High-performance GPU Computing

Stan Tomov
Research Asst. Professor

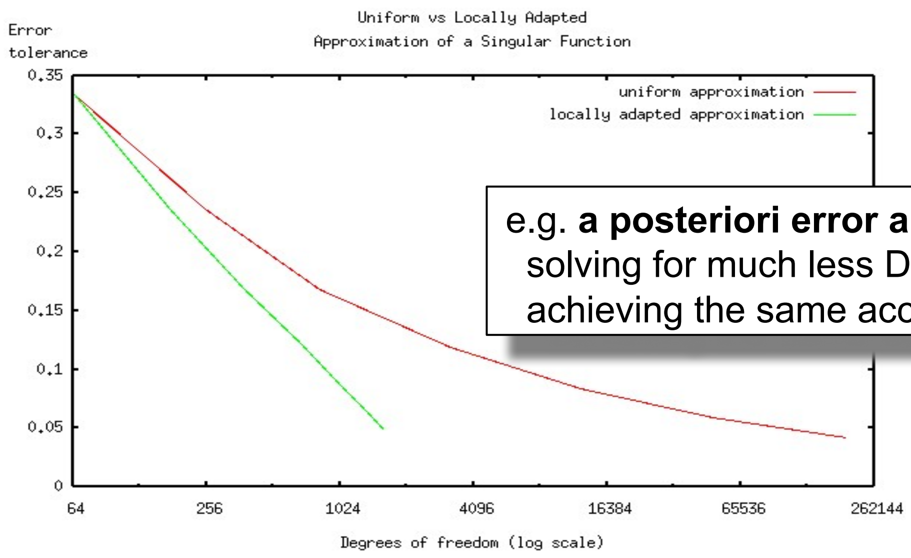
The Innovative Computing Laboratory
Department of Electrical Engineering and Computer Science
University of Tennessee, Knoxville

2017 Summer Research Experiences for Undergraduate (REU)
Research Experiences in Computational Science, Engineering, and Mathematics (RECSEM)
Knoxville, TN

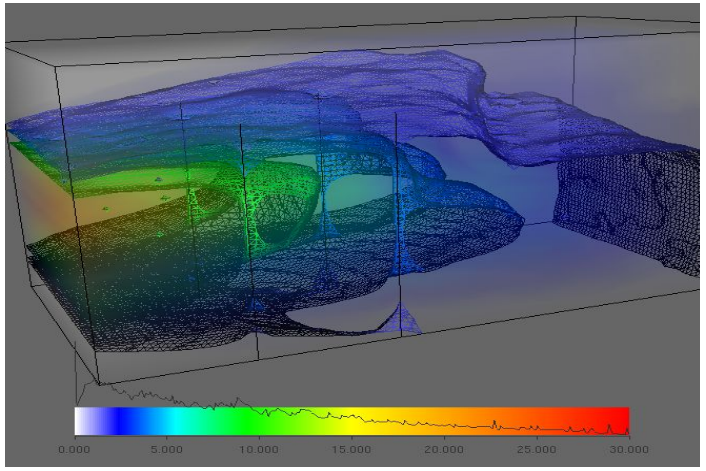
THE UNIVERSITY of TENNESSEE
Department of Electrical Engineering
and Computer Science

Speeding up Computer Simulations

- **Better numerical methods**

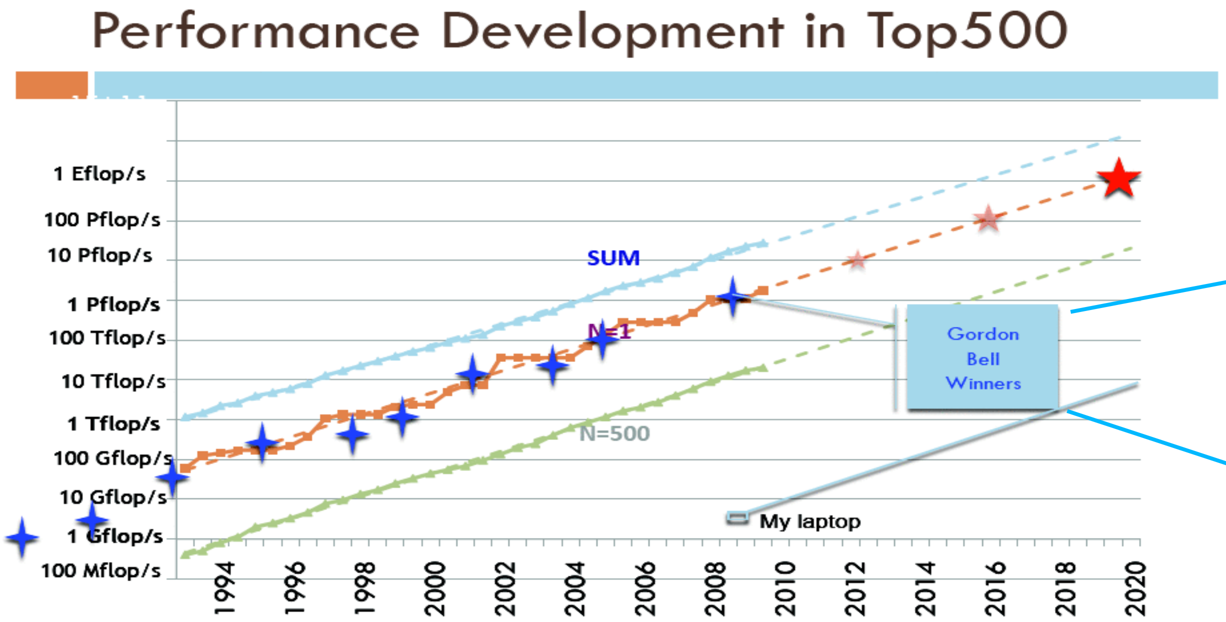


e.g. **a posteriori error analysis**: solving for much less DOF but achieving the same accuracy



http://www.cs.utk.edu/~tomov/cflow/

- **Exploit advances in hardware**



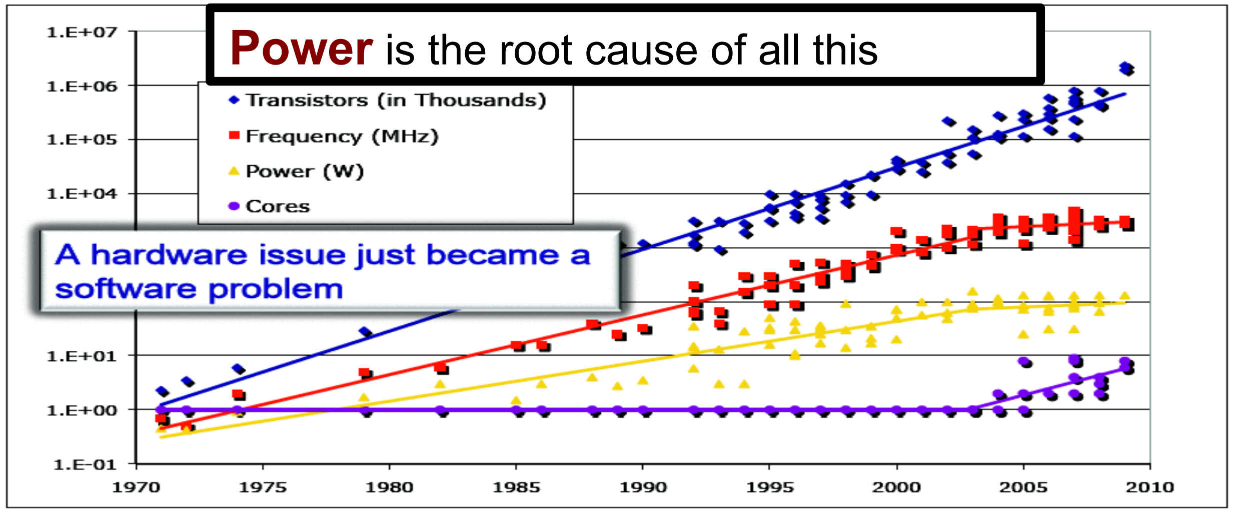
- Manage to use hardware efficiently for real-world HPC applications
- Match LU benchmark in performance !

Hardware trends and GPUs

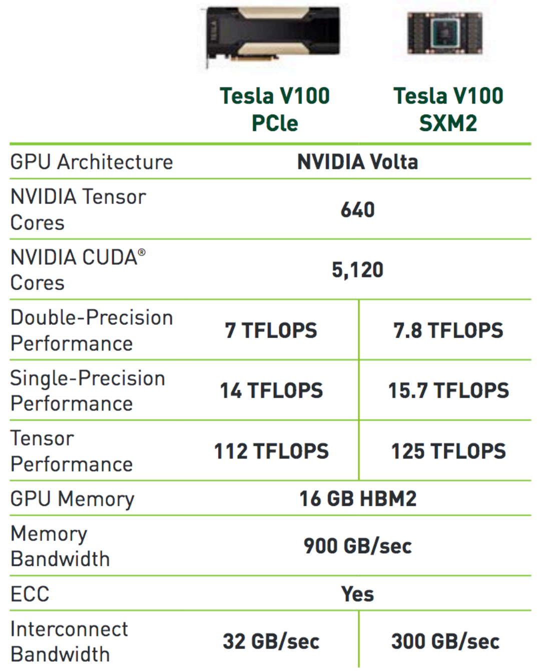
Hardware trends

- Multicore
- GPUs

Performance Has Also Slowed, Along with Power



(Source: slide from Kathy Yelick)

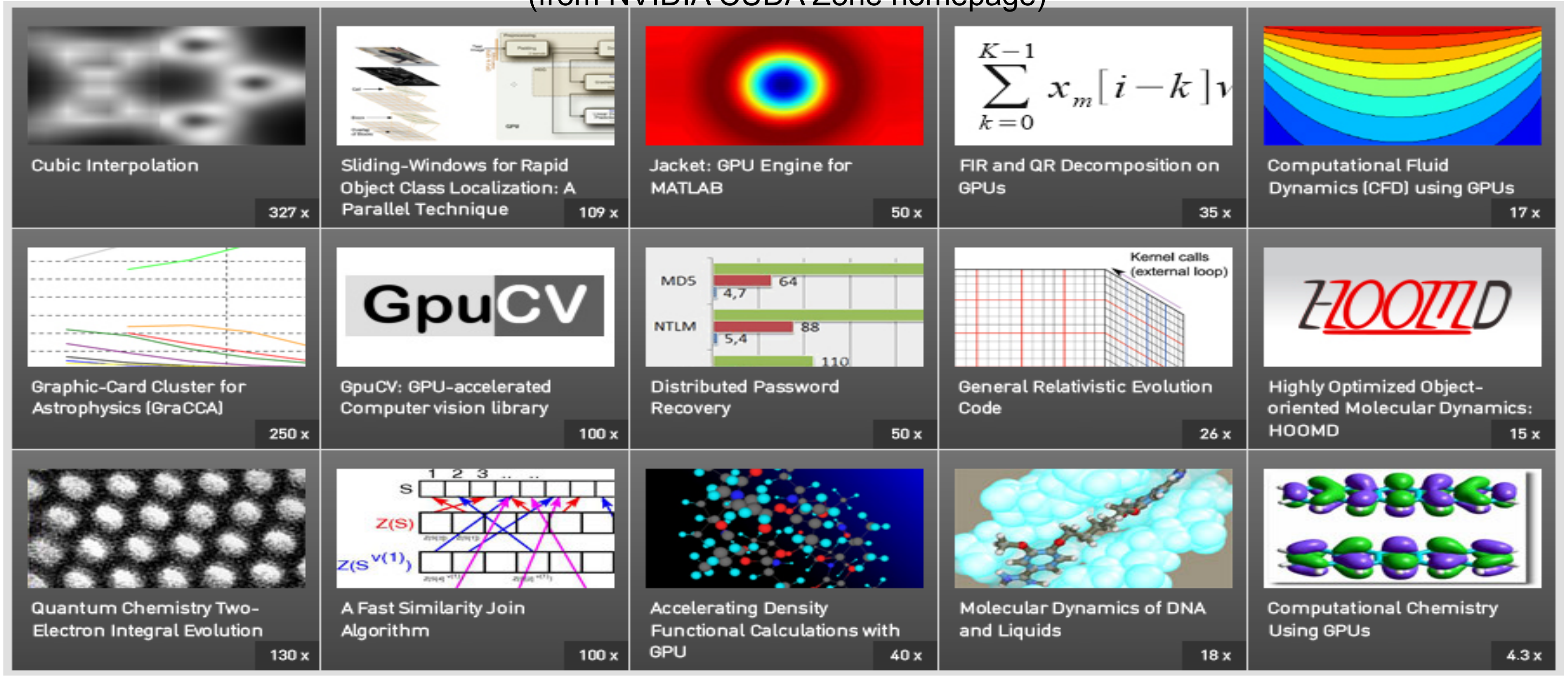


	Tesla V100 PCle	Tesla V100 SXM2
GPU Architecture	NVIDIA Volta	
NVIDIA Tensor Cores	640	
NVIDIA CUDA® Cores	5,120	
Double-Precision Performance	7 TFLOPS	7.8 TFLOPS
Single-Precision Performance	14 TFLOPS	15.7 TFLOPS
Tensor Performance	112 TFLOPS	125 TFLOPS
GPU Memory	16 GB HBM2	
Memory Bandwidth	900 GB/sec	
ECC	Yes	
Interconnect Bandwidth	32 GB/sec	300 GB/sec

Main software development issues

- **Increase in parallelism** *1
- **Increase in communication cost (vs computation)** *2
- **Hybrid Computing** *3

Despite issues, **high speedups** on HPC applications are reported using GPUs
(from NVIDIA CUDA Zone homepage)



Cubic Interpolation 327 x	Sliding-Windows for Rapid Object Class Localization: A Parallel Technique 109 x	Jacket: GPU Engine for MATLAB 50 x	FIR and QR Decomposition on GPUs 35 x	Computational Fluid Dynamics (CFD) using GPUs 17 x
Graphic-Card Cluster for Astrophysics (GraCCA) 250 x	GpuCV: GPU-accelerated Computer vision library 100 x	Distributed Password Recovery 50 x	General Relativistic Evolution Code 26 x	Highly Optimized Object-oriented Molecular Dynamics: HOOMD 15 x
Quantum Chemistry Two-Electron Integral Evolution 130 x	A Fast Similarity Join Algorithm 100 x	Accelerating Density Functional Calculations with GPU 40 x	Molecular Dynamics of DNA and Liquids 18 x	Computational Chemistry Using GPUs 4.3 x

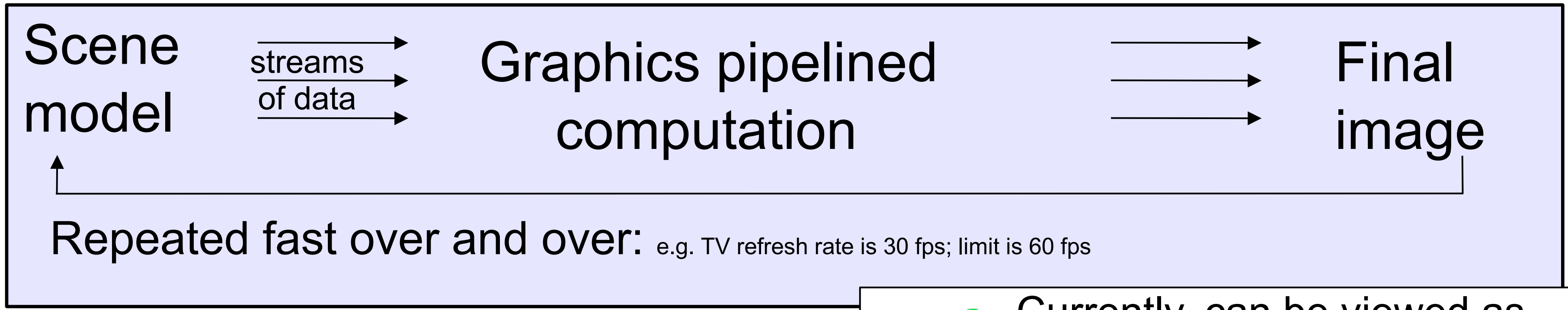
<u>**CUDA architecture & programming**</u>: *1
- A data-parallel approach that scales
- Similar amount of efforts on using CPUs *vs* GPUs by domain scientists demonstrate the GPUs' potential

Processor speed improves 59% / year but memory bandwidth by 23% latency by 5.5% *2

e.g., schedule small non-parallelizable tasks on the CPU, and large and paralelizable on the GPU *3

Evolution of GPUs

GPUs: excelling in graphics rendering


Scene model
streams of data
Graphics pipelined computation
Final image
Repeated fast over and over: e.g. TV refresh rate is 30 fps; limit is 60 fps

- Currently, can be viewed as **multithreaded multicore vector units**

This type of computation:

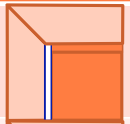
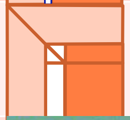
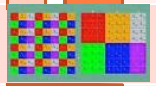
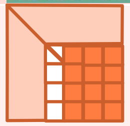
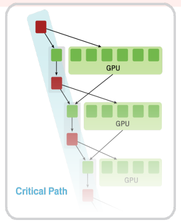
- Requires **enormous computational power**
- Allows for **high parallelism**
- Needs **high bandwidth** ***vs*** **low latency**

(as low latencies can be compensated with deep graphics pipeline)

Obviously, this pattern of computation is common with many other applications

Programming model

- **CUDA**
  Compute Unified Device Architecture
- **BLAS**
- **Hybrid algorithms**

Software/Algorithms follow hardware evolution in time		
LINPACK (70's) (Vector operations)		Level 1 **BLAS**
LAPACK (80's) (Blocking, cache friendly)		Level 3 **BLAS**
ScaLAPACK (90's) (Distributed Memory)		**PBLAS**
PLASMA (00's) New Algorithms (many-core friendly)		**BLAS** on tiles + DAG scheduling
MAGMA Hybrid Algorithms (heterogeneity friendly)		**BLAS** tasking + (CPU / GPU / Xeon Phi) hybrid scheduling

Algorithms expressed in terms of various BLAS calls

CUDA references

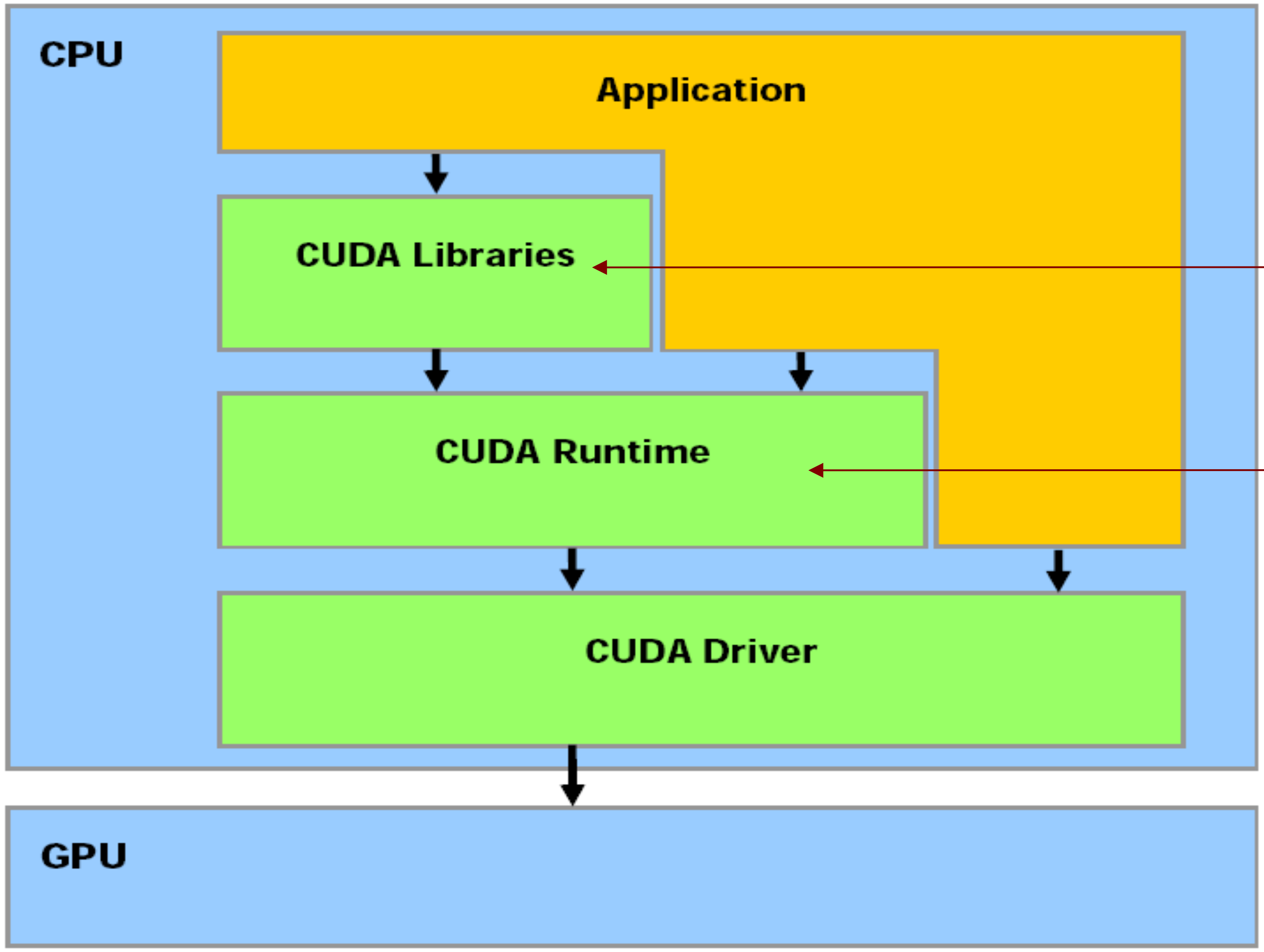
- **NVIDIA CUDA Programming Guide**

  https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

- **CUDA Samples**

  Typically installed with CUDA, e.g., in /usr/local/cuda/samples

CUDA Software Stack

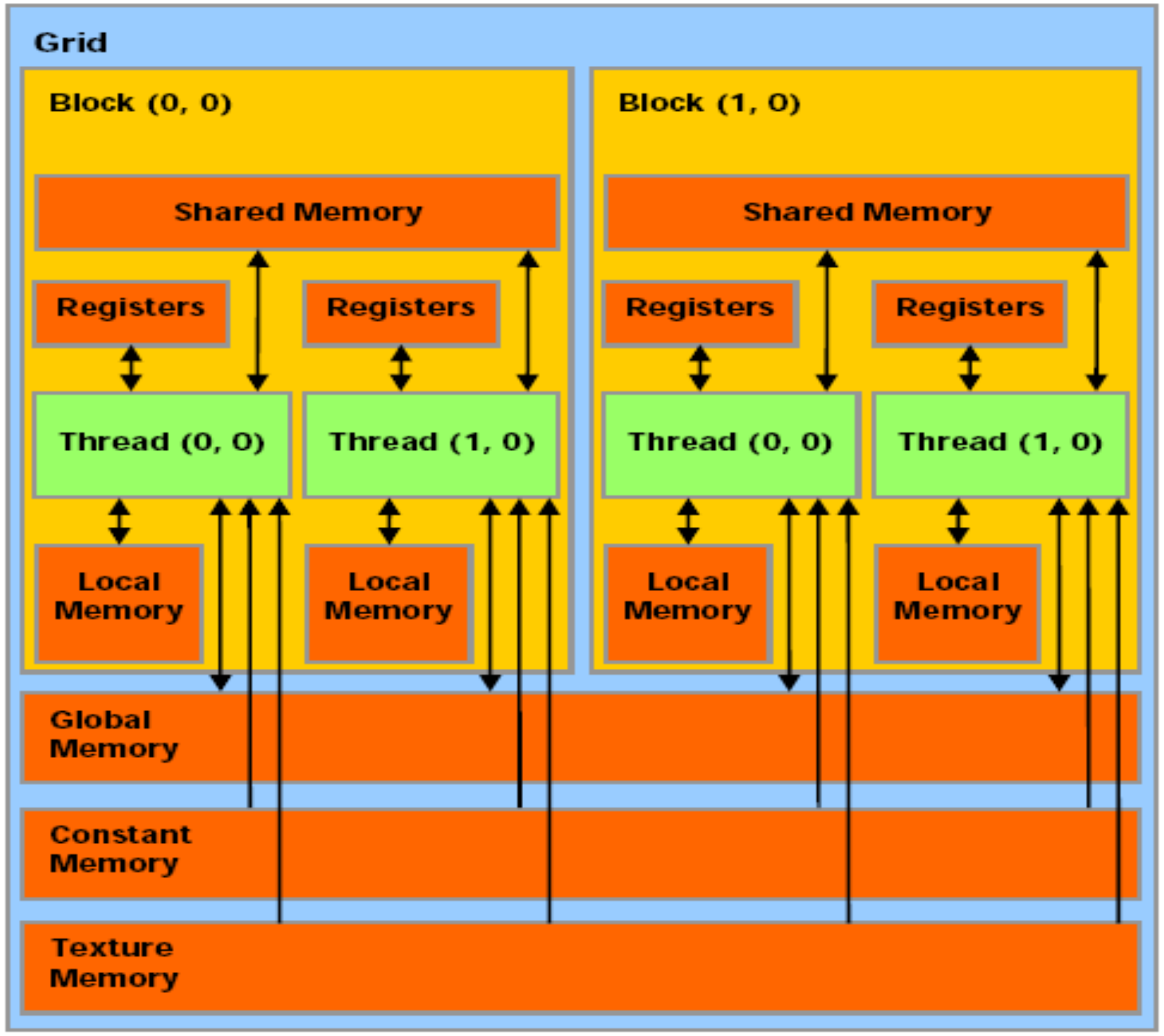


CUBLAS, CUFFT, MAGMA, ...

C like API

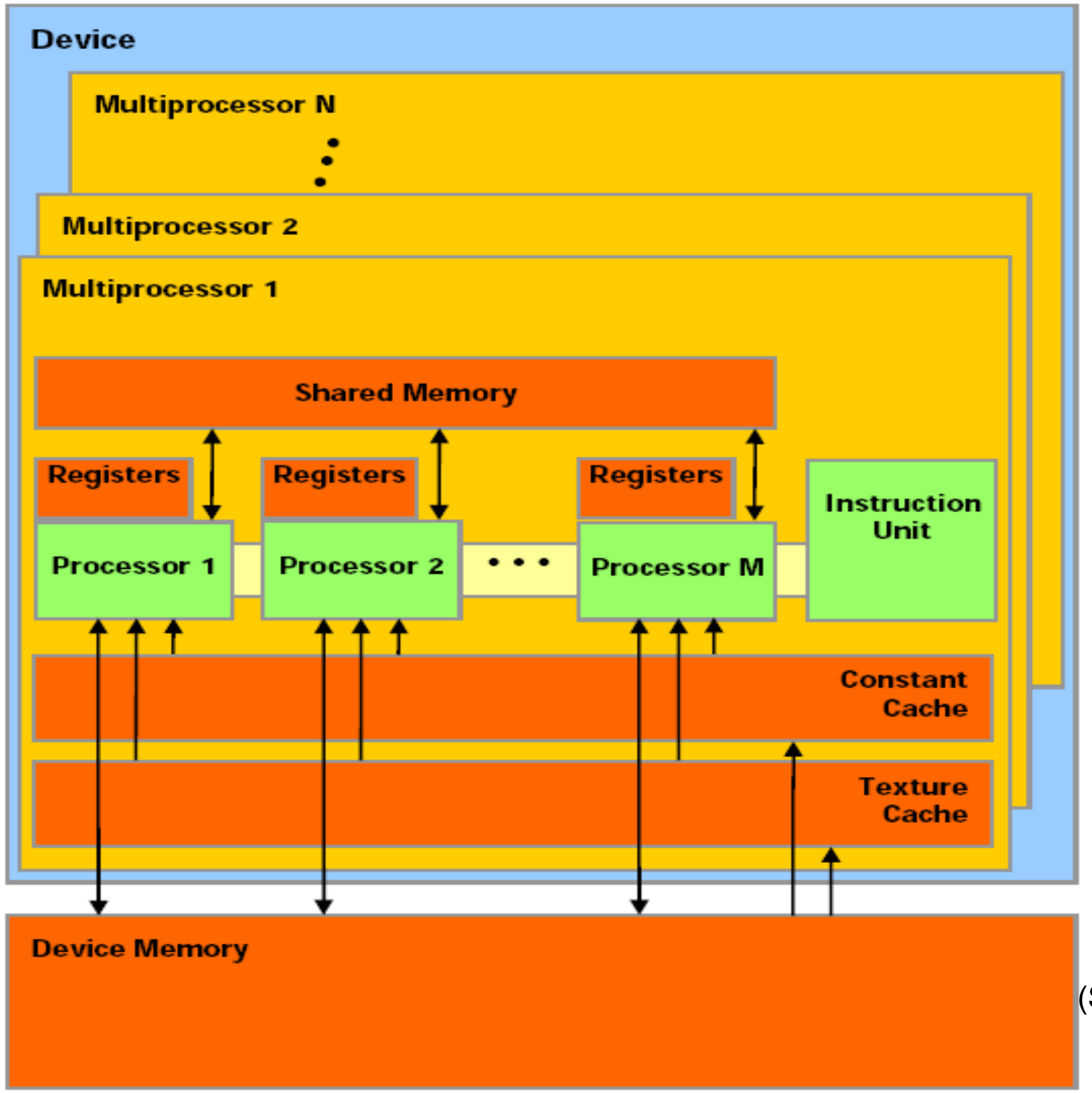
(Source: NVIDIA CUDA Programming Guide)

CUDA Memory Model



(Source: NVIDIA CUDA Programming Guide)

CUDA Hardware Model



(Source: NVIDIA CUDA Programming Guide)

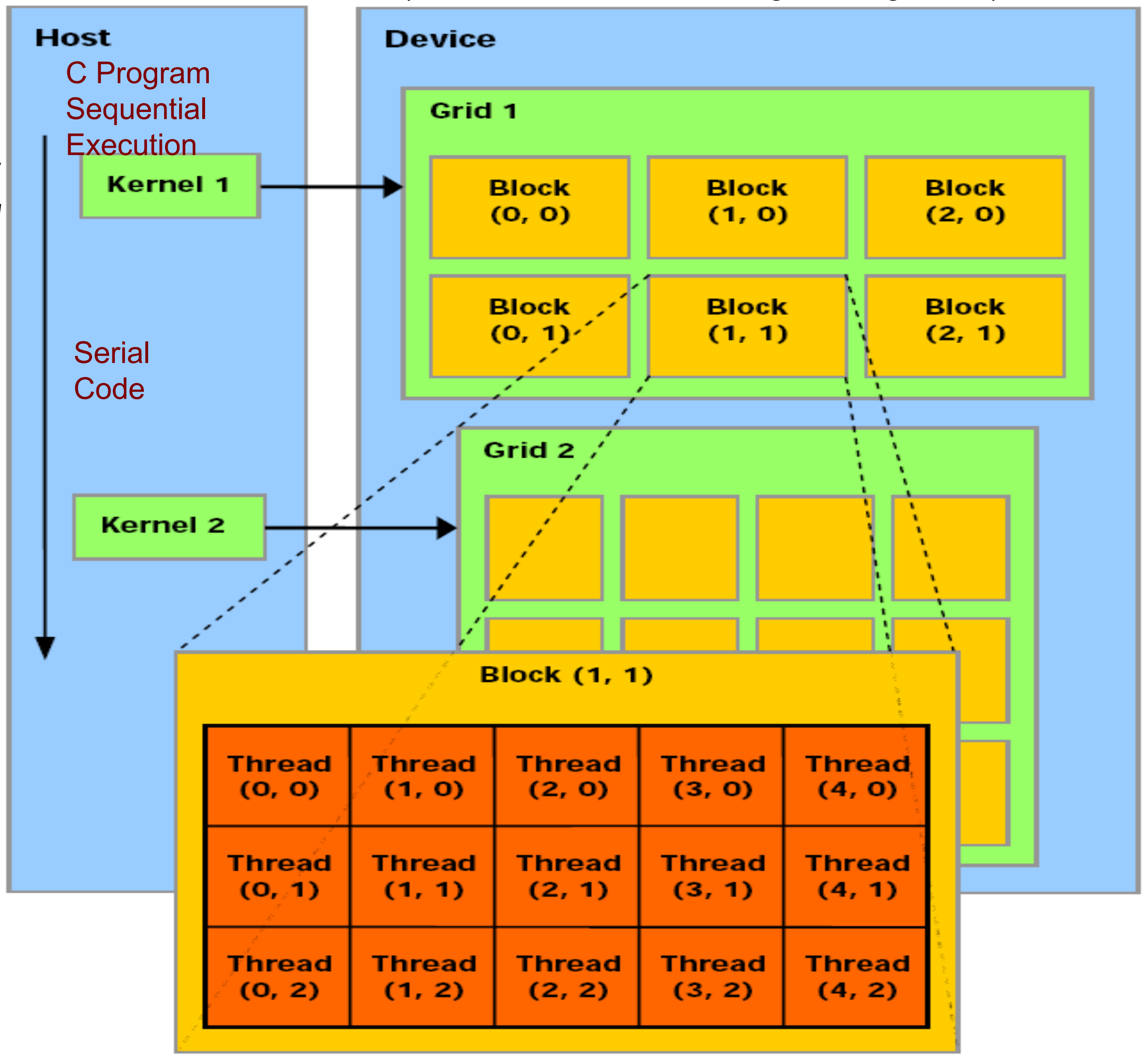
CUDA Programming Model

- **Grid of thread blocks**
  (blocks of the same dimension, grouped together to execute the same kernel)
- **Thread block**
  (a batch of threads with fast shared memory executes a kernel)
- **Sequential code launches asynchronously GPU kernels**

```
// set the grid and thread configuration
Dim3 dimGrid(2,3);
Dim3 dimTBlock(3,5);

// Launch the device computation
MatVec<<<dimGrid, dimTBlock>>>( . . . );
```

```
__global__ void MatVec( . . . ) {
  // Block index
  int bx = blockIdx.x;
  int by = blockIdx.y;

  // Thread index
  int tx = threadIdx.x;
  int ty = threadIdx.y;
  . . .
}
```

(Source: NVIDIA CUDA Programming Guide)



Hello World!

```
int main(void) {
        printf("Hello World!\n");
        return 0;
}
```

- Standard C that runs on the host
- NVIDIA compiler (nvcc) can be used to compile programs with no *device* code

Output:

```
$ nvcc
hello_world.cu
$ a.out
Hello World!
$
```

Hello World! with Device Code

Slides from NVIDIA presentation Introduction_to_CUDA_C.pptx

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

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

Hello World! with Device Code

Slides from NVIDIA presentation Introduction_to_CUDA_C.pptx

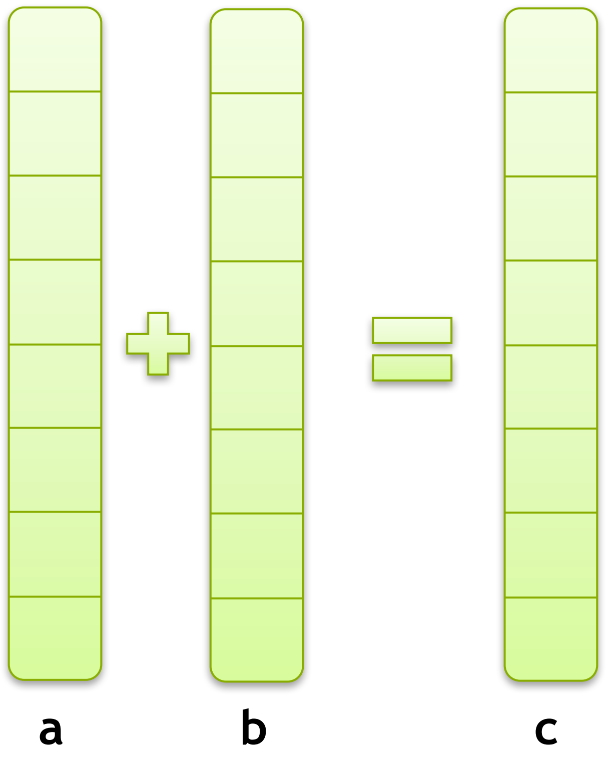
```
__global__ void mykernel(void) {
    printf("Hello World from block %d, thread %d!\n”,
            blockIdx.x, threadIdx.x);
}

int main(void) {
      mykernel<<<10,10>>>();
      cudaDeviceSynchronize();
      return 0;
}
```

Parallel Programming in CUDA C/C++

- But wait… GPU computing is about massive parallelism!

- We need a more interesting example…

- We'll start by adding two integers and build up to vector addition



Addition on the Device: `add()`

Slides from NVIDIA presentation Introduction_to_CUDA_C.pptx

- Returning to our `add()` kernel

```
__global__ void add(int *a, int *b, int *c) {
     *c = *a + *b;
}
```

- Let's take a look at main()…

Addition on the Device: `main()`

```
int main(void) {
        int a, b, c;                    // host copies of a, b, c
        int *d_a, *d_b, *d_c;           // device copies of a, b, c
        int size = sizeof(int);

        // Allocate space for device copies of a, b, c
        cudaMalloc((void **)&d_a, size);
        cudaMalloc((void **)&d_b, size);
        cudaMalloc((void **)&d_c, size);

        // Setup input values
        a = 2;
        b = 7;
```

Addition on the Device: `main()`

Slides from NVIDIA presentation Introduction_to_CUDA_C.pptx

```
        // Copy inputs to device
        cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
        cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);

        // Launch add() kernel on GPU
        add<<<1,1>>>(d_a, d_b, d_c);

        // Copy result back to host
        cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);

        // Cleanup
        cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
        return 0;
    }
```

Moving to Parallel

- GPU computing is about massive parallelism
  - So how do we run code in parallel on the device?

```
add<<< 1, 1 >>>();
          ⇩
add<<< N, 1 >>>();
```

- Instead of executing `add()` once, execute N times in parallel

Vector Addition on the Device: `add()`

- Returning to our parallelized `add()` kernel

```
__global__ void add(int *a, int *b, int *c) {
        c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

- Let's take a look at main()…

Vector Addition on the Device: `main()`

```
#define N 512
int main(void) {
    int *a   *b   *c                // host copies of a, b, c
    int *d_a, *d_b, *d_c; // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

Vector Addition on the Device: main()

Slides from NVIDIA presentation Introduction_to_CUDA_C.pptx

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU with N blocks
    add<<<N,1>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Vector Addition Using Threads: main()

Slides from NVIDIA presentation Introduction_to_CUDA_C.pptx

```
#define N 512
int main(void) {
    int *a, *b, *c;                          // host copies of a, b, c
    int *d_a, *d_b, *d_c;               // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

Vector Addition Using Threads: `main()`

Slides from NVIDIA presentation Introduction_to_CUDA_C.pptx

```
     // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU with N threads
    add<<<1,N>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Combining Blocks and Threads

- **We've seen parallel vector addition using:**
  - Many blocks with one thread each
  - One block with many threads

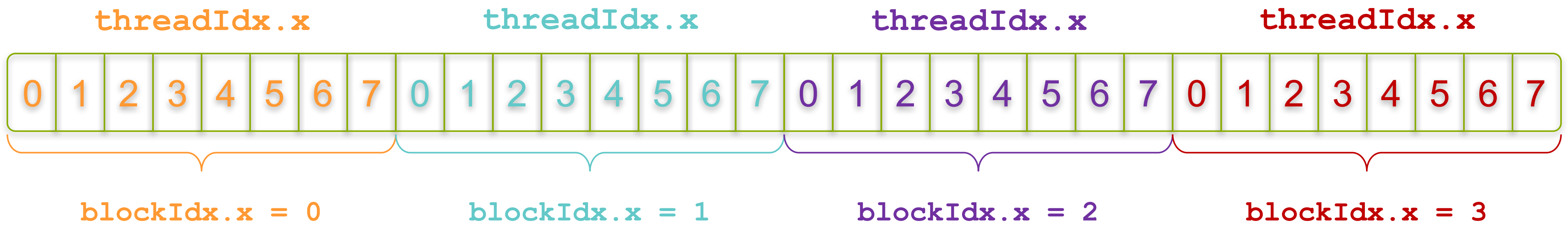
- **Let's adapt vector addition to use both blocks and threads**

- **Why? We'll come to that…**

- **First let's discuss data indexing…**

Indexing Arrays with Blocks and Threads

- No longer as simple as using `blockIdx.x` and `threadIdx.x`
  - Consider indexing an array with one element per thread (8 threads/block)

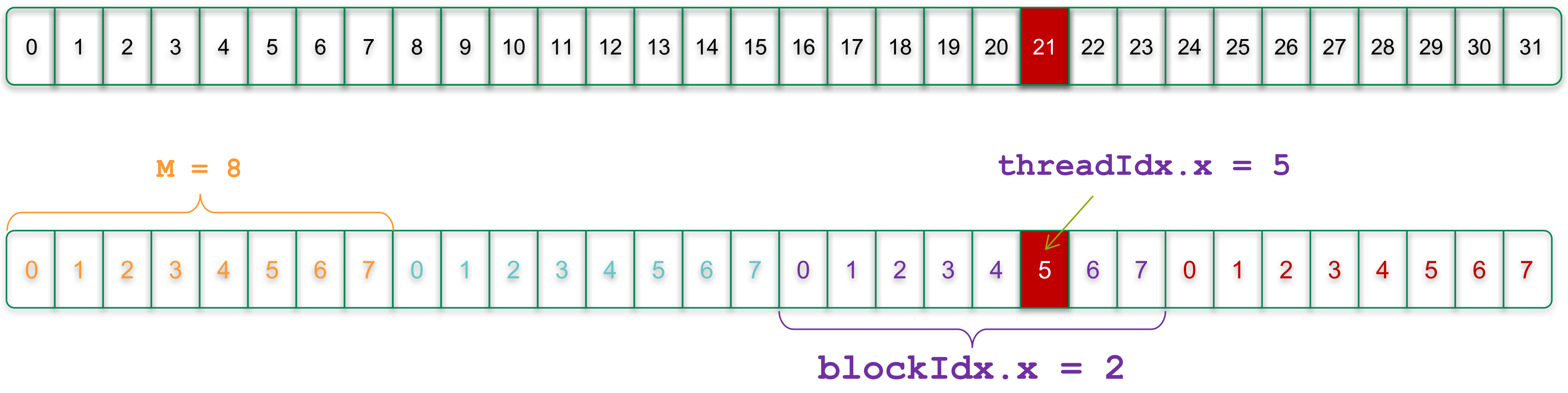


- With M threads/block a unique index for each thread is given by:

```
int index = threadIdx.x + blockIdx.x * M;
```

Indexing Arrays: Example

- **Which thread will operate on the red element?**



```
int index = threadIdx.x + blockIdx.x * M;
          =       5     +      2      * 8;
          = 21;
```

Vector Addition with Blocks and Threads

- Use the built-in variable `blockDim.x` for threads per block

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

- Combined version of `add()` to use parallel threads *and* parallel blocks

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

- What changes need to be made in `main()`?

Addition with Blocks and Threads: `main()`

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main(void) {
    int *a, *b, *c;                         // host copies of a, b, c
    int *d_a, *d_b, *d_c;              // device copies of a, b, c
    int size = N * sizeof(int);

    // Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    // Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size);
```

Addition with Blocks and Threads: `main()`

```
    // Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<N/THREADS_PER_BLOCK THREADS_PER_BLOCK>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Handling Arbitrary Vector Sizes

- Typical problems are not friendly multiples of `blockDim.x`

- Avoid accessing beyond the end of the arrays:

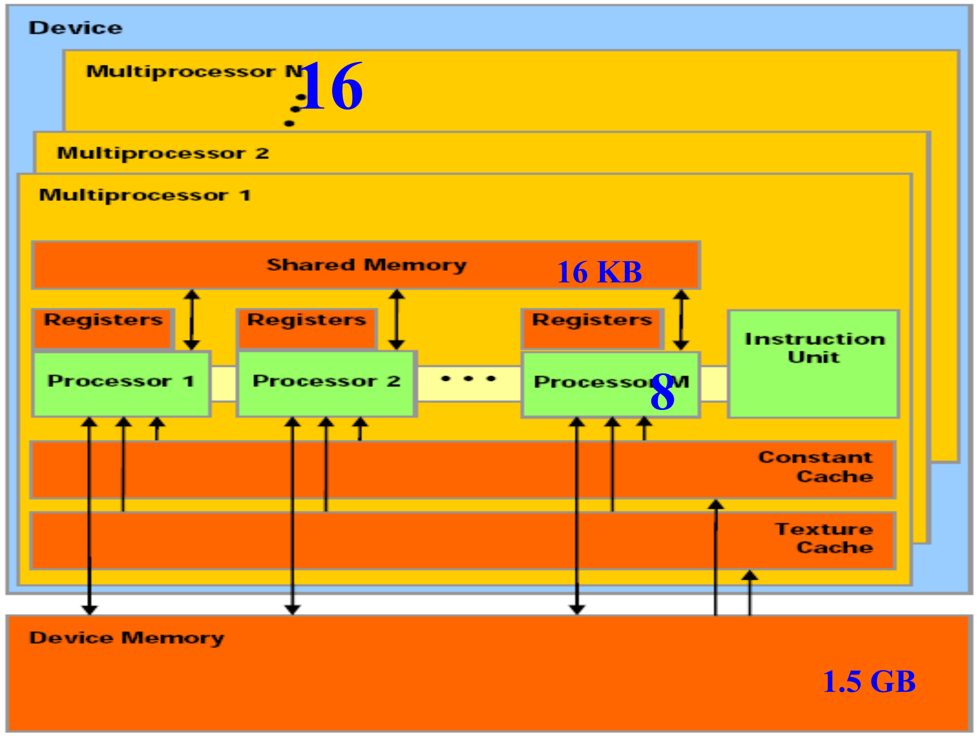
```
__global__ void add(int *a, int *b, int *c, int n) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    if (index < n)
        c[index] = a[index] + b[index];
}
```

- Update the kernel launch:

```
add<<<(N + M-1) / M,M>>>(d_a, d_b, d_c, N);
```

GPUs & Challenges

- Programming is 'easier' with NVIDIA's Compute Unified Device Architecture (CUDA)



Quadro FX 5600

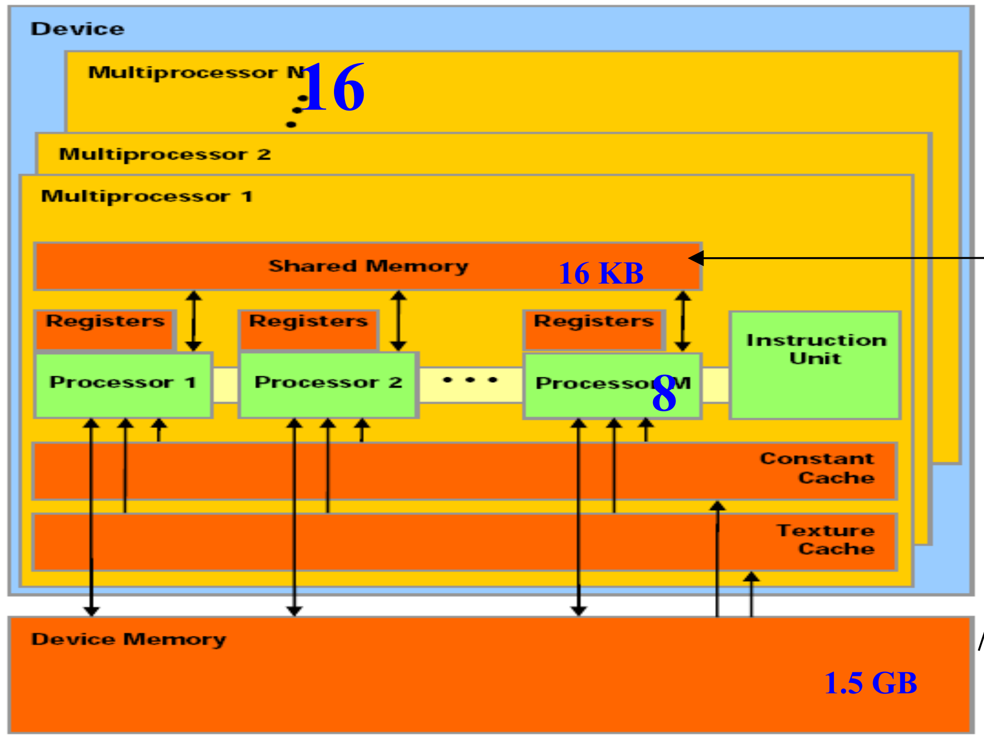
Installed (at ICL) on a 4 x Dual Core AMD Opteron Processor 265 (1800 MHz, 1024 KB cache)

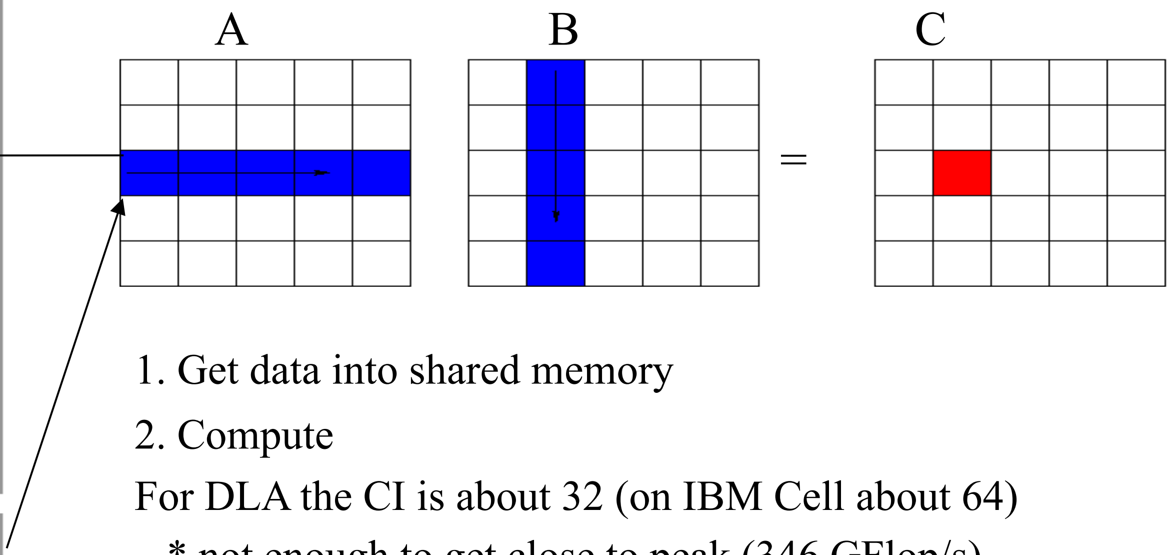
Some numbers:

processors: 128 (total)	max performance: 346 GFlop/s
registers : 8192 / block	memory bandwidth: 76.8 GB/s
warp size : 32	bandwidth to CPU: 8 GB/s
max threads / block: 512	shared memory: 16 KB among 8 processors on a multiproc.

GPUs & Challenges

- Programming is 'easier' with NVIDIA's Compute Unified Device Architecture (CUDA)





1. Get data into shared memory
2. Compute

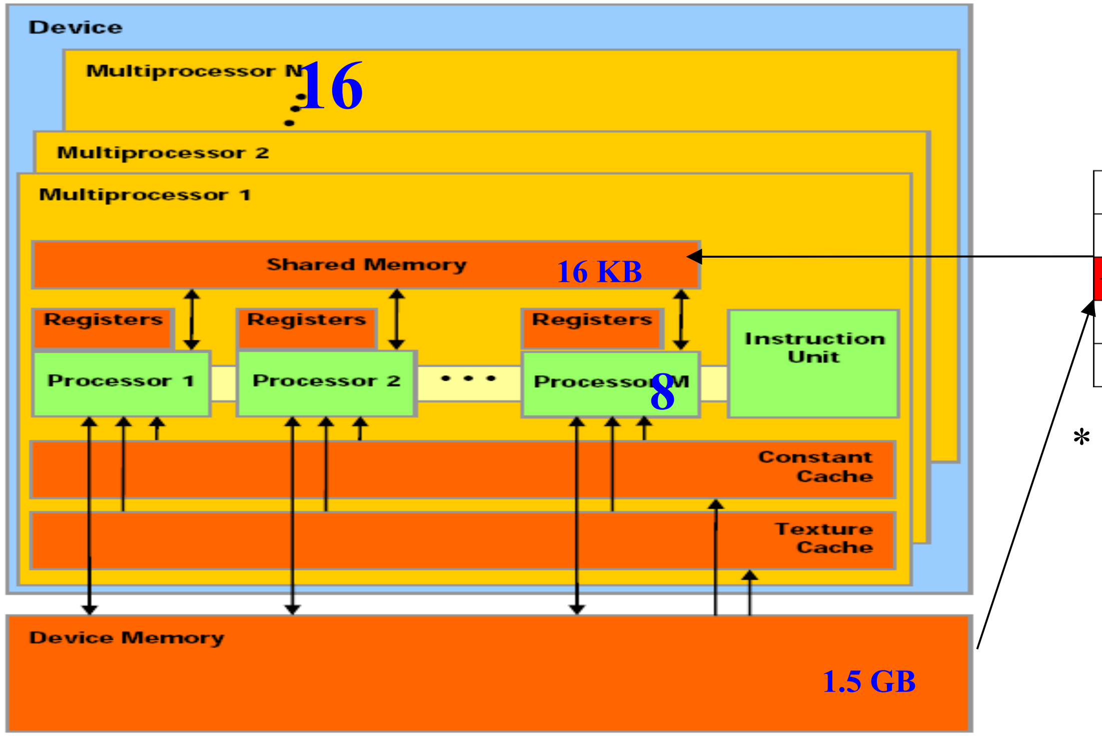
For DLA the CI is about 32 (on IBM Cell about 64)

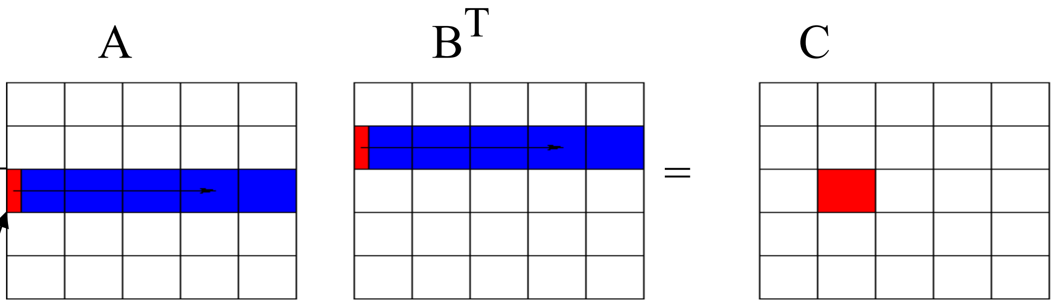
* not enough to get close to peak (346 GFlop/s)

* CUBLAS sgemm is about 120 Gflop/s

GPUs & Challenges

- Programming is 'easier' with NVIDIA's Compute Unified Device Architecture (CUDA)





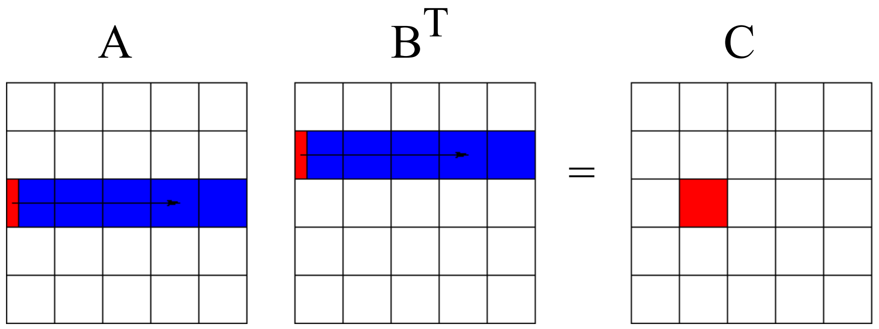
* **Small** red rectangles (to overlap communication & computation) are of size 32 x 4 and are red by 32 x 2 threads

2008. Volkov and Demmel. ***Benchmarking GPUs to tune dense linear algebra***, SC08
http://mc.stanford.edu/cgi-bin/images/6/65/SC08_Volkov_GPU.pdf

2010. Nath, Tomov, and Dongarra. ***An Improved MAGMA GEMM for Fermi Graphics Processing Units***, IJHPCA
http://www.netlib.org/lapack/lawnspdf/lawn227.pdf

SGEMM Example

2008. Volkov and Demmel. ***Benchmarking GPUs to tune dense linear algebra***, SC08
http://mc.stanford.edu/cgi-bin/images/6/65/SC08_Volkov_GPU.pdf



* **Small** red rectangles (to overlap communication & computation) are of size 32 x 4 and are red by 32 x 2 threads

```
// GPU kernel: compute C = alpha A B' + beta C
__global__ void sgemmNT( const float *A, int lda,
                         const float *B, int ldb,
                         float* C, int ldc, int k,
                         float alpha, float beta  )
{
    int inx = threadIdx.x;
    int iny = threadIdx.y;
    int ibx = blockIdx.x * 32;
    int iby = blockIdx.y * 32;

    A += ibx + inx + __mul24( iny, lda );
    B += iby + inx + __mul24( iny, ldb );
    C += ibx + inx + __mul24( iby + iny, ldc );

    float c[16] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
    for( int i = 0; i < k; i += 4 )
    {
        __syncthreads();
        __shared__ float a[4][32];
        __shared__ float b[4][32];

        a[iny][inx] = A[i*lda];
        a[iny+2][inx] = A[(i+2)*lda];

        b[iny][inx]   = B[i*ldb];
        b[iny+2][inx] = B[(i+2)*ldb];
        __syncthreads();

        for( int j = 0; j < 4; j++ )
        {
            float _a = a[j][inx];
            float *_b = &b[j][0] + iny;

            c[0] += _a*_b[0];
            c[1] += _a*_b[2];
            c[2] += _a*_b[4];
            c[3] += _a*_b[6];
            c[4] += _a*_b[8];
            c[5] += _a*_b[10];
            c[6] += _a*_b[12];
            c[7] += _a*_b[14];
            c[8] += _a*_b[16];
            c[9] += _a*_b[18];
            c[10] += _a*_b[20];
            c[11] += _a*_b[22];
            c[12] += _a*_b[24];
            c[13] += _a*_b[26];
            c[14] += _a*_b[28];
            c[15] += _a*_b[30];
        }
    }

    for( int i = 0; i < 16; i++, C += 2*ldc )
        C[0] = alpha * c[i] + beta * C[0];
}


void ourSgemm (char transa, char transb,
               int m, int n, int k,
               float alpha,
               const float *A, int lda,
               const float *B, int ldb,
               float beta,
               float *C, int ldc)
{
    assert( (transa == 'N' || transa == 'n') &&
            (transb == 'T' || transb == 't') &&
            ((m|n|k|lda|ldb)&31) == 0,
            "unsupported parameters in ourSgemm()" );

    dim3 grid( m/32, n/32, 1 );
    dim3 threads2( 32, 2, 1 );
    sgemmNT<<<grid, threads2>>>( A, lda,
                                 B, ldb,
                                 C, ldc,
                                 k, alpha, beta );
}
```

Exercises

- Go on the xsede system

```
> ssh login.xsede.org -l userid
> gsissh bridges
> interact –gpu
> module load cuda
```

- Try the hello_world.cu
- Try codes in samples, e.g.,

```
> cp -R /opt/packages/cuda/8.0/samples/ .
> cd samples/0_Simple/vectorAdd
> export CUDA_PATH=/opt/packages/cuda/8.0
> make
> ./vectorAdd
```

- Look at the codes, how are they compiled, run, maybe try to change them …

MAGMA Tutorial

http://icl.utk.edu/projectsfiles/magma/tutorial/ecp2018-magma-tutorial.pdf

- **Install MAGMA on bridges**

```
> ssh login.xsede.org -l userid
> gsissh bridges
> module load cuda intel/17.4
> wget http://icl.utk.edu/projectsfiles/magma/downloads/magma-2.3.0.tar.gz
> tar zxvf magma-2.3.0.tar.gz
> cd magma-2.3.0
> cp make.inc-examples/make.inc.mkl-gcc make.inc
> export CUDADIR=/opt/packages/cuda/8.0/
> make lib –j
> cd testing
> make testing_dgemm

> gsissh bridges
> interact –gpu
> module load cuda
> ./testing_dgemm –l -c
```

Machine Learning

- Coursera
  https://www.coursera.org/
  - Sign up (give name, email, and password) and sign in
- Search “machine learning”
  - Choose "Machine learning Stanford University”
  - Go through the material for Weeks 1, 2, 3, 4, and 5 (just listen the videos)
  - For weeks 4 & 5 do assignments
- Search “deep learning” and select
  “Convolutional Neural Networks by deeplearning.ai”
- MagmaDNN
  http://icl.cs.utk.edu/projectsfiles/magma/pubs/71-MagmaDNN.pdf

Collaborators and Support

MAGMA team

http://icl.cs.utk.edu/magma

PLASMA team

http://icl.cs.utk.edu/plasma

Collaborating partners

University of Tennessee, Knoxville
Lawrence Livermore National Laboratory
University of California, Berkeley
University of Colorado, Denver
INRIA, France (StarPU team)
KAUST, Saudi Arabia