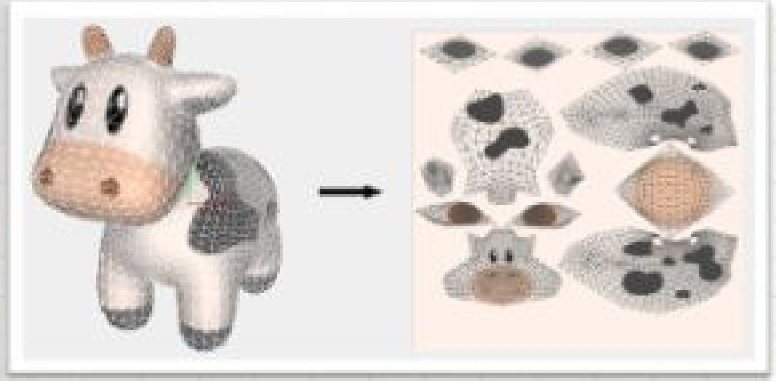
Graphical Pipeline:

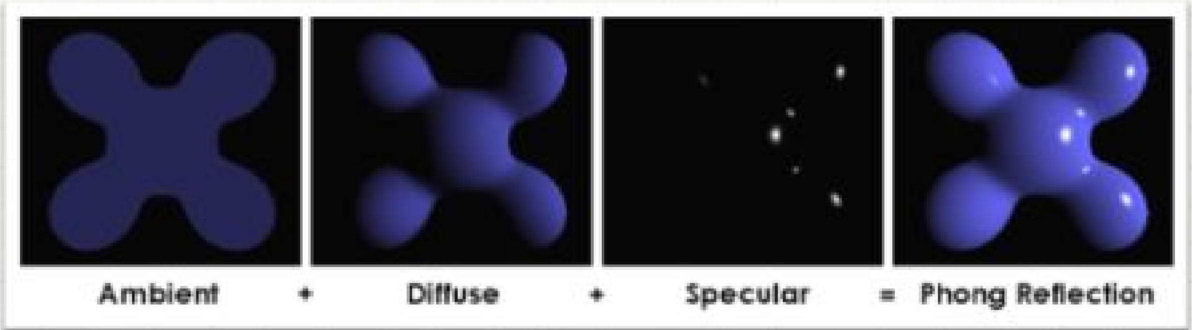
- Set of steps need to be taken in order to turn 3D scene into 2D image
  - The graphical pipeline is conceptual model
  - Highly dependent on the underlying available Software and Hardware accelerators
- The model of graphical pipeline is usually used in real-time rendering
  - Each step is backed with efficient algorithms that are usually hardware accelerated (e.g., using GPUs)
- How do you represent 3D surfaces ?
- A 3D mesh (usually triangles):
  - Simple objects can be composed of thousands of triangles!

Graphic Processing – Some History

- 1990s: Real-time 3D rendering for video games were becoming common
  - Doom, Quake, Descent, … (Nostalgia!)
- 3D graphics processing is immensely computation-intensive



Texture mapping

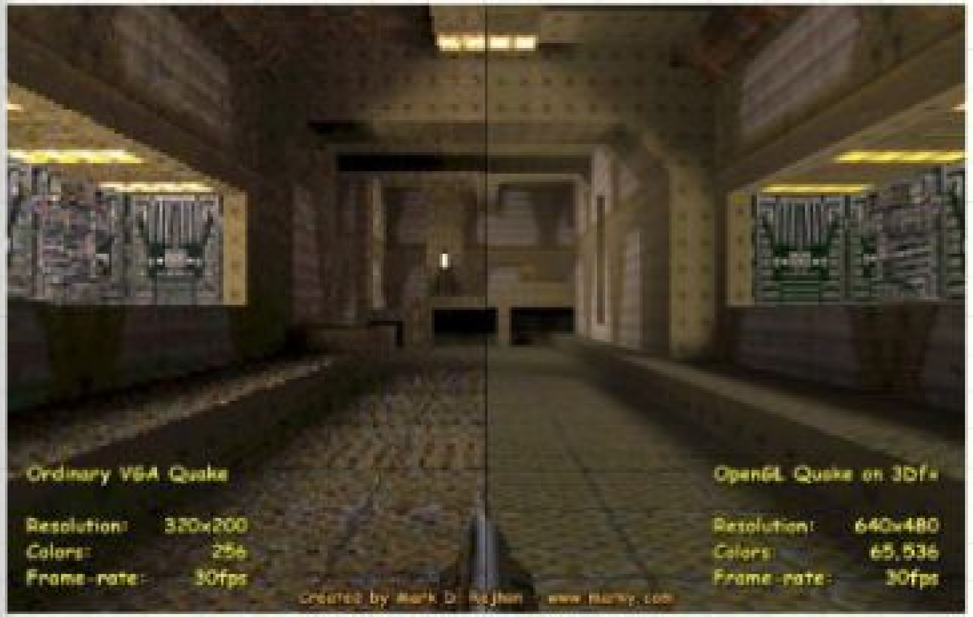


Shading

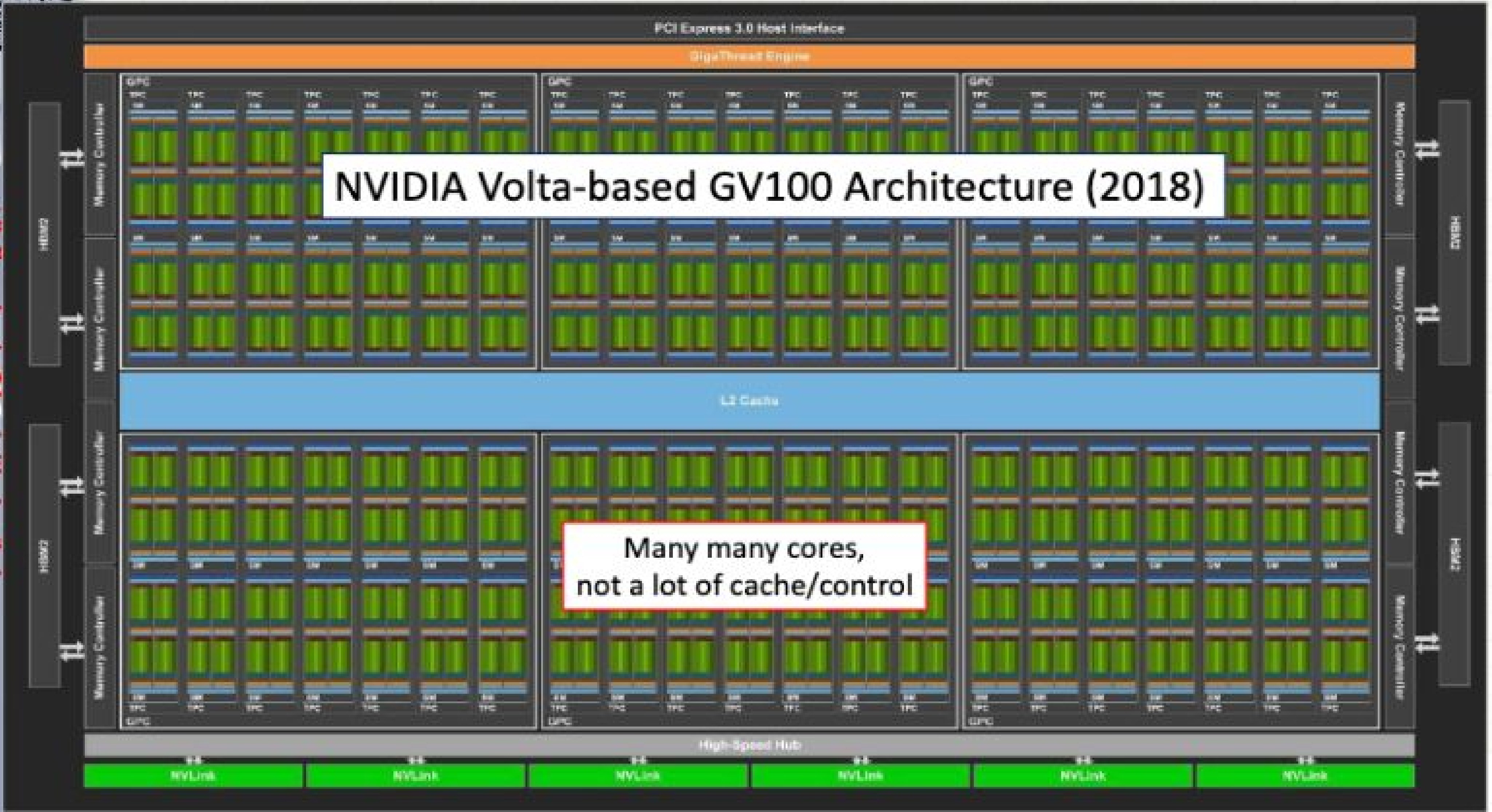
Warren Moore, "Textures and Samplers in Metal," Metal by Example, 2014

Introduction of 3D Accelerator Cards

- ❑ Much of 3D processing is short algorithms repeated on a lot of data
  - o pixels, polygons, textures, …
- ❑ Dedicated accelerators with simple, massively parallel computation



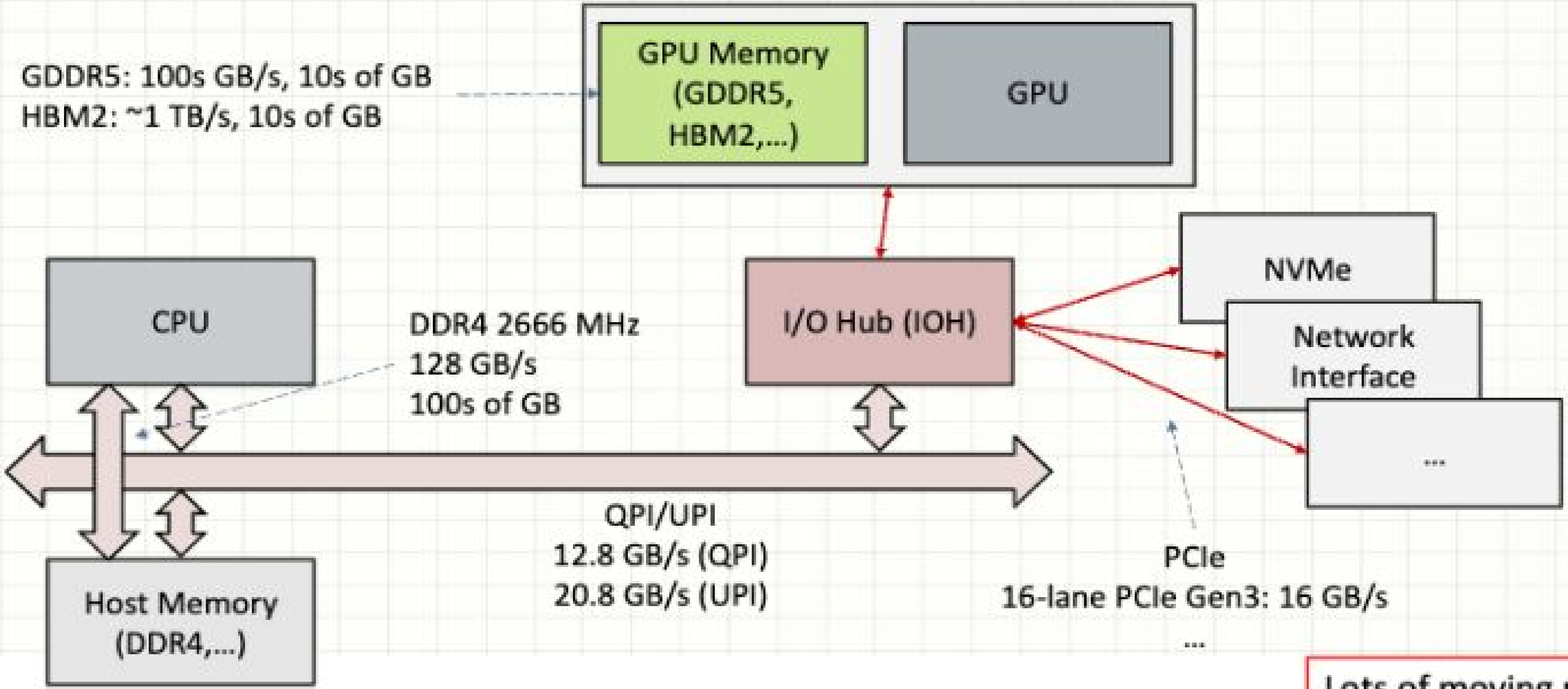
A Diamond Monster 3D, using the Voodoo chipset (1997)
(Konstantin Lanzet, Wikipedia)



NVIDIA Volta-based GV100 Architecture (2018)

Many many cores,
not a lot of cache/control

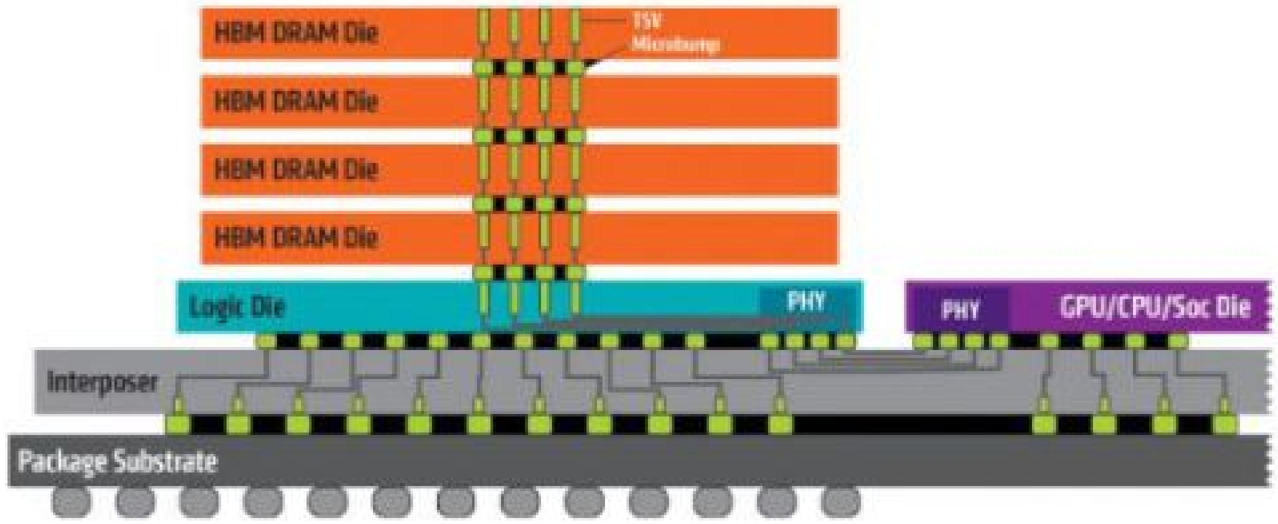
System Architecture Snapshot With a GPU



Lots of moving parts!

High-Performance Graphics Memory

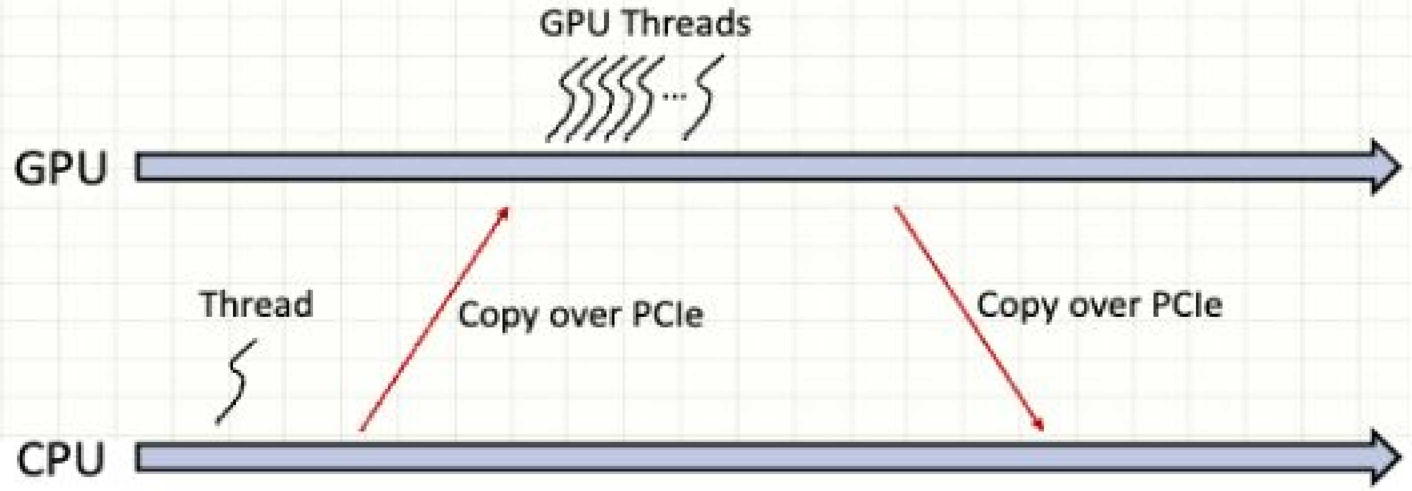
- ❑ Modern GPUs even employing 3D-stacked memory via silicon interposer
  - o Very wide bus, very high bandwidth
  - o e.g., HBM2 in Volta



Graphics Card Hub, "GDDR5 vs GDDR5X vs HBM vs HBM2 vs GDDR6 Memory Comparison," 2019

Massively Parallel Architecture For Massively Parallel Workloads!

- NVIDIA CUDA (Compute Uniform Device Architecture) – 2007
  - A way to run custom programs on the massively parallel architecture!
- OpenCL specification released – 2008
- Both platforms expose synchronous execution of a massive number of threads



NVIDIA GPU Architecture

- A scalable array of multithreaded Streaming Multiprocessors (SMs), each SM consists of
  - 8 Scalar Processor (SP) cores
  - 2 special function units for transcendentals
  - A multithreaded instruction unit
  - On-chip shared memory
- GDDR3 SDRAM
- PCIe interface

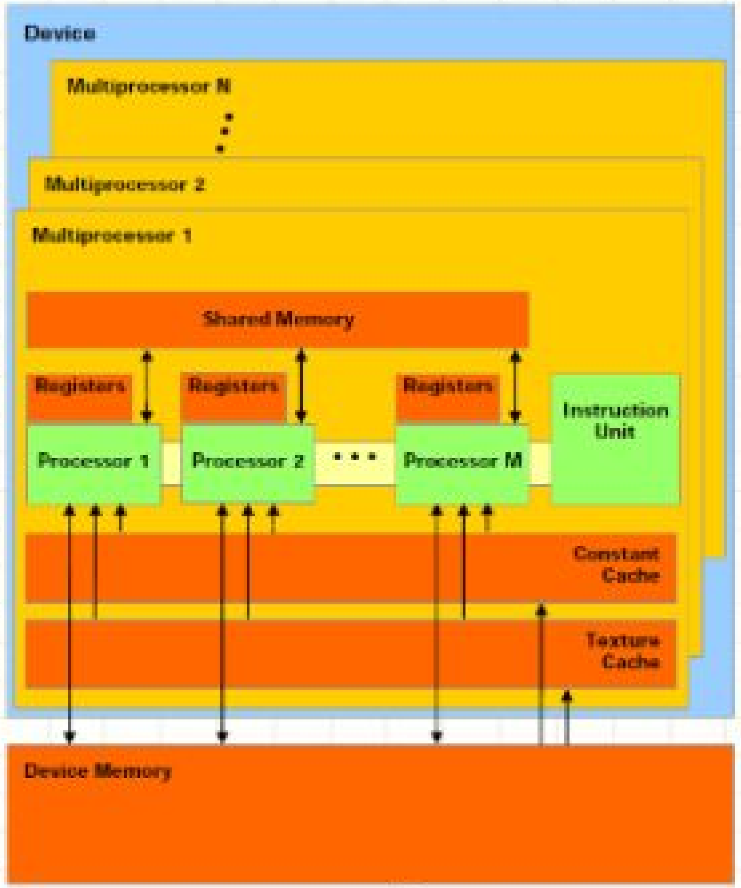
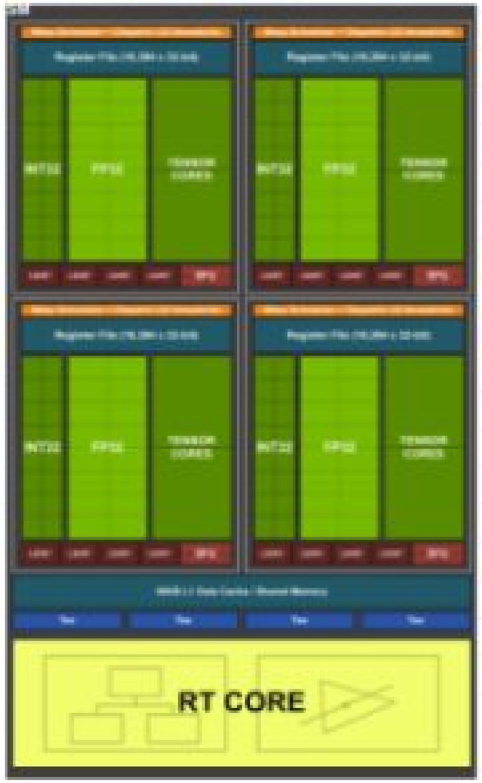


Figure is courtesy of NVIDIA

Volta Execution Architecture

- 64 INT32 Cores, 64 FP32 Cores, 4 Tensor Cores, Ray-tracing cores..
  - Specialization to make use of chip space…?
- Not much on-chip memory per thread
  - 96 KB Shared memory
  - 1024 Registers per FP32 core
- Hard limit on compute management
  - 32 blocks AND 2048 threads AND 1024 threads/block
  - e.g., 2 blocks with 1024 threads, or 4 blocks with 512 threads
  - Enough registers/shared memory for all threads must be available (all context is resident during execution)



More threads than cores – Threads interleaved to hide memory latency

Resource Balancing Details

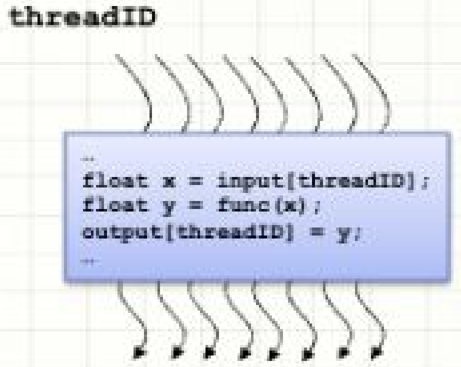
- How many threads in a block?
- Too small: 4x4 window == 16 threads
  - 128 blocks to fill 2048 thread/SM
  - SM only supports 32 blocks -> only 512 threads used
    - SM has only 64 cores… does it matter? Sometimes!
- Too large: 32x48 window == 1536 threads
  - Threads do not fit in a block!
- Too large: 1024 threads using more than 64 registers
- Limitations vary across platforms (Fermi, Pascal, Volta, …)

Warp Scheduling Unit

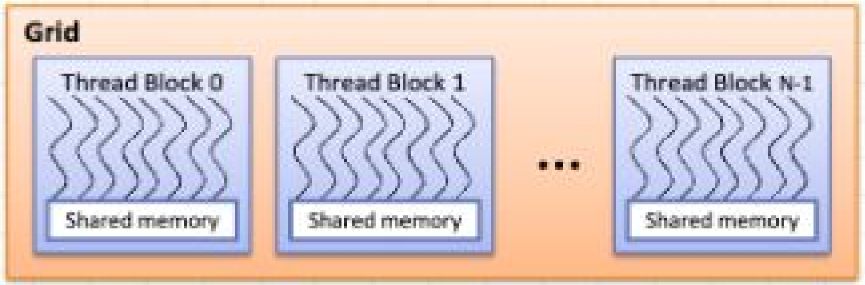
- Threads in a block are executed in 32-thread "warp" unit
  - Not part of language specs, just architecture specifics
  - A warp is SIMD – Same PC, same instructions executed on every core
- What happens when there is a conditional statement?
  - Prefix operations, or control divergence
  - More on this later!
- Warps have been 32-threads so far, but may change in the future

CUDA Programming Model

- A CUDA kernel is executed by an array of threads
  - All threads run the same code (SPMD)
  - Each thread has an ID that it uses to compute memory addresses and make control decisions
- Threads are arranged as a grid of thread blocks
  - Threads within a block have access to a segment of shared memory



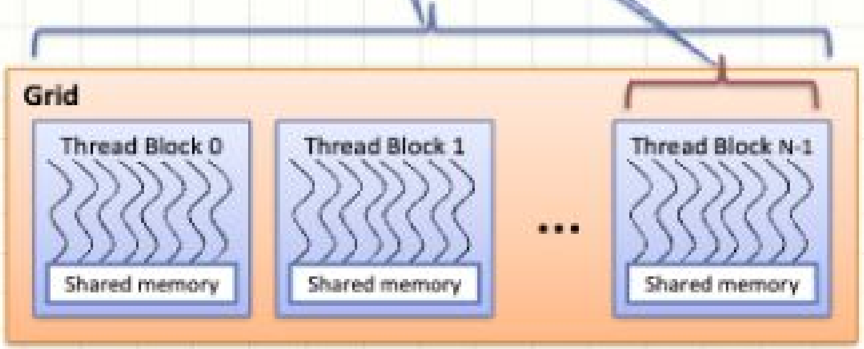
```
…
float x = input[threadID];
float y = func(x);
output[threadID] = y;
…
```



Kernel Invocation Syntax

grid & thread block dimensionality

```
vecAdd<<<32, 512>>>(devPtrA, devPtrB, devPtrC);
```



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

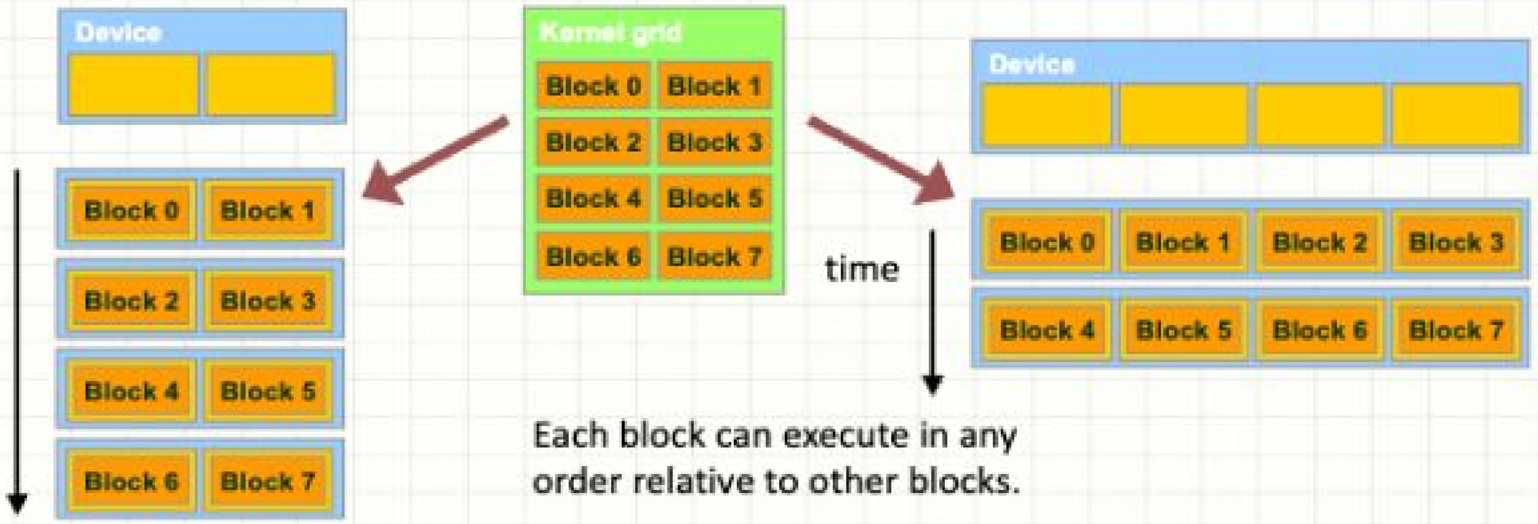
block ID within a grid

number of theards per block

thread ID within a thread block

Mapping Threads to the Hardware

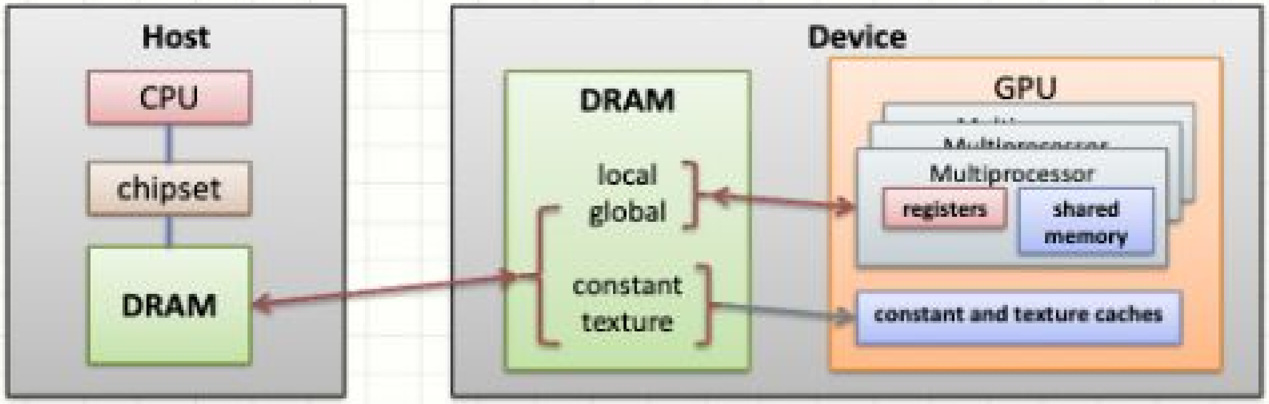
- Blocks of threads are transparently assigned to SMs
  - A block of threads executes on one SM & does not migrate
  - Several blocks can reside concurrently on one SM



Each block can execute in any order relative to other blocks.

Slide is courtesy of NVIDIA

GPU Memory Hierarchy



Memory	Location	Cached	Access	Scope	Lifetime
Register	On-chip	N/A	R/W	One thread	Thread
Local	Off-chip	No	R/W	One thread	Thread
Shared	On-chip	N/A	R/W	All threads in a block	Block
Global	Off-chip	No	R/W	All threads + host	Application
Constant	Off-chip	Yes	R	All threads + host	Application
Texture	Off-chip	Yes	R	All threads + host	Application

Simple CUDA Example

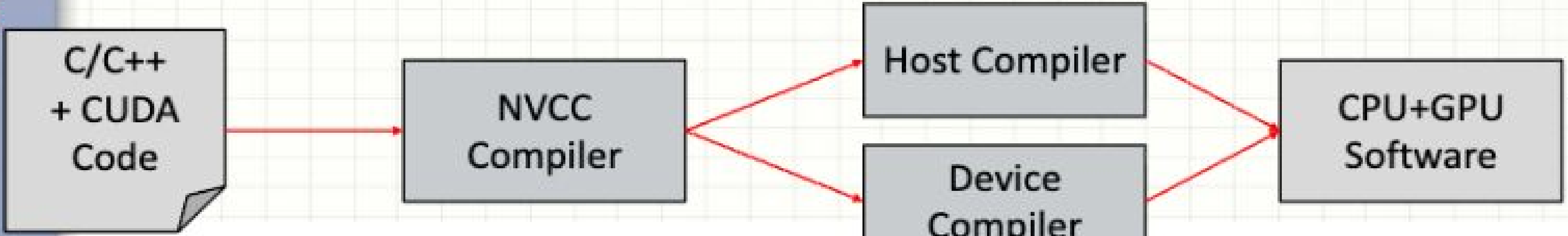
Asynchronous call

CPU side

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

GPU side

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



Simple CUDA Example

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

}
```

1 block

N threads per block

Should wait for kernel to finish

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

Only void allowed

Which of N threads am I?
See also: blockIdx

N instances of VecAdd spawned in GPU

__global__:
In GPU, called from host/GPU

__device__:
In GPU, called from GPU

__host__:
In host, called from host

One function can be both

CPU-Only Version

```
void vecAdd(int N, float* A, float* B, float* C) {
   for (int i = 0; i < N; i++)  C[i] = A[i] + B[i];
}
```

Computational kernel

```
int  main(int argc, char **argv)
{
   int N = 16384;  // default vector size

   float *A = (float*)malloc(N * sizeof(float));
   float *B = (float*)malloc(N * sizeof(float));
   float *C = (float*)malloc(N * sizeof(float));
```

Memory allocation

```
   vecAdd(N, A, B, C);  // call compute kernel
```

Kernel invocation

```
   free(A); free(B); free(C);
}
```

Memory de-allocation

Adding GPU support

```
int  main(int argc, char **argv)
{
   int  N = 16384;  // default vector size

   float *A = (float*)malloc(N * sizeof(float));
   float *B = (float*)malloc(N * sizeof(float));
   float *C = (float*)malloc(N * sizeof(float));

   float *devPtrA, *devPtrB, *devPtrC;

   cudaMalloc((void**)&devPtrA, N * sizeof(float));
   cudaMalloc((void**)&devPtrB, N * sizeof(float));
   cudaMalloc((void**)&devPtrC, N * sizeof(float));

   cudaMemcpy(devPtrA, A, N * sizeof(float),  cudaMemcpyHostToDevice);
   cudaMemcpy(devPtrB, B, N * sizeof(float),  cudaMemcpyHostToDevice);
```

Memory allocation on the GPU card

Copy data from the CPU (host) memory to the GPU (device) memory

Adding GPU support

```
vecAdd<<<N/512, 512>>>(devPtrA, devPtrB, devPtrC);
```

Kernel invocation

```
cudaMemcpy(C, devPtrC, N * sizeof(float),  cudaMemcpyDeviceToHost);
```

Copy results from device memory to the host memory

```
cudaFree(devPtrA);
cudaFree(devPtrB);
cudaFree(devPtrC);

free(A); free(B); free(C);
}
```

Device memory de-allocation

GPU Kernel

- **CPU version**

```
void vecAdd(int N, float* A, float* B, float* C)
{
   for (int i = 0; i < N; i++)
      C[i] = A[i] + B[i];
}
```

- **GPU version**

```
__global__  void vecAdd(float* A, float* B, float* C)
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   C[i] = A[i] + B[i];
}
```

Indian Institute of Technology Delhi

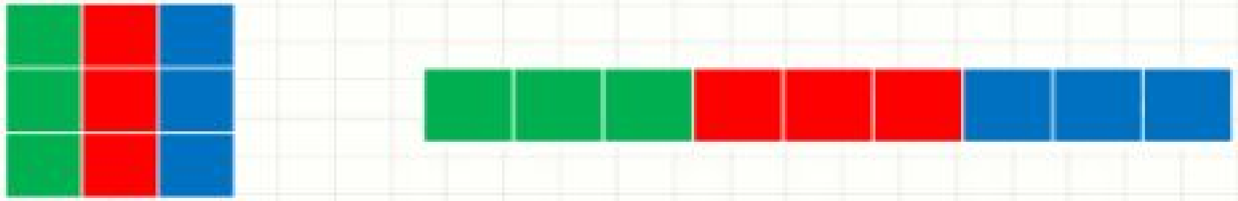
```c
int main(int argc, char* argv[])
{
   int N = 1024;

   struct timeval t1, t2, ta, tb;
   long msec1, msec2;
   float flop, mflop, gflop;

   float *a = (float *)malloc(N*N*sizeof(float));
   float *b = (float *)malloc(N*N*sizeof(float));
   float *c = (float *)malloc(N*N*sizeof(float));

   minit(a, b, c, N);

   gettimeofday(&t1, NULL);
   mmult(a, b, c, N);  // a = b * c
   gettimeofday(&t2, NULL);

   mprint(a, N, 5);

   free(a);
   free(b);
   free(c);

   msec1 = t1.tv_sec * 1000000 + t1.tv_usec;
   msec2 = t2.tv_sec * 1000000 + t2.tv_usec;
   msec2 -= msec1;
   flop = N*N*N*2.0f;
   mflop = flop / msec2;
   gflop = mflop / 1000.0f;
   printf("msec = %10ld   GFLOPS = %.3f\n", msec2, gflop);
}
```

```c
// a = b * c
void mmult(float *a, float *b, float *c, int N)
{
   for (int j = 0; j < N; j++)
      for (int k = 0; k < N; k++)
         for (int i = 0; i < N; i++)
            a[i+j*N] += b[i+k*N]*c[k+j*N];
}


void minit(float *a, float *b, float *c, int N)
{
   for (int j = 0; j < N; j++)
      for (int i = 0; i < N; i++) {
         a[i+N*j] = 0.0f;
         b[i+N*j] = 1.0f;
         c[i+N*j] = 1.0f;
      }
}

void mprint(float *a, int N, int M)
{
   int i, j;

   for (int j = 0; j < M; j++)
   {
      for (int i = 0; i < M; i++)
         printf("%.2f ", a[i+N*j]);
      printf("...\n");
   }
   printf("...\n");
}
```

Matrix Representation in Memory

```
for (i = 0; i < n; ++i)
    for (j = 0; j < n; ++j)
        for (k = 0; k < n; ++k)
            a[i+n*j] += b[i+n*k] * c[k+n*j];
```

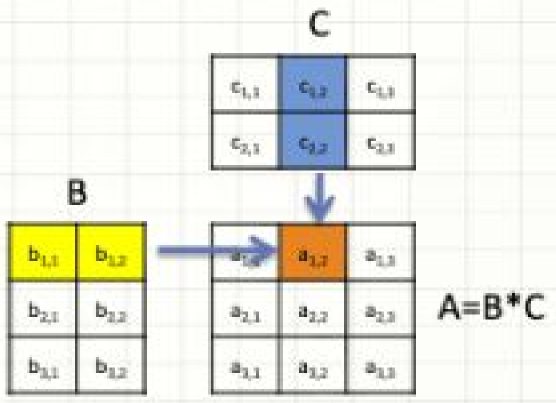
- Matrices are stored in column-major order



- For reference, jki-ordered version runs at 1.7 GFLOPS on 3 GHz Intel Xeon (single core)

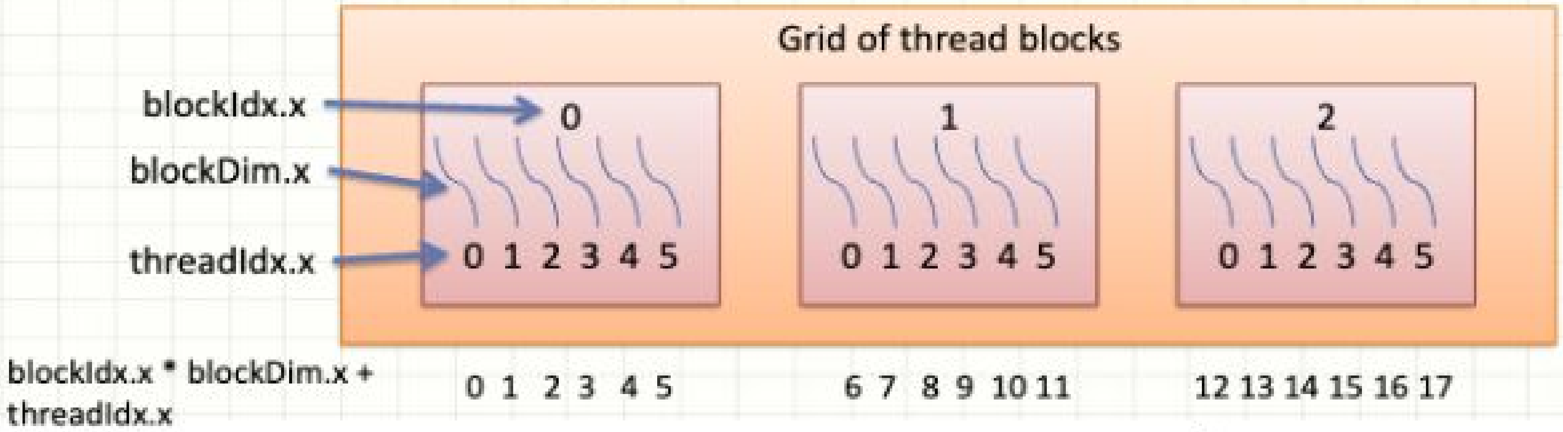
Map this code:

```
for (i = 0; i < n; ++i)
   for (j = 0; j < n; ++j)
      for (k = 0; k < n; ++k)
         a[i+n*j] += b[i+n*k] * c[k+n*j];
```

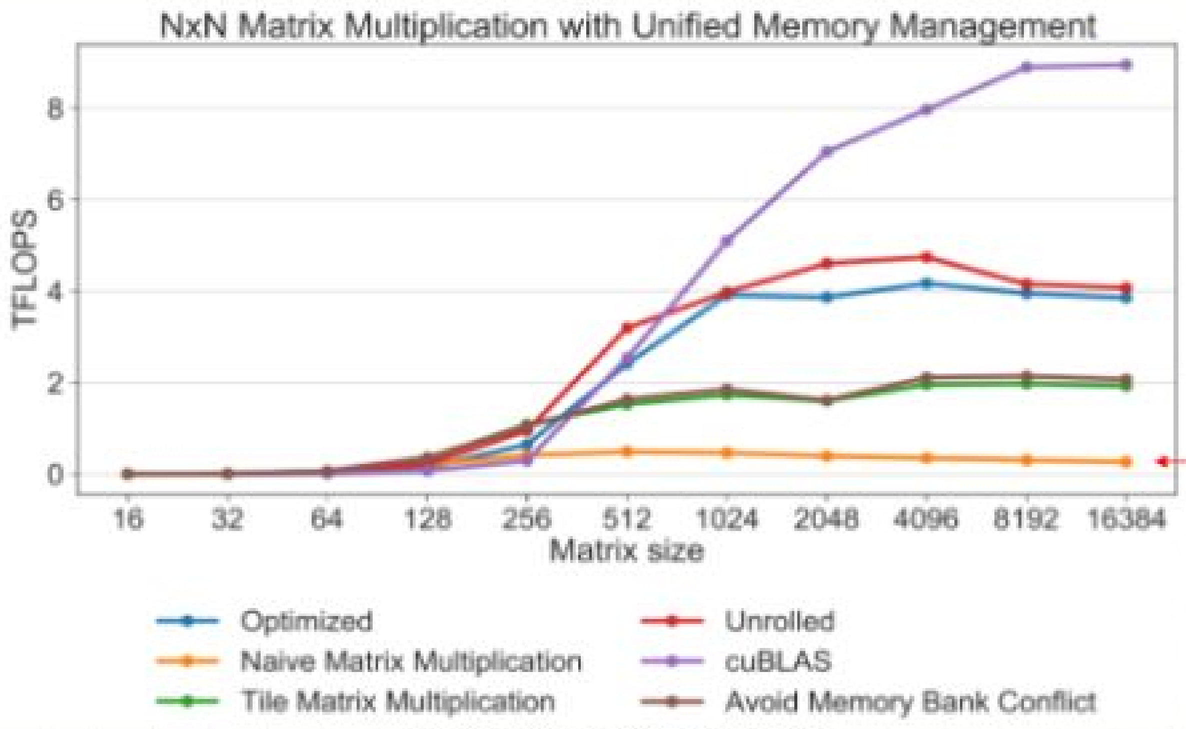

C
$c_{1,1}$ $c_{1,2}$ $c_{1,3}$
$c_{2,1}$ $c_{2,2}$ $c_{2,3}$
B
$b_{1,1}$ $b_{1,2}$
$b_{2,1}$ $b_{2,2}$
$b_{3,1}$ $b_{3,2}$
$a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,1}$ $a_{3,2}$ $a_{3,3}$
A=B*C

$a_{1,2}=b_{1,1}*c_{1,2}+b_{1,2}*c_{2,2}$

into this (logical) architecture:



Matrix Multiplication Performance Engineering



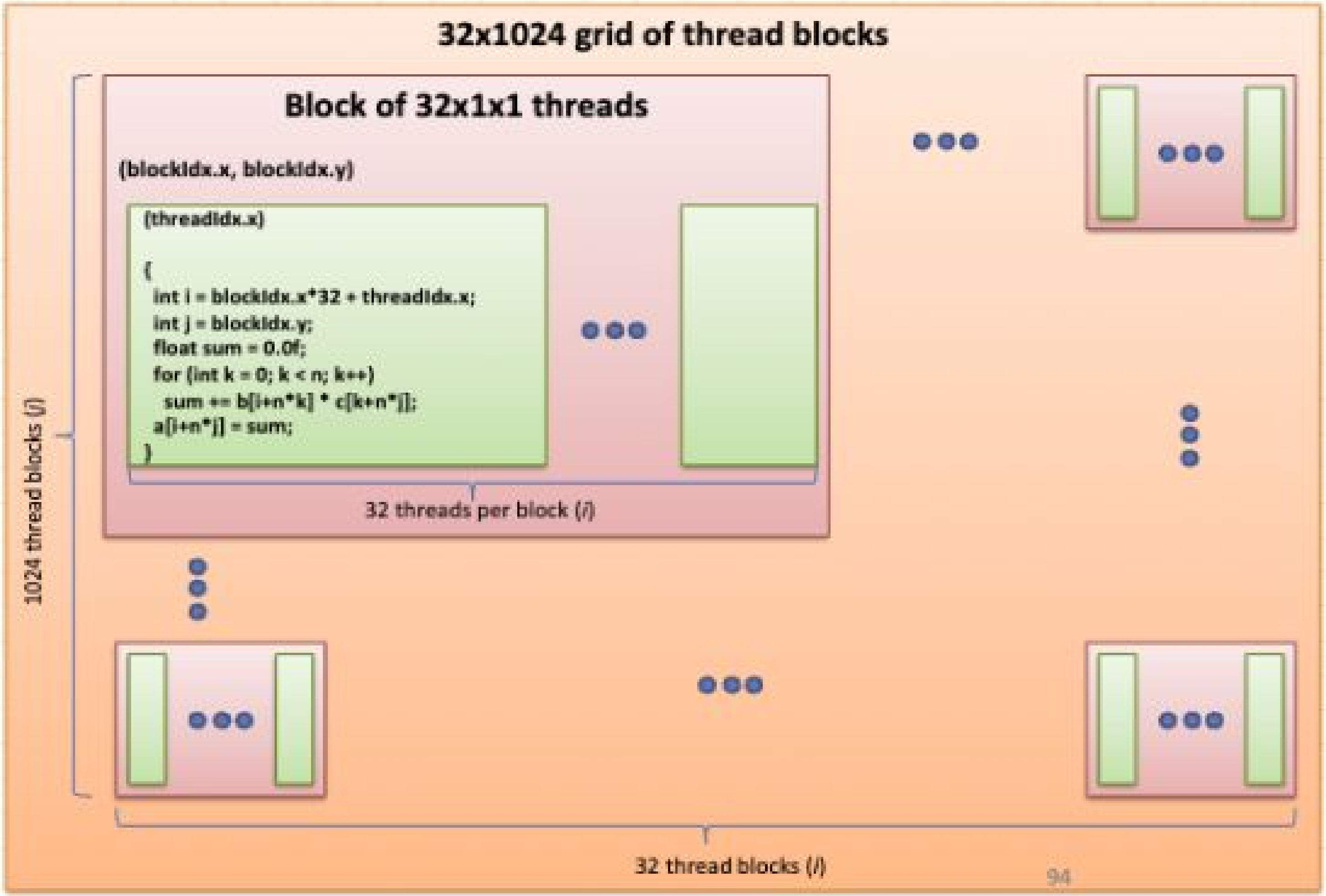
No faster than CPU

Results from NVIDIA P100

Coleman et. al., “Efficient CUDA,” 2017

Architecture knowledge is needed (again)

```
dim3 grid(1024/32, 1024);
dim3 threads (32);
```



```
{
  int i = blockIdx.x*32 + threadIdx.x;
  int j = blockIdx.y;
  float sum = 0.0f;
  for (int k = 0; k < n; k++)
    sum += b[i+n*k] * c[k+n*j];
  a[i+n*j] = sum;
}
```

Kernel

Original CPU kernel

```
void mmult(float *a, float *b, float *c, int N)
{
   ~~for (int i = 0; i < N; i++)~~
      ~~for (int j = 0; j < N; j++)~~
         for (int k = 0; k < N; k++)
            a[i+j*N] += b[i+k*N]*c[k+j*N];
}
```

GPU Kernel

```
__global__
void mmult(float *a, float *b, float *c, int N)
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   int j = blockIdx.y;
   float sum = 0.0;

   for (int k = 0; k < N; k++)
      sum += b[i+N*k] * c[k+N*j];

   a[i+N*j] = sum;
}
```

```
dim3 dimGrid(32, 1024);
dim3 dimBlock(32);
mmult<<<dimGrid, dimBlock>>>(devPtrA, devPtrB, devPtrC, N);
```

```
int main(int argc, char* argv[])
{
   int N = 1024;

   struct timeval t1, t2;
   long msec1, msec2;
   float flop, mflop, gflop;

   float *a = (float *)malloc(N*N*sizeof(float));
   float *b = (float *)malloc(N*N*sizeof(float));
   float *c = (float *)malloc(N*N*sizeof(float));

   minit(a, b, c, N);

   // allocate device memory
   float *devPtrA, *devPtrB, *devPtrC;
   cudaMalloc((void**)&devPtrA, N*N*sizeof(float));
   cudaMalloc((void**)&devPtrB, N*N*sizeof(float));
   cudaMalloc((void**)&devPtrC, N*N*sizeof(float));

   // copu input arrays to the device meory
   cudaMemcpy(devPtrB, b, N*N*sizeof(float),  cudaMemcpyHostToDevice);
   cudaMemcpy(devPtrC, c, N*N*sizeof(float),  cudaMemcpyHostToDevice);
```

```
gettimeofday(&t1, NULL);
msec1 = t1.tv_sec * 1000000 + t1.tv_usec;

// define grid and thread block sizes
dim3 dimGrid(32, 1024);
dim3 dimBlock(32);

// launch GPU kernel
mmult<<<dimGrid, dimBlock>>>(devPtrA, devPtrB, devPtrC, N);

// check for errors
cudaError_t err = cudaGetLastError();
if (cudaSuccess != err)
{
   fprintf(stderr, "CUDA error: %s.\n", cudaGetErrorString( err) );
   exit(EXIT_FAILURE);
}

// wait until GPU kernel is done
cudaThreadSynchronize();

gettimeofday(&t2, NULL);
msec2 = t2.tv_sec * 1000000 + t2.tv_usec;
```

```
    // copy results to host
    cudaMemcpy(a, devPtrA, N*N*sizeof(float),  cudaMemcpyDeviceToHost);

    mprint(a, N, 5);

    // free device memory
    cudaFree(devPtrA);
    cudaFree(devPtrB);
    cudaFree(devPtrC);

    free(a);
    free(b);
    free(c);

    msec2 -= msec1;
    flop = N*N*N*2.0f;
    mflop = flop / msec2;
    gflop = mflop / 1000.0f;
    printf("msec = %10ld   GFLOPS = %.3f\n", msec2, gflop);
}
```