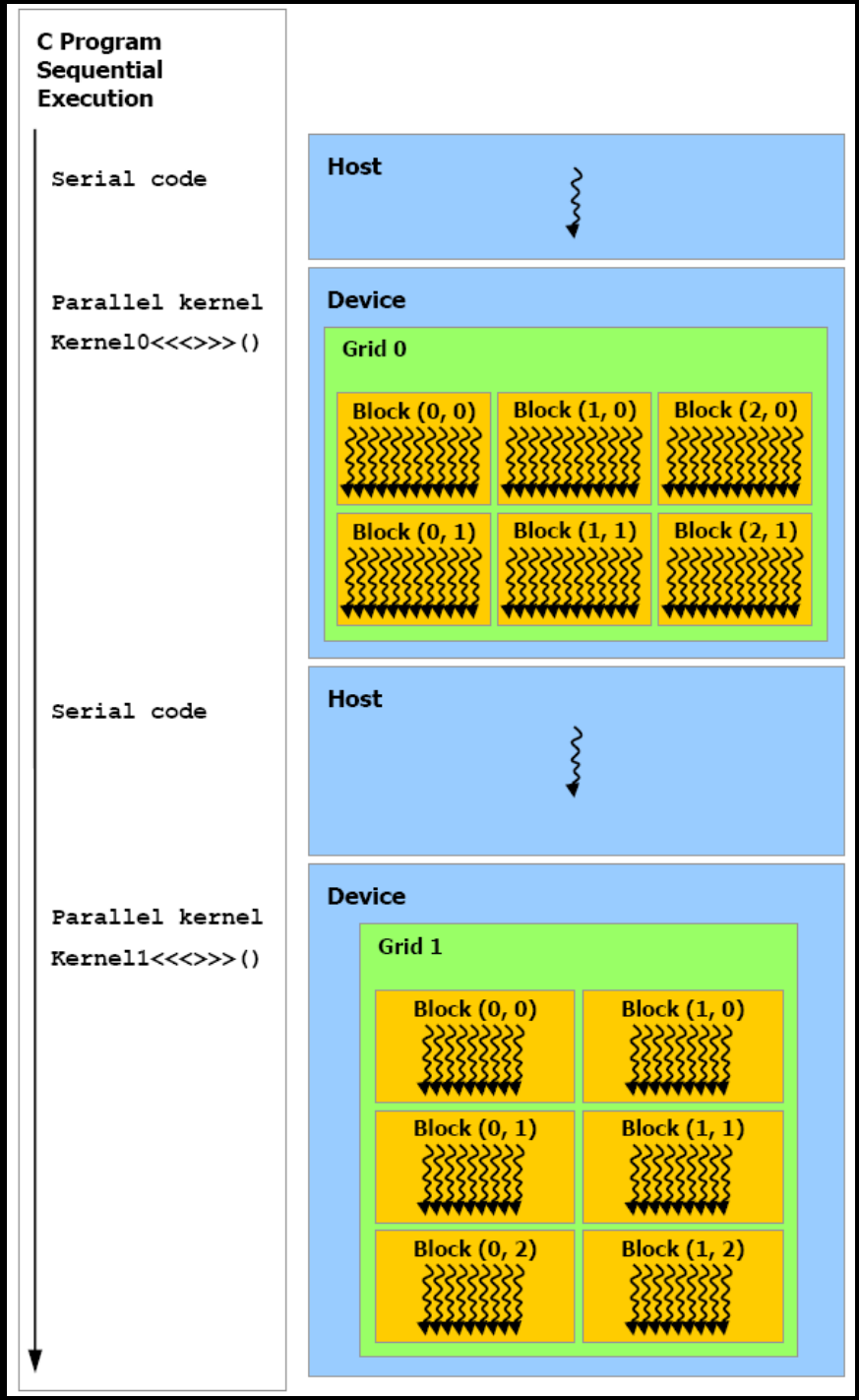
```
int main (void) {
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;

    // 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

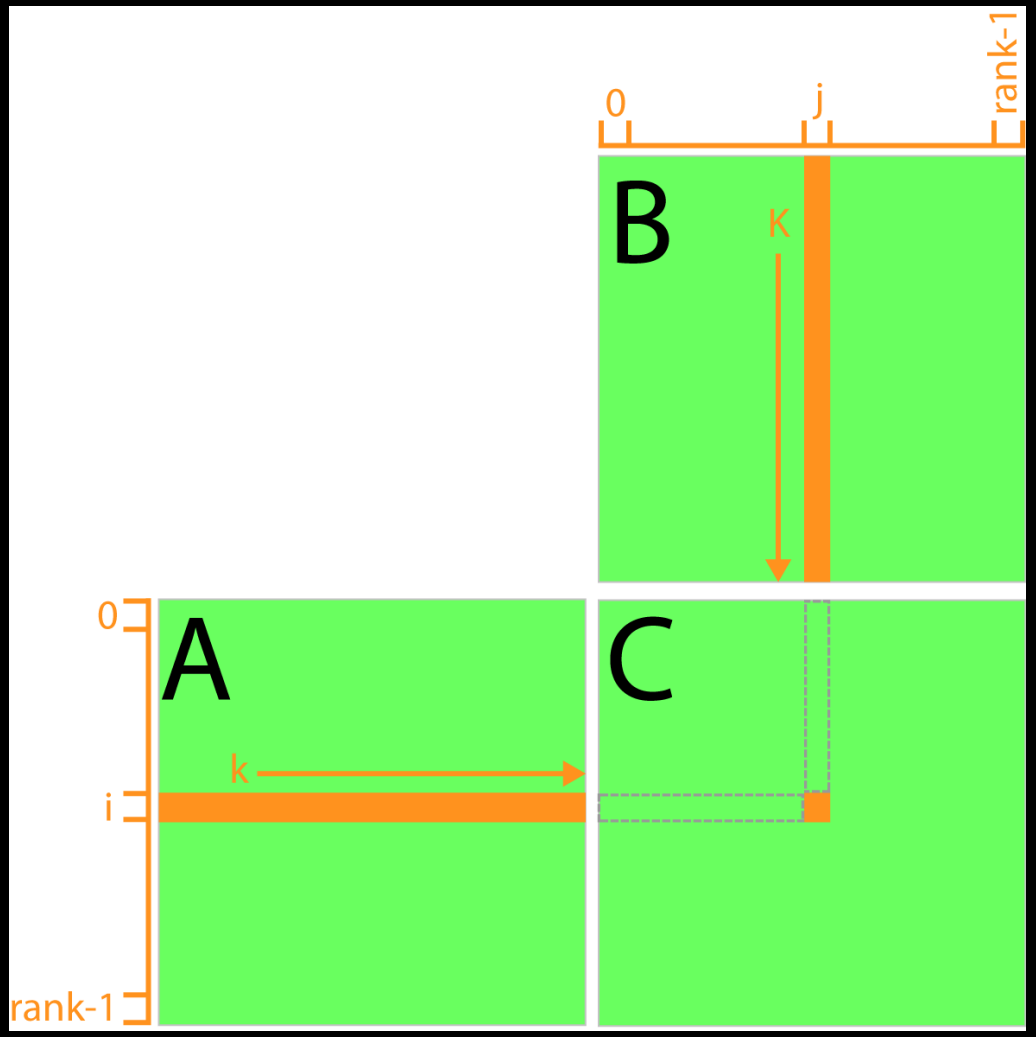
Exercise 5: Matrix Multiplication

- 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

	Compute Capability							
Technical Specifications	**1.0**	**1.1**	**1.2**	**1.3**	**2.x**	**3.0**	**3.5**	**5.0**
Maximum dimensionality of grid of thread blocks	2				3			
Maximum x-dimension of a grid of thread blocks	65535					$2^{31}-1$		
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	512				1024			
Maximum z-dimension of a block	64							
Maximum number of threads per block	512				1024			
Warp size	32							
Maximum number of resident blocks per multiprocessor	8					16		32
Maximum number of resident warps per multiprocessor	24		32		48	64		
Maximum number of resident threads per multiprocessor	768		1024		1536	2048		
Number of 32-bit registers per multiprocessor	8 K		16 K		32 K	64 K		
Maximum number of 32-bit registers per thread	128				63		255	
Maximum amount of shared memory per multiprocessor	16 KB				48 KB			64 KB
Maximum amount of shared memory per thread block	16 KB				48 KB			
Number of shared memory banks	16				32			
Amount of local memory per thread	16 KB				512 KB			
Constant memory size	64 KB							
Cache working set per multiprocessor for constant memory	8 KB							10 KB
Cache working set per multiprocessor for texture memory	Device dependent, between 6 KB and 8 KB				12 KB		Between 12 KB and 48 KB	
Maximum width for a 1D texture reference bound to a CUDA array	8192				65536			

Cuda version, DeviceQuery

/home/guest/NVIDIA_CUDA-7.5_Samples/1_Utilities/deviceQuery/deviceQuery

- Device 0: "Tesla S2050"
- CUDA Driver Version / Runtime Version 7.5 / 7.5
- CUDA Capability Major/Minor version number: 2.0
- Total amount of global memory: 2687 MBytes (2817982464 bytes)
- (14) Multiprocessors, (32) CUDA Cores/MP: 448 CUDA Cores
- GPU Max Clock rate: 1147 MHz (1.15 GHz)
- Memory Clock rate: 1546 Mhz
- Memory Bus Width: 384-bit
- L2 Cache Size: 786432 bytes

S2050 compute capability 2.0

- 448 cores, 32core/MP, 14MP
- Maximum number of threads per block: 1024 or 32x32
- Maximum shared memory per block: 48kB
- Number of 32bit registers per thread: 63
- Constant memory size: 64kB
- Maximum resident blocks per MP: 8

BlockDim, GridDim for S2050

- MxM with a size of 1024x1024
  - BlockDim = 32x32 (total thread number1024)
  - GridDim = 32x32
  - Number of registers per thread used
    ~ 5 << 63
  - Shared memory for two tiles with blockDim
    2*32x32*4 Byte = 8kB < 48kB
- MxM with a size of NxN
  - BlockDim = 32x32 (total thread number1024)
  - GridDim = (N/BlockDim.x,N/BlockDim.y)

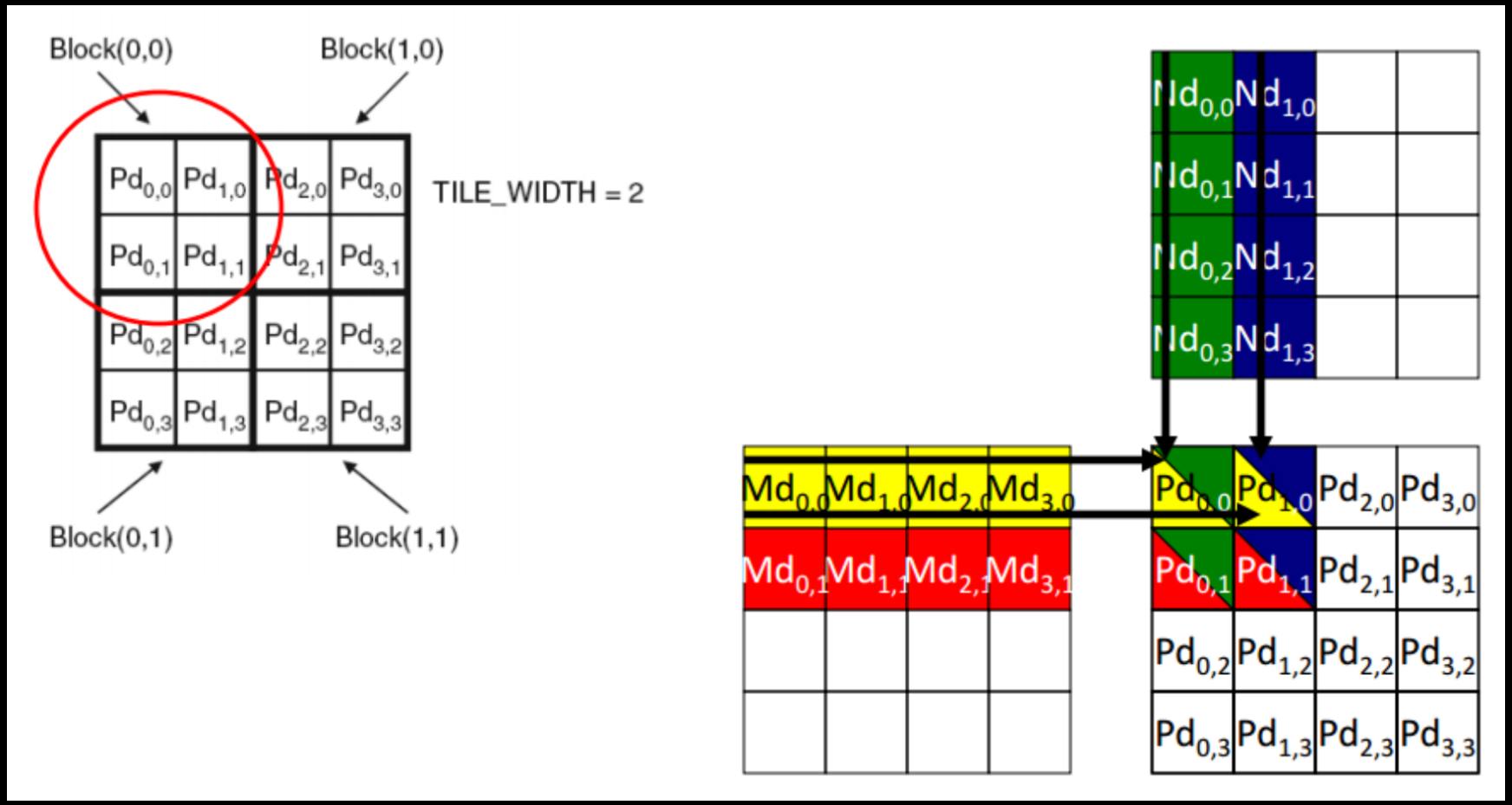
AI for MatMul using global memory

- Cvalue += A[row*N+k]*B[k*N+col];
- 2 flop/8Bytes = 0.25 flop/Byte
- Performance
  - S2050
    - 0.25 flop/Byte * 148GB/sec ~ 37 Gflop/s
  - K40
    - 0.25 flop/Byte * 288GB/sec ~ 72 Gflop/s

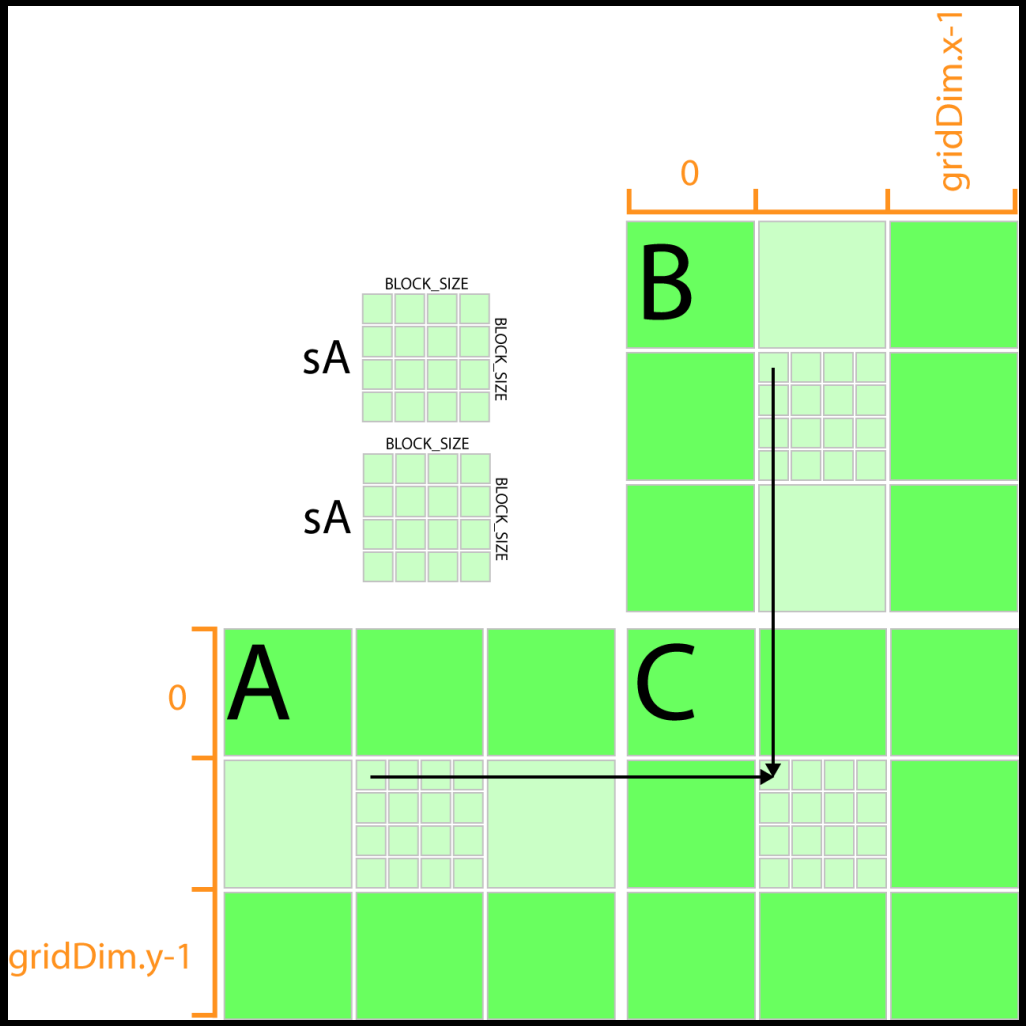
Reducing Global Memory Traffic

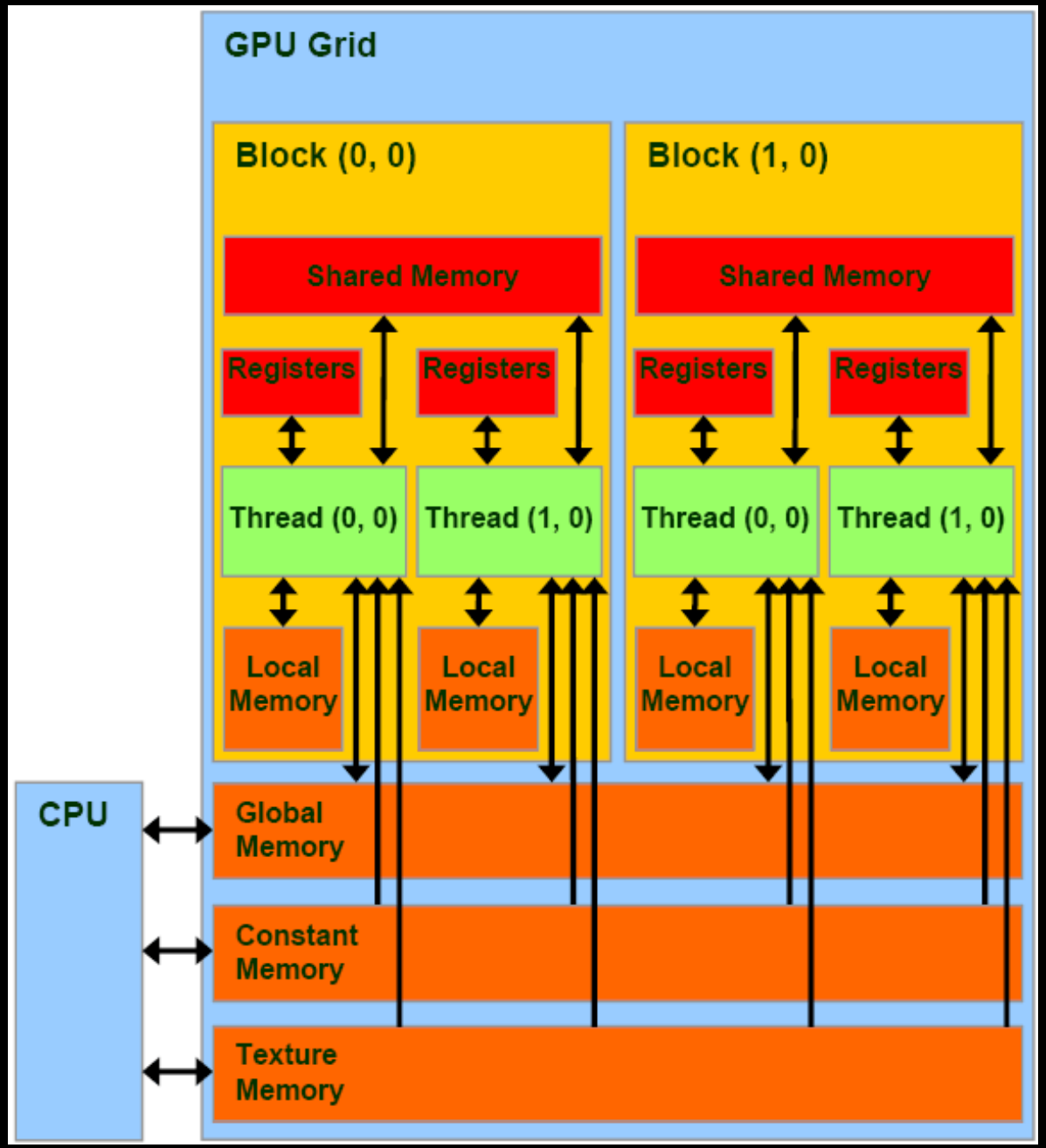
- Reducing global memory access enhances performance
- tiling: partition of data into subsets called tiles, such that each tile fits into fast (shared) memory

Matrix Mul: tiling



Matrix-Multiplication: shared

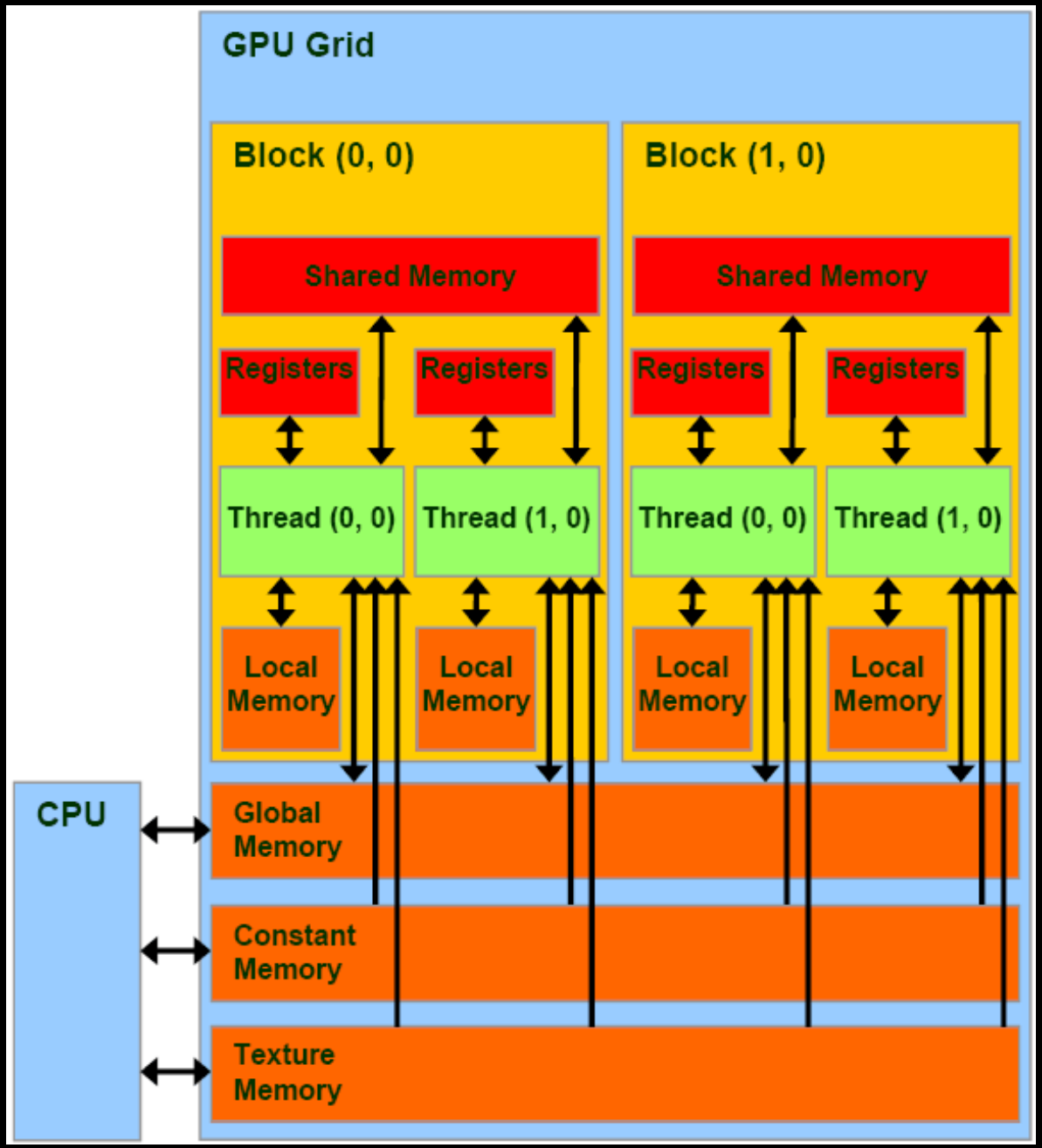




Register
- Fastest
- Thread scope, thread lifetime

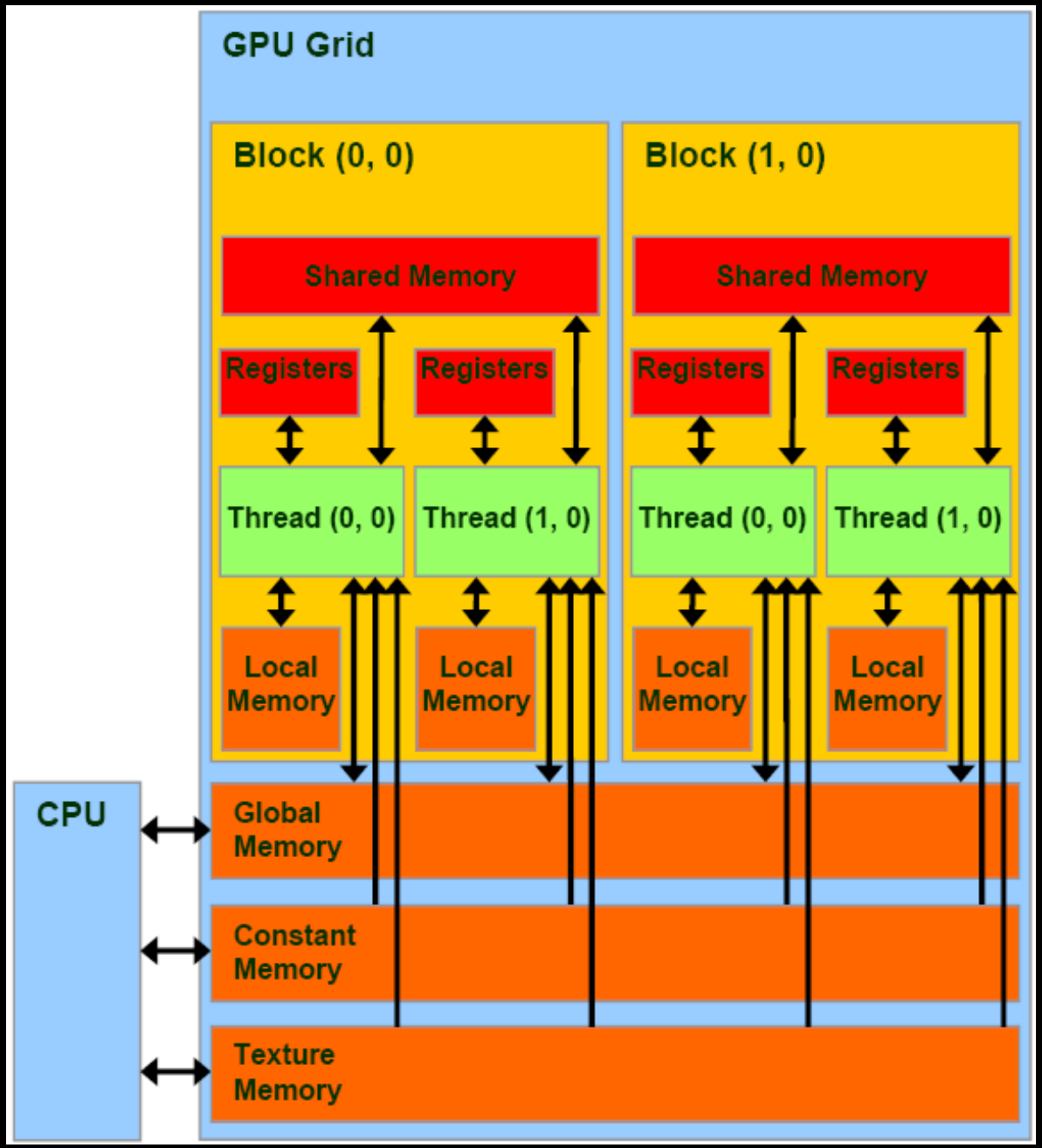
Local
- Does not exist
- Abstraction of thread scope

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



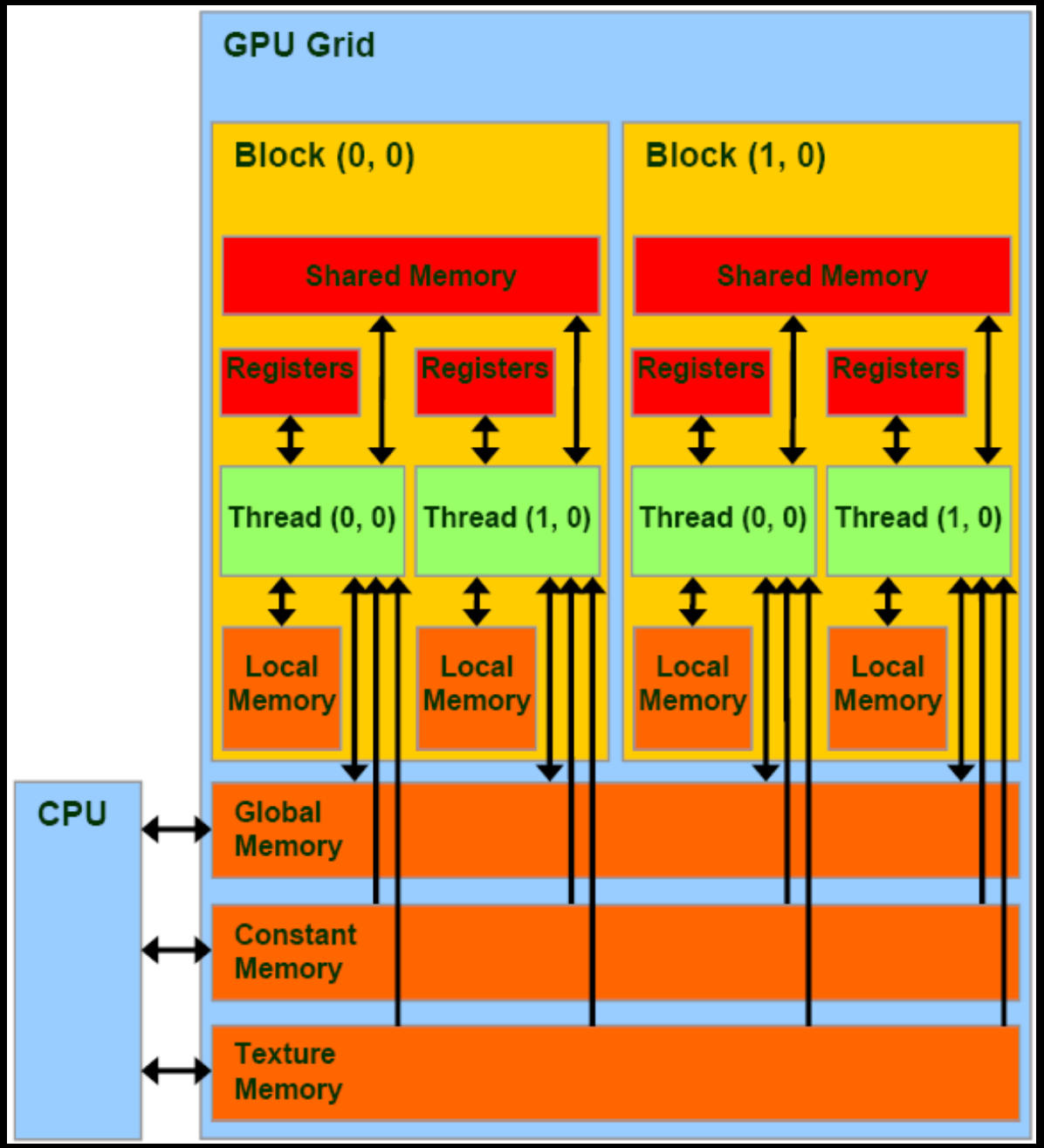
Shared memory

- 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
```

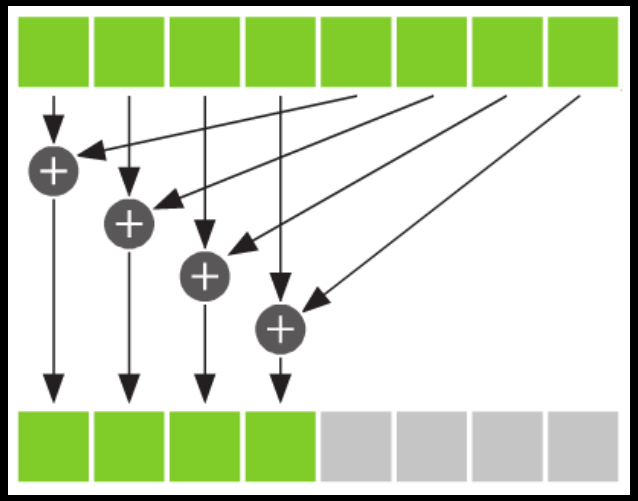
Exercise 4: Dot Product

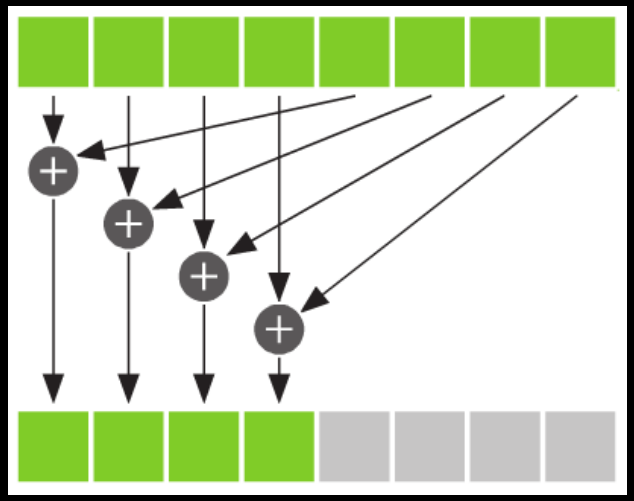
```
#include <stdio.h>
#define sum_squares(x) (x*(x+1)*(2*x+1)/6)

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

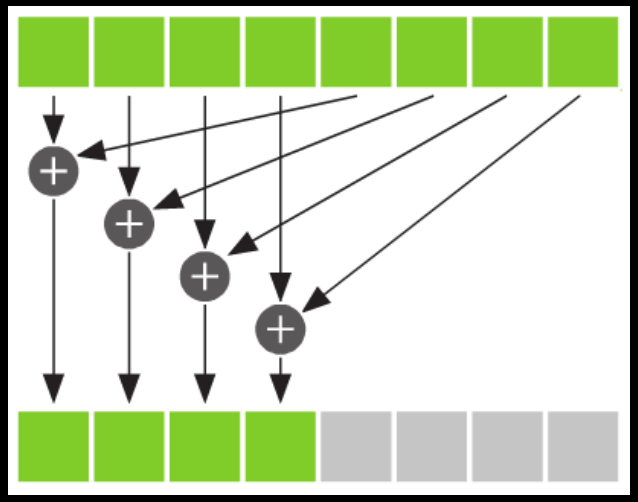
blockIdx 0

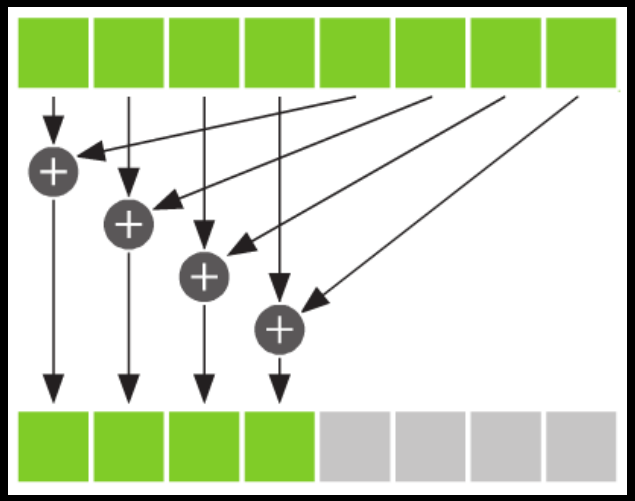




blockIdx 1

blockIdx 2





blockIdx 3

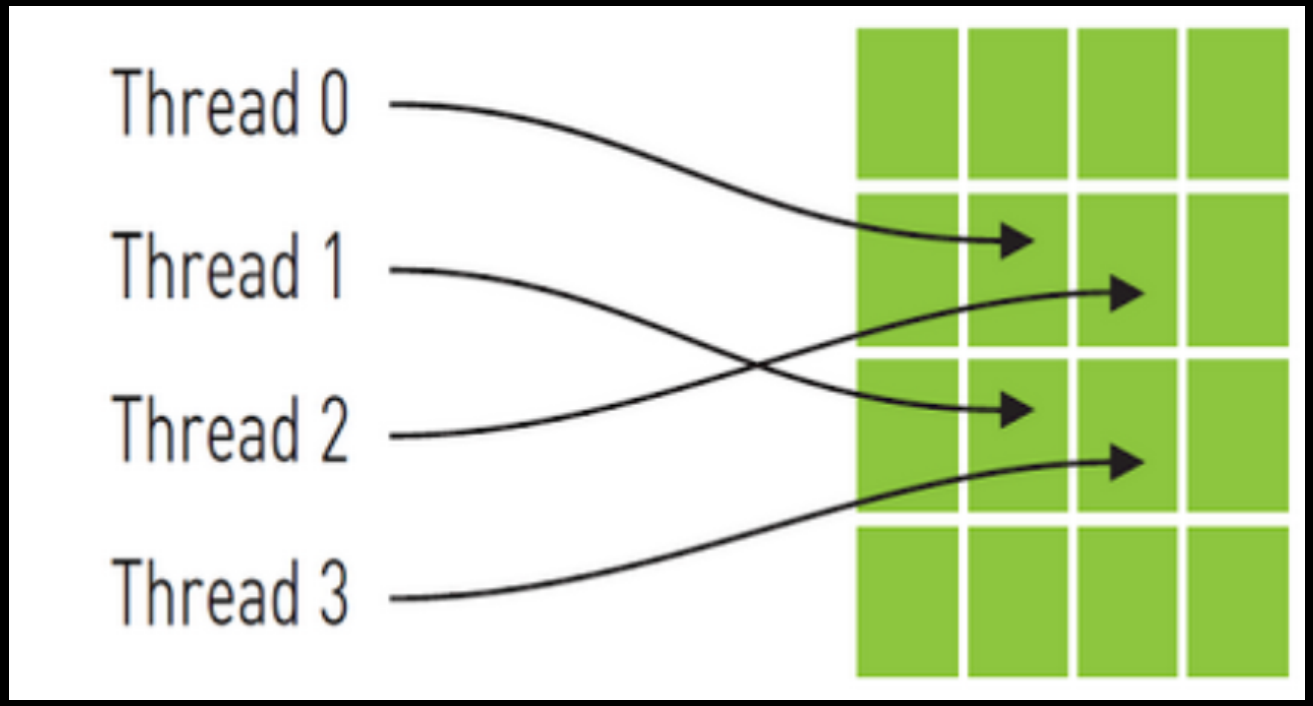
cuBLAS

- 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_20 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

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

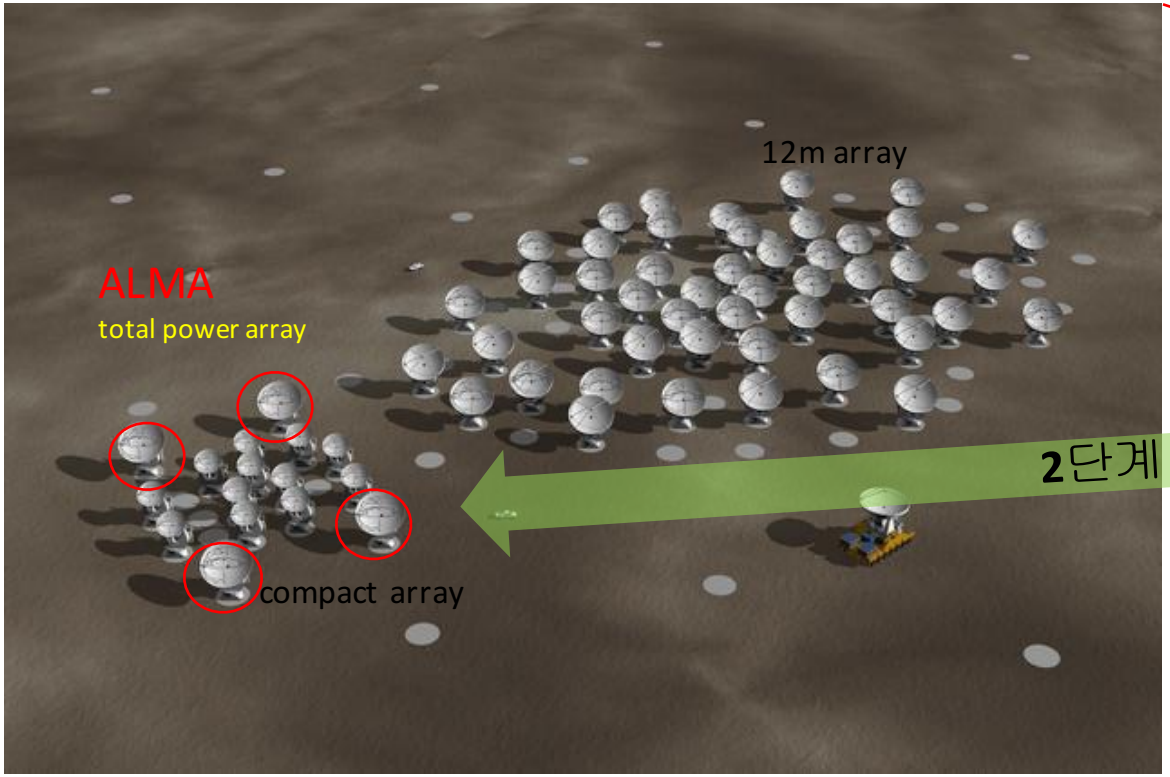
$$H(f) = \int_{-T_0}^{T_0} Ae^{-j2\pi ft} dt$$

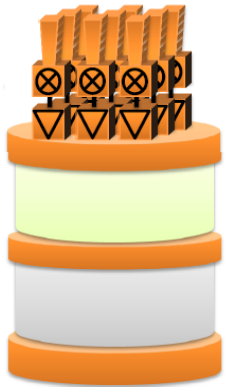
$$= A\int_{-T_0}^{T_0} \cos(2\pi ft)\, dt - jA\int_{-T_0}^{T_0} \sin(2\pi ft)\, dt = \frac{A}{2\pi f}\sin(2\pi ft)\,|_{-T_0}^{T_0}$$

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

기기 개발 계획

- 2015-2018 ASTE 다중빔 수신기 제작
- 2019-2020 ALMA 다중빔 수신기 시작품 제작
- 2021-2024 ALMA 다중빔 수신기 제작



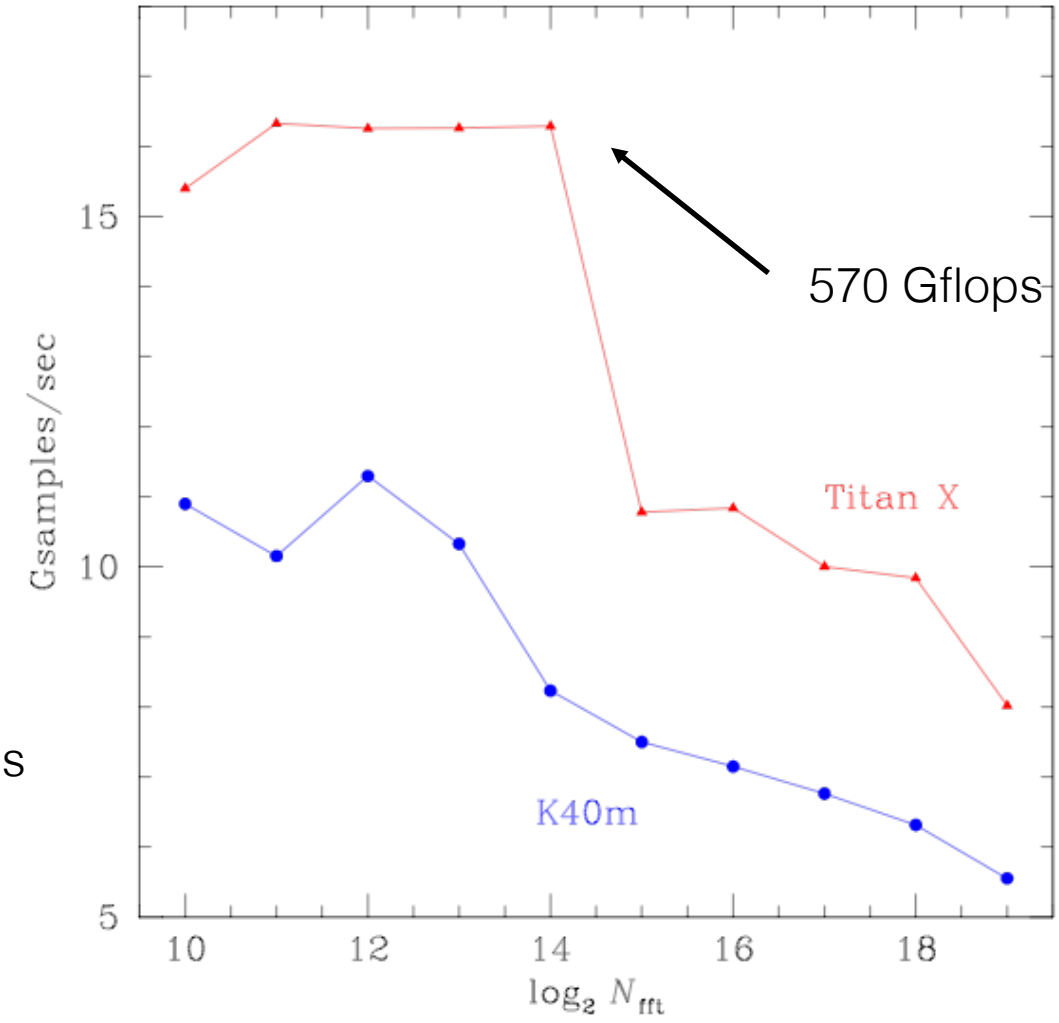


band 7, band 8 차세대 다중빔 수신기(총 8+2 개)

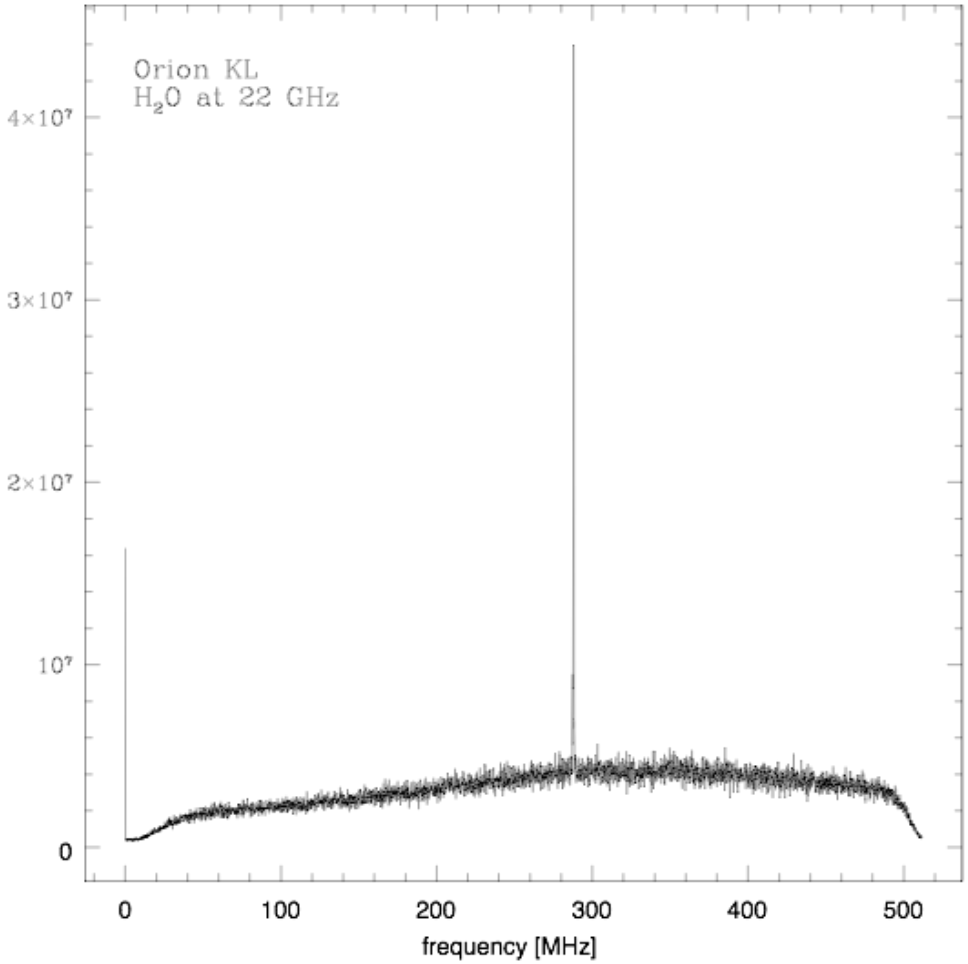
cuFFT

```
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

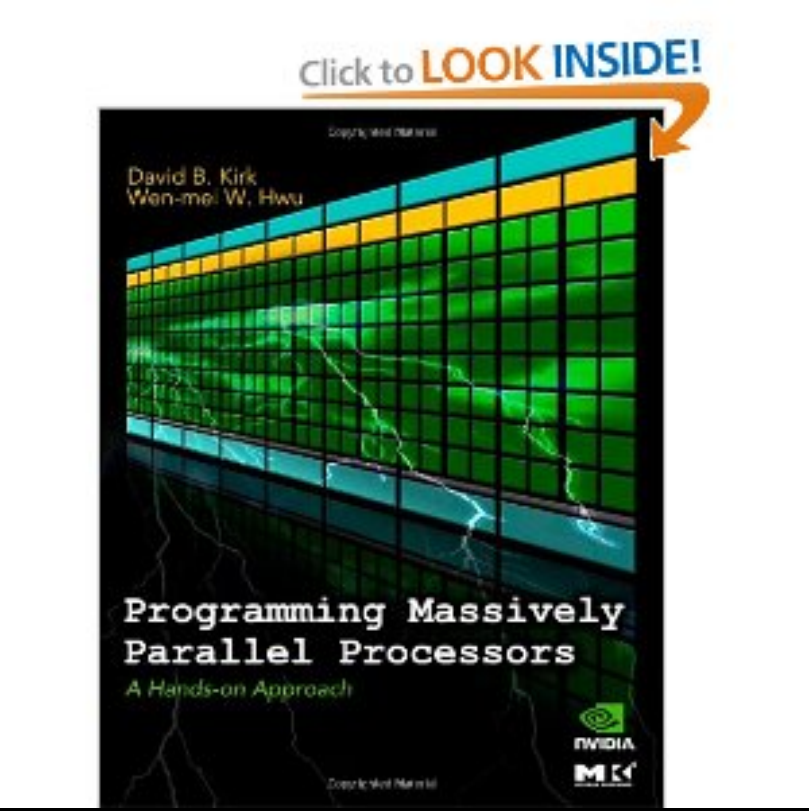


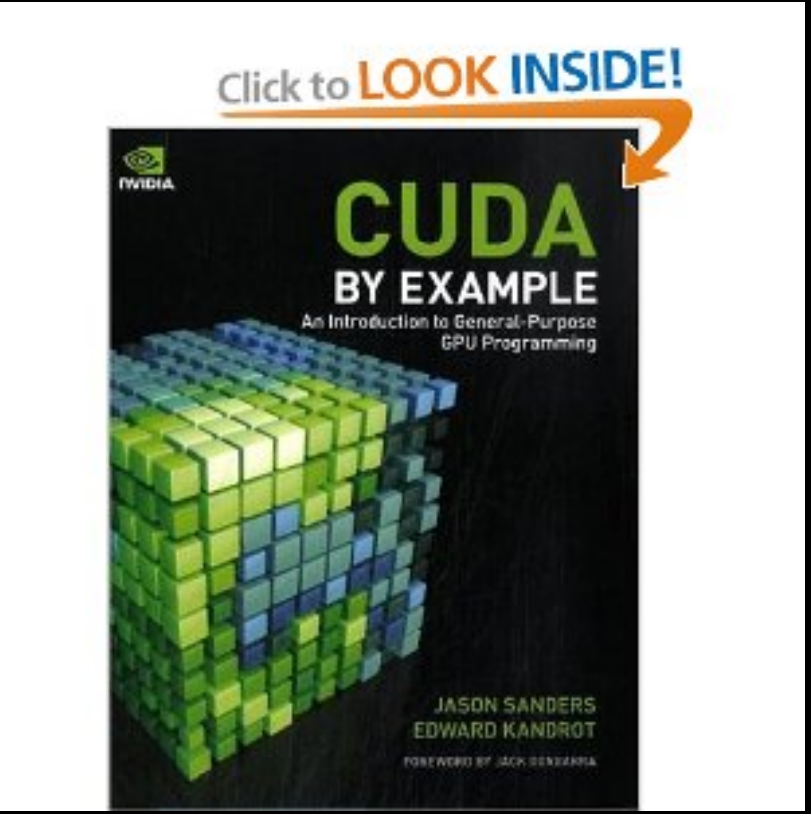
Test Data:
KVN 22GHz H2O maser line observation



- recorded data with 512 MHz bandwidth
- 2 bits / sample

Cuda Books


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


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

Useful information

- http://developer.nvidia.com
- CUDA toolkit 7.5
- Getting Started Guide (installation)
- CUDA C programming Guide