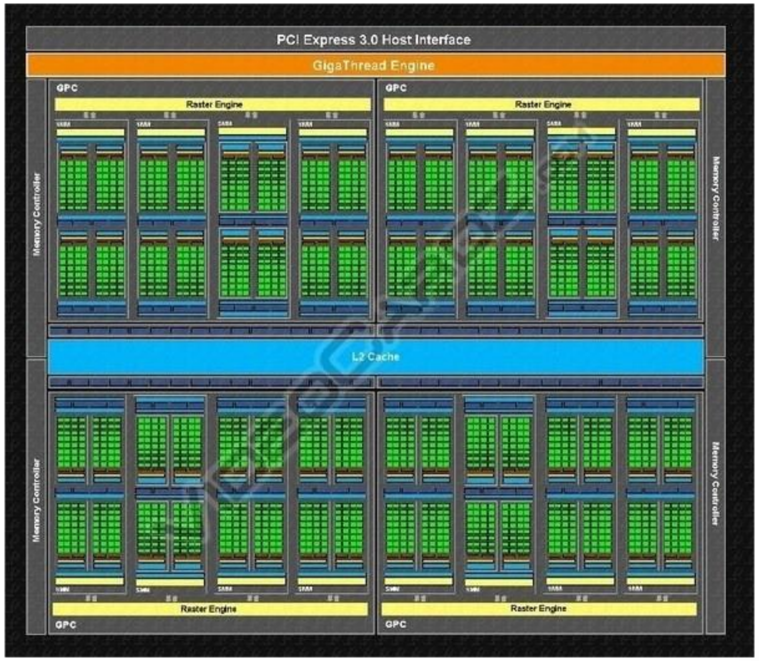
cse5441 - parallel computing

https://cdn8.bigcommerce.com/s-o1ibh36yyf/images/stencil/1280x1280/products/18124/29821/hpgraphiccardsvideocardstesla__52934.1516995900.jpg?c=2&imbypass=on
https://i0.wp.com/www.visitcocosisland.com/wp-content/uploads/2017/01/animalia-life-club.jpg?resize=640%2C492
https://qph.fs.quoracdn.net/main-qimg-257ea3b2d296c1605a50e5fd5db21cfa-c
https://pictures.topspeed.com/IMG/jpg/201205/plymouth-hemi-cuda-p.jpg



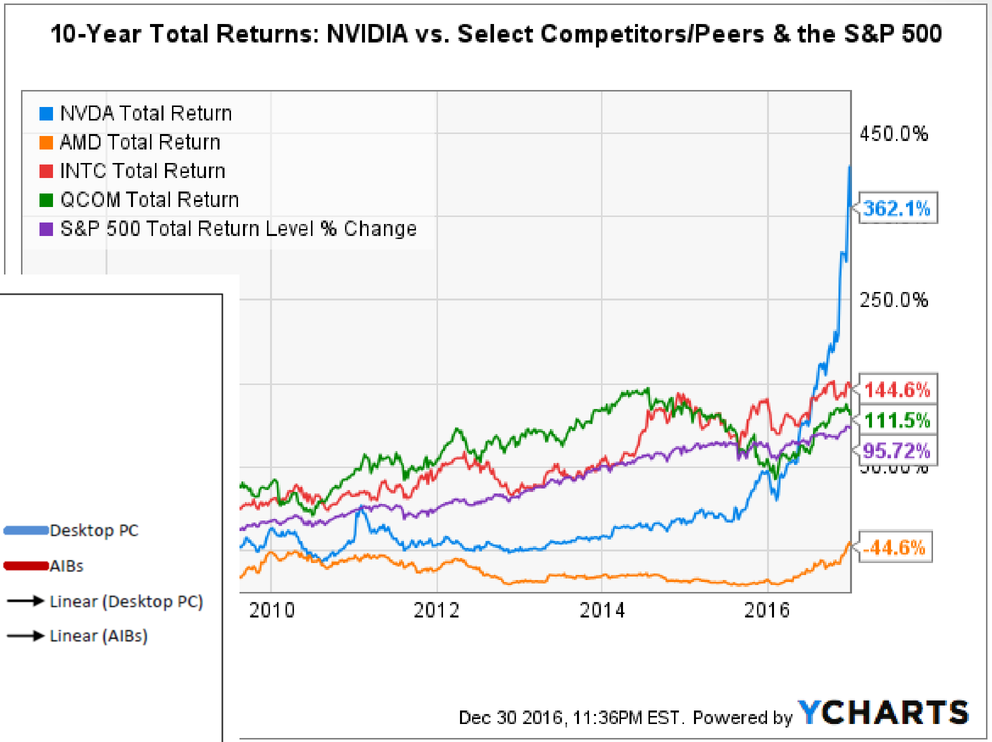
‘cuda

why cuda?

- robust development environment
- robust developer community
- significant market share
- leading feature development
- used by Ohio Supercomputer Center
- used by OSU High Performance Computing group

- quickly evolving environment

Nvidia developer’s forum: https://devtalk.nvidia.com/
Nvidia CUDA toolkit: http://docs.nvidia.com/cuda/index.html



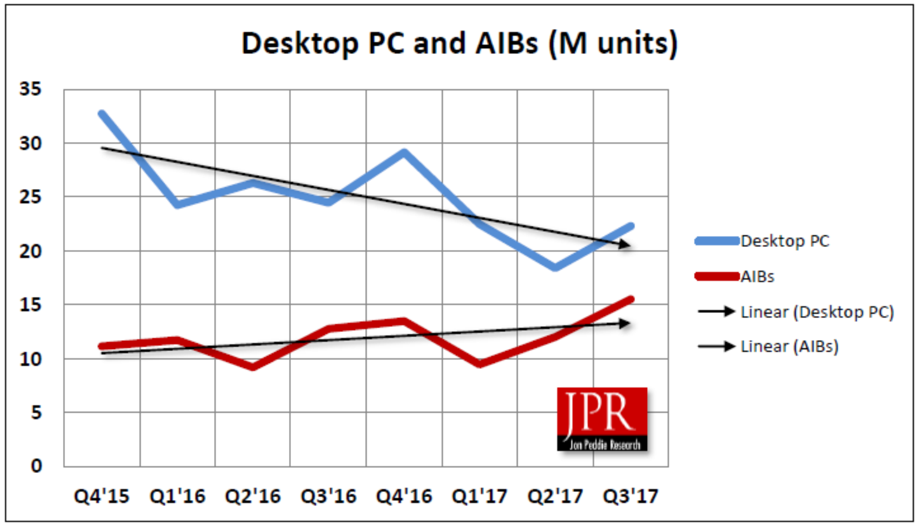
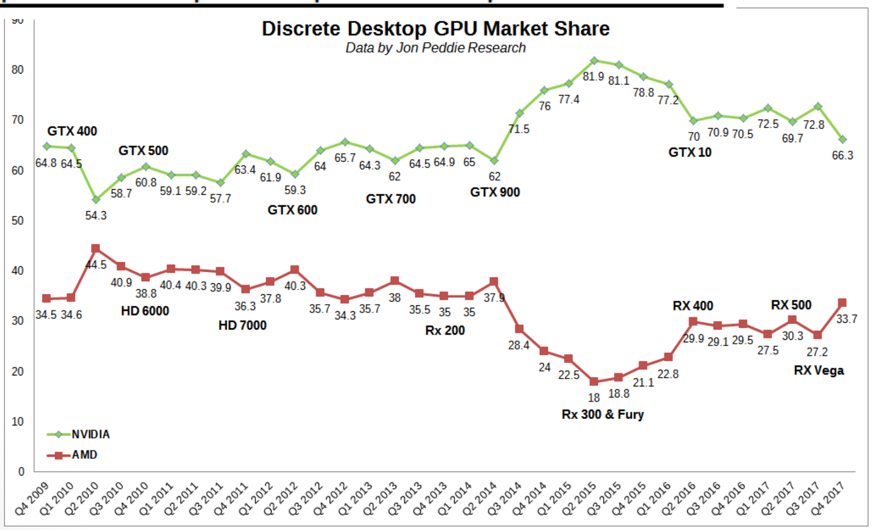


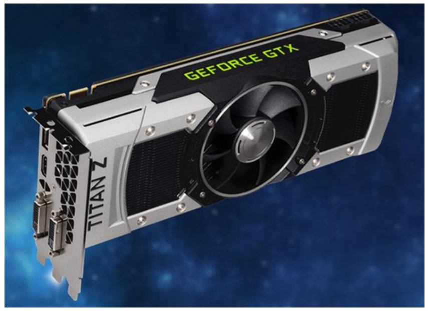
Figure 3: Shipments of desktop PCs compared to desktop AIBs over time



https://media.ycharts.com/charts/77796c8da1fbd66c389b74054ecf4d2d.png
https://cdn.wccftech.com/wp-content/uploads/2017/12/Q3-2017-Discrete-GPU-Market-Share-NVIDIA-AMD_3.png
https://cdn.wccftech.com/wp-content/uploads/2017/12/Q3-2017-Discrete-GPU-Market-Share-NVIDIA-AMD_1.png

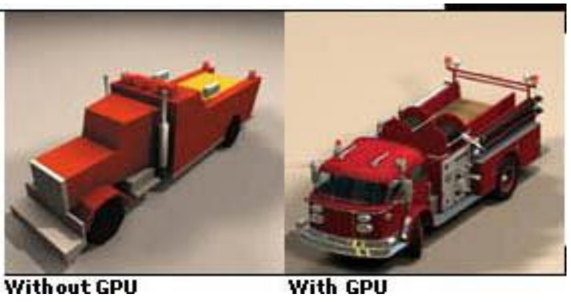
what is a gpu?

"a single chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second."



applications:

Abacus/Standard
Abinit
AcuSolve
ADF
Amber
ArrayFire
AxRecon
COSMO
Desmond
Empro
Fluent
Optistruct
RADIOSS
Savant
Semcad-X
TeraChem
Theano
Torch7
Tsunami RTM
UGENE
Vega ZZ
Wolfram

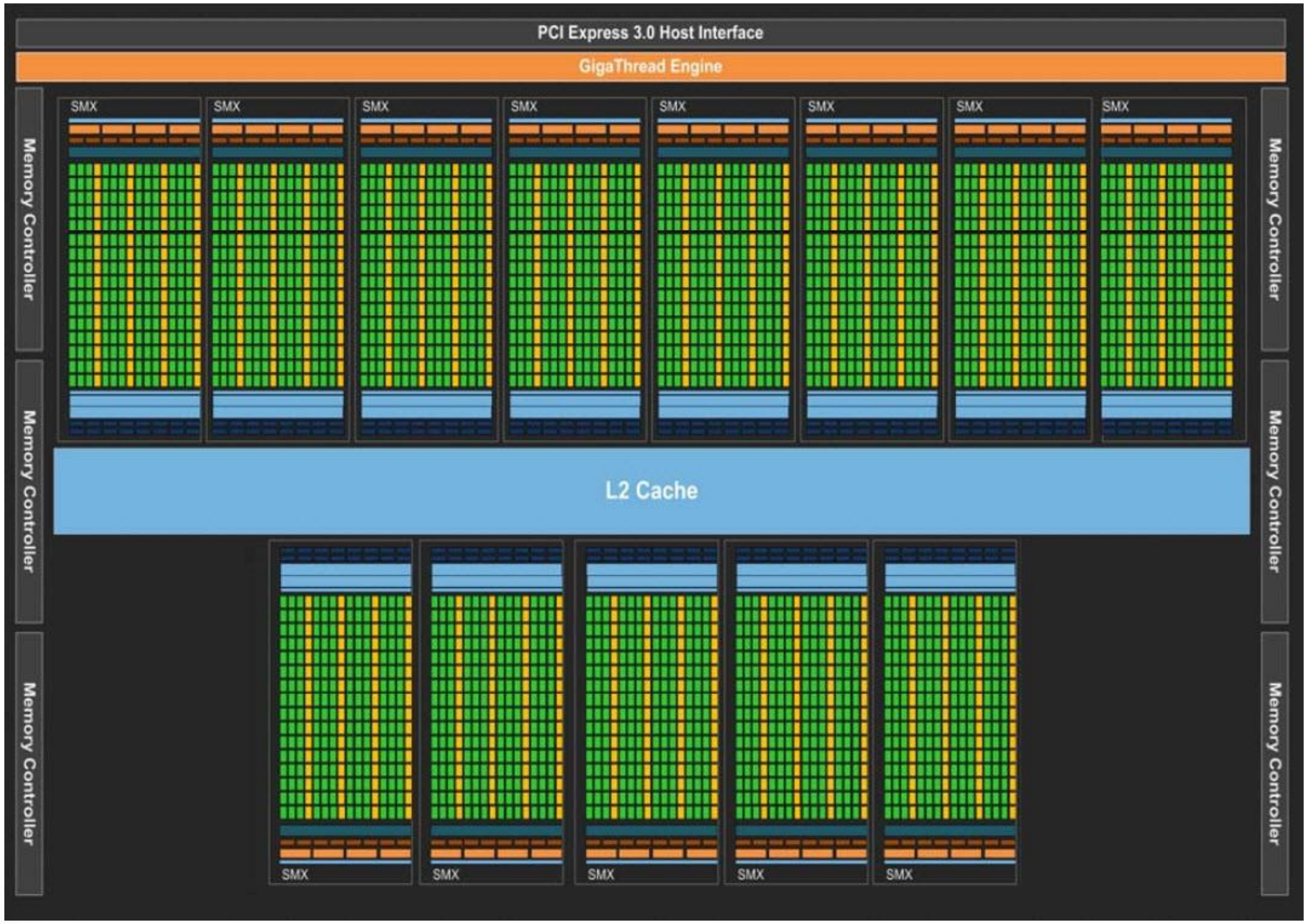


application fields:

computational finance
computational physics
database/data science
design automation
fluid dynamics
media & entertainment
medical imaging
molecular dynamics
numerical analysis
quantum chemistry
structural mechanics
weather & climate

h http://www.nvidia.com/object/gpu.html
ttps://upload.wikimedia.org/wikipedia/commons/4/44/6600GT_GPU.jpg
http://hothardware.com/newsimages/Item29925/EVGA_Superclocked_Titan_Z.jpg

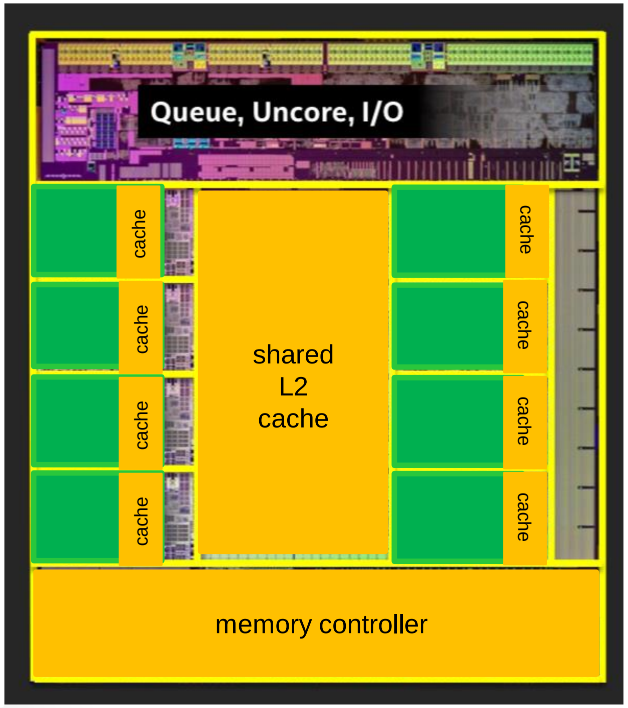
what is a gpu?



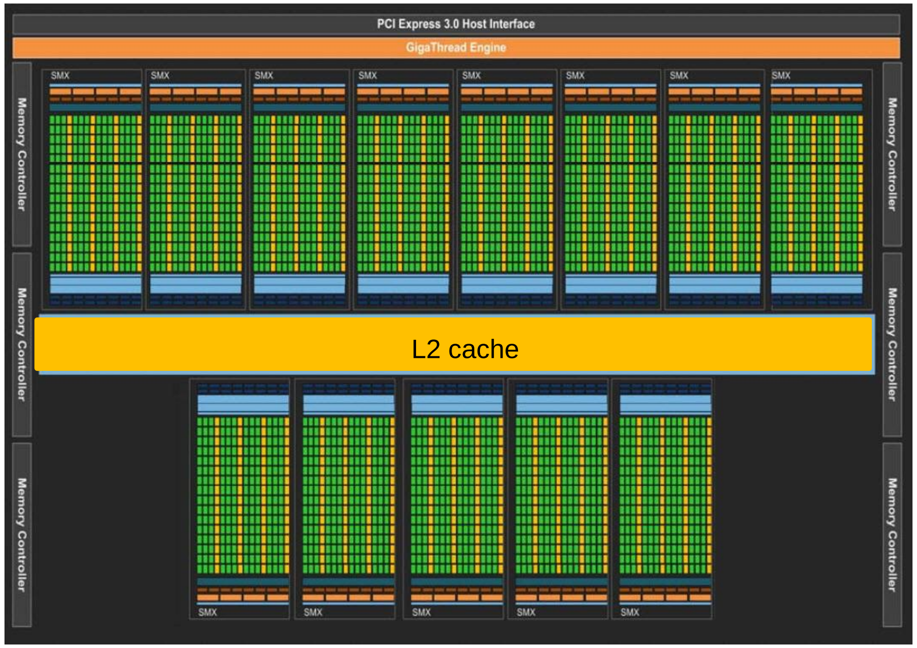
http://cdn.wccftech.com/wp-content/uploads/2013/03/GK110-GPU-Cut-down.jpg

gpu / cpu comparison

"CPU"



GPU



🟩 computational

🟨 memory-related

http://cdn.wccftech.com/wp-content/uploads/2013/03/GK110-GPU-Cut-down.jpg
http://cdn.arstechnica.net/wp-content/uploads/2014/08/Screen-Shot-2014-08-28-at-3.27.22-PM.png

pthreads -vs- OpenMP -vs- cuda

pthreads

- rigorous analysis of variable scope and sharing
- host processing
- global shared memory
- asynchronous peer threads

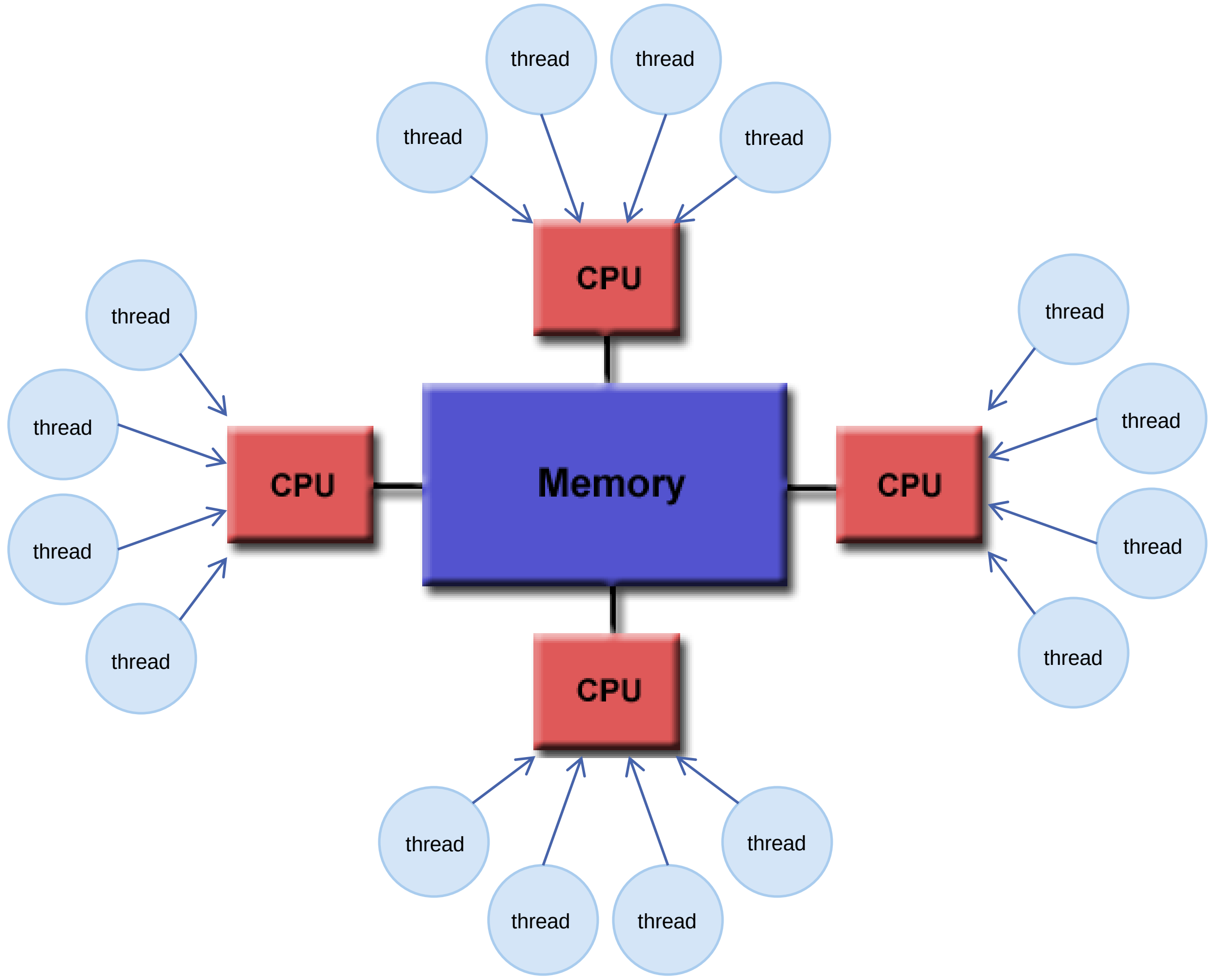
OpenMP

- rigorous analysis of variable scope and sharing
- host processing
- global shared memory
- asynchronous peer threads

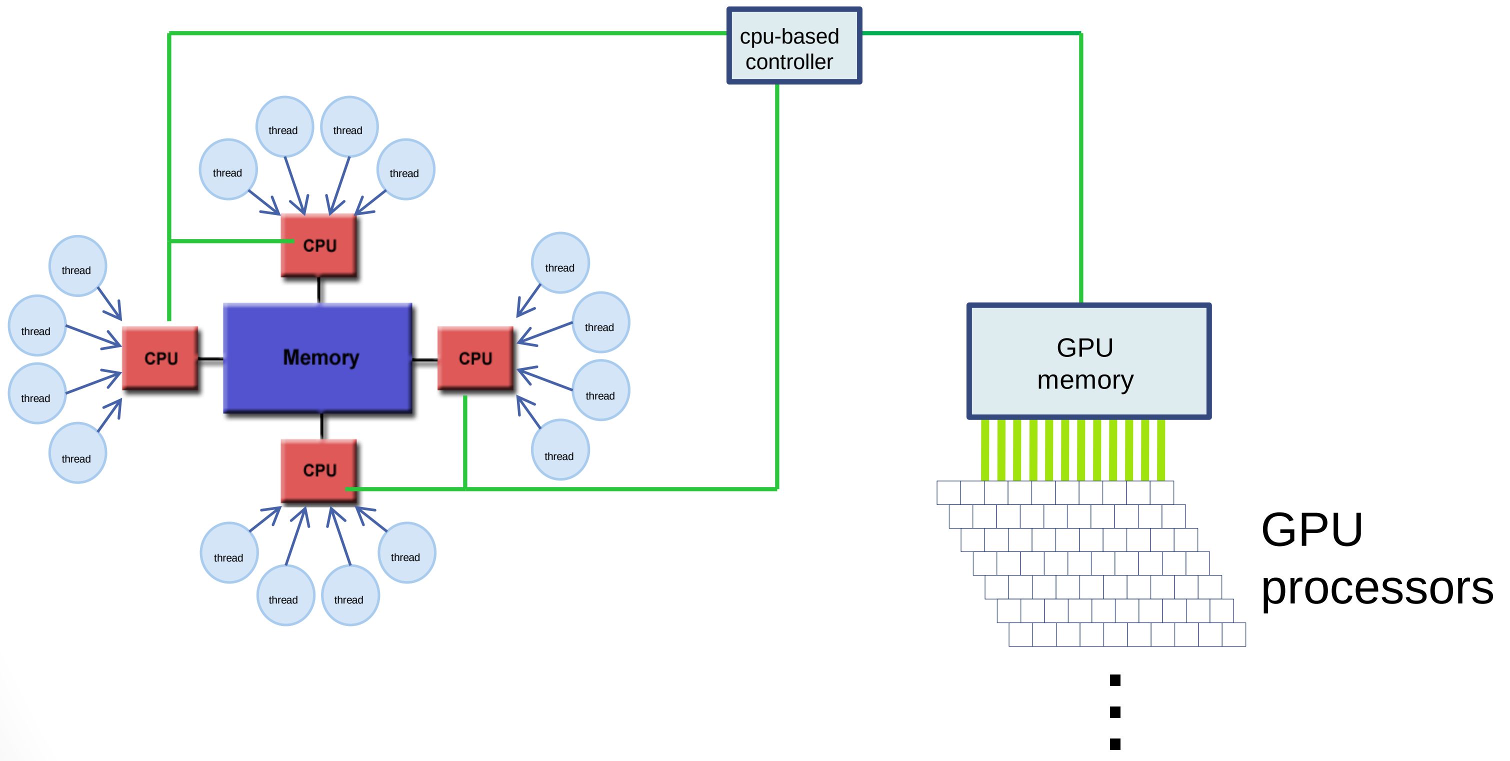
- rigorous analysis of variable scope and sharing
- device / host processing
- non-continguous, non-homogenous memory
- hierarchical, synchronous thread teams

http://www.mulberry-muse.com/wp-content/uploads/2013/02/MB_00574_Monty_Python_Hand.jpg

cpu global shared memory
review



cpu / gpu non-contiguous memory



“memory speed”

“parallel-access memory speed”

cuda terminology

• **host**	*a general-purpose computer (CPU) with which a GPU is associated*
•**device**	*a GPU which is associated with a CPU*
• **(cuda) program**	*a program written with both host and GPU (kernel) portions*
• **kernel**	*the GPU portion of a (cuda) program*
• **barrier**	*various synchronization methods which control the progress of threads*
• **thread hierarchy**	*an organizational method by which threads are organized*
• **grid** and **block**	*dimensional aspects of a thread hierarchy*

cuda terminology

• **thread**	*unit of kernel execution with a GPU*
• **thread index**	*the coordinates of a specific thread within a thread hierarchy*
• **thread id**	*the linearized value of the thread index*
• ***dim3***	*a structured type used to store the grid and block sizes, each of which themselves may be in 1, 2 or 3 dimensions*
• **SM**	“streaming multiprocessor,” low clock, tightly coupled, high data-width processor group
• **warp**	*unit of kernel scheduling, a group of threads which are executed simultaneously on a specific SM*

cuda kernel threads

a kernel is a data-parallel function

- invoking a kernel creates super-lightweight threads
- each thread has a context within the kernel
- threads are generated and scheduled by hardware
- Nvidia features 0-cycle thread swap

basic cuda application structure

ON HOST:

- instantiate host data structures
- perform initialization and pre-process data
- instantiate device data structures
- initialize device data strucutures
- define device thread hierarchy
- launch kernel

- copy results from device to host data structures
- de-allocate device data structures

- perform single-threaded application functions

- instantiate device data structures
- initialize device data strucutures
- define device thread hierarchy
- launch kernel

- copy results from device to host data structures
- de-allocate device data structures

- perform single-threaded application functions

(wash, rinse, repeat …)

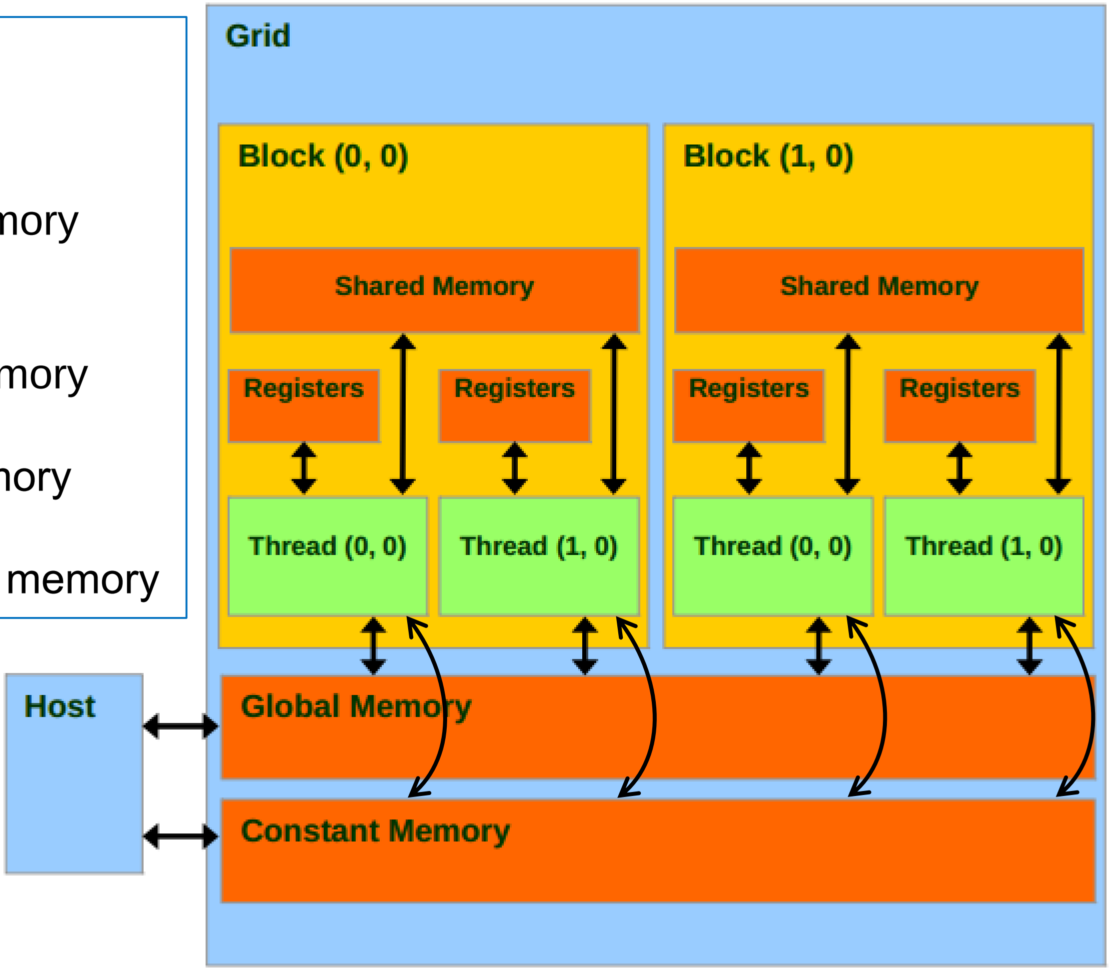
ON DEVICE:

cuda memory transfers

device side

device threads can:

- R/W per-thread registers
- R/W per-thread local memory (not shown)
- R/W per-block shared memory
- R/W per-grid "global" memory
- R-only per-grid "constant" memory



https://sidkashyap.files.wordpress.com/2013/05/cuda_memory_classes.png

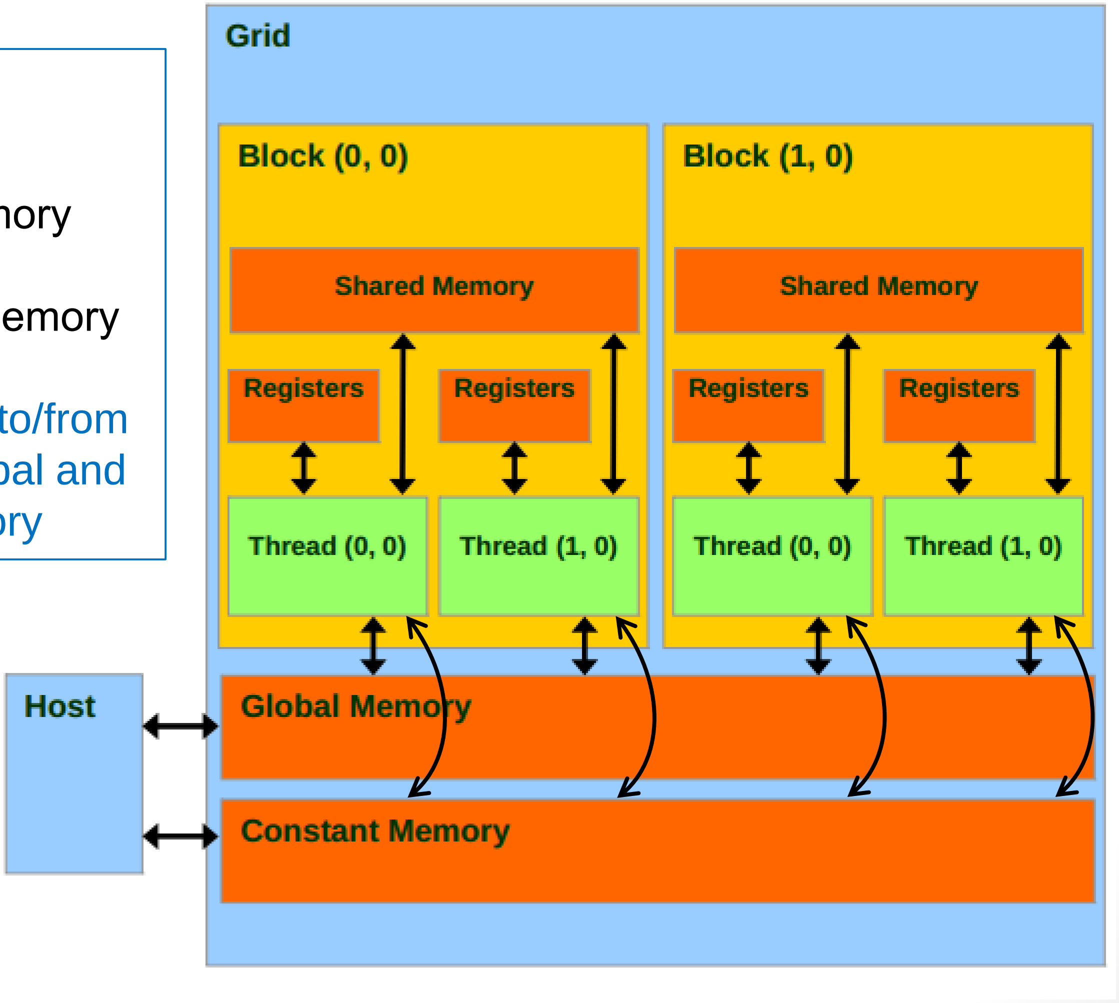
cuda memory transfers

host side

Host functions can:

- R/W per-grid “global” memory
- R/W per-grid “constant” memory

transfer memory contents to/from host memory to/from global and constant GPU memory



https://sidkashyap.files.wordpress.com/2013/05/cuda_memory_classes.png

cuda memory transfers - allocation

host program

```
int  *h_a;                  //declare pointer for host memory
int  *d_a;                  //declare pointer for device memory

// allocate host and device memory
size_t  memSize;

    memSize    = num_blocks * num_th_per_blk * sizeof(int);
    h_a          = (int*) malloc(memSize);
    cudaMalloc( (void**) &d_a, memSize);
```

sizeof "whatever"
int as an example
v

cuda memory transfers - copy

host program

```
int  *h_a;                  //declare pointer for host memory
int  *d_a;                  //declare pointer for device memory

// allocate host and device memory
size_t  memSize;

    memSize    = num_blocks * num_th_per_blk * sizeof(int);
    h_a            = (int*) malloc(memSize);
    cudaMalloc( (void**) &d_a, memSize);

    // first put interesting stuff into h_a, then ...
    cudaMemcpy( d_a, h_a, memSize, cudaMemcpyHostToDevice);
```

cuda memory transfers - retrieve

host program

```
int  *h_a;                  //declare pointer for host memory
int  *d_a;                  //declare pointer for device memory

// allocate host and device memory
size_t  memSize;

    memSize    = num_blocks * num_th_per_blk * sizeof(int);
    h_a        = (int*) malloc(memSize);
    cudaMalloc( (void**) &d_a, memSize);

    cudaMemcpy( d_a, h_a, memSize, cudaMemcpyHostToDevice);

    (launch kernel)

    cudaMemcpy( h_a, d_a, memSize, cudaMemcpyDeviceToHost);
    cudaFree(d_a);
```

cuda memory transfers

host program

```
int  *d1_a;                    //declare pointer for device memory
int  *d2_a;                    //declare another pointer for device memory

   cudaMemcpy( d2_a, d1_a, memSize, cudaMemcpyDeviceToDevice);
```

```
int  *h1_a;                    //declare pointer for host memory
int  *h2_a;                    //declare another pointer for host memory

   cudaMemcpy( h2_a, h1_a, memSize, cudaMemcpyHostToHost);
```

launching kernel

```
// launch kernel “initArray”

    dim3  dimGrid(num_blocks);
    dim3  dimBlock(num_th_per_blk);
    initArray<<< dimGrid, dimBlock >>>(d_a);
```

a simple cuda program

```
//  initialize an array using GPU

int main()
{
int  *h_a;                      //pointer for host memory
int  *d_a;                      //pointer for device memory

// define thread hierarchy
int  num_blocks         = 8;
int  num_th_per_blk   = 8;

// allocate host and device memory
size_t  memSize;

    memSize = num_blocks * num_th_per_blk * sizeof(int);
    h_a         = (int*) malloc(memSize);
    cudaMalloc( (void**) &d_a, memSize);

// launch kernel
    dim3  dimGrid(num_blocks);
    dim3  dimBlock(num_th_per_blk);
    initArray<<< dimGrid, dimBlock >>>(d_a);

// retrieve results
    cudaMemcpy( h_a, d_a, memSize,
                        cudaMemcpyDeviceToHost);
. . .
```

← for this example, assume we will have one thread per data item

```
//  kernel – find linearized threadId, and set A[ tid ] = tid

__global__ void initArray( int  *A)
{
int  tid;

    tid        =  blockIdx.x * blockDim.x + threadIdx.x;
    A[ tid ]  =  tid;
}
```

^ use "__global__ void" type for base cuda kernel

initializes an array according to the linearized thread_id

it's your turn . . . reverseArray_singleBlock

```
//  initialize an array using GPU

int main()
{
int  *h_a;                    //pointer for host memory
int  *d_a;                    //pointer for device memory

// define thread hierarchy
int  num_blocks         = 8;
int  num_th_per_blk   = 8;

// allocate host and device memory
size_t  memSize;

    memSize = num_blocks * num_th_per_blk * sizeof(int);
    h_a          = (int*) malloc(memSize);
    cudaMalloc( (void**) &d_a, memSize);

// launch kernel
    dim3  dimGrid(num_blocks);
    dim3  dimBlock(num_th_per_blk);
    initArray<<< dimGrid, dimBlock >>>(d_a);

// retrieve results
    cudaMemcpy( h_a, d_a, memSize,
                        cudaMemcpyDevice To Host);
. . .
```

```
//  kernel – find linearized threadId, and set A[ tid ] = tid

__global__ void initArray( int  *A)
{
int  tid;

   tid        =  blockIdx.x * blockDim.x + threadIdx.x;
   A[ tid ]  =  tid;
}
```

modify host and initArray kernel to:

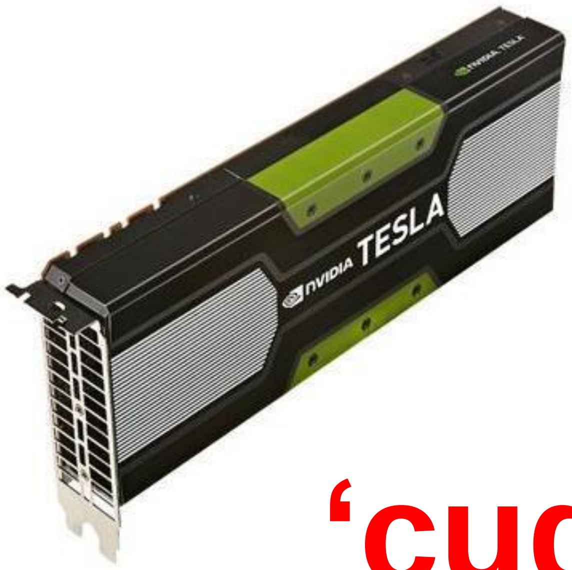
- use a single block of 256 threads
- use an array of 256 integers
- on host, initialize the values of an array to equal the array offset
- in the kernel, reverse the contents of the array
- on the host, verify the kernel's result

reverseArray_singleBlock

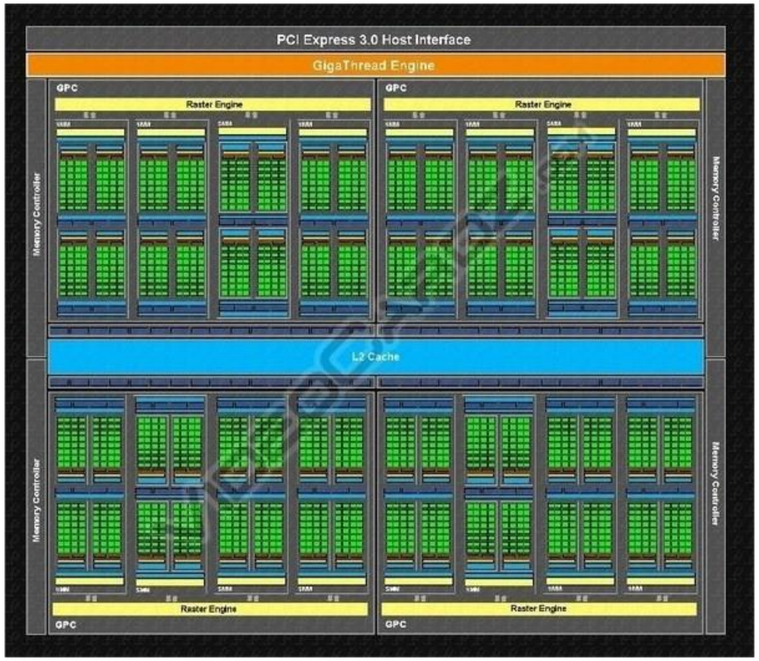
IYT answer

cse5441 - parallel computing

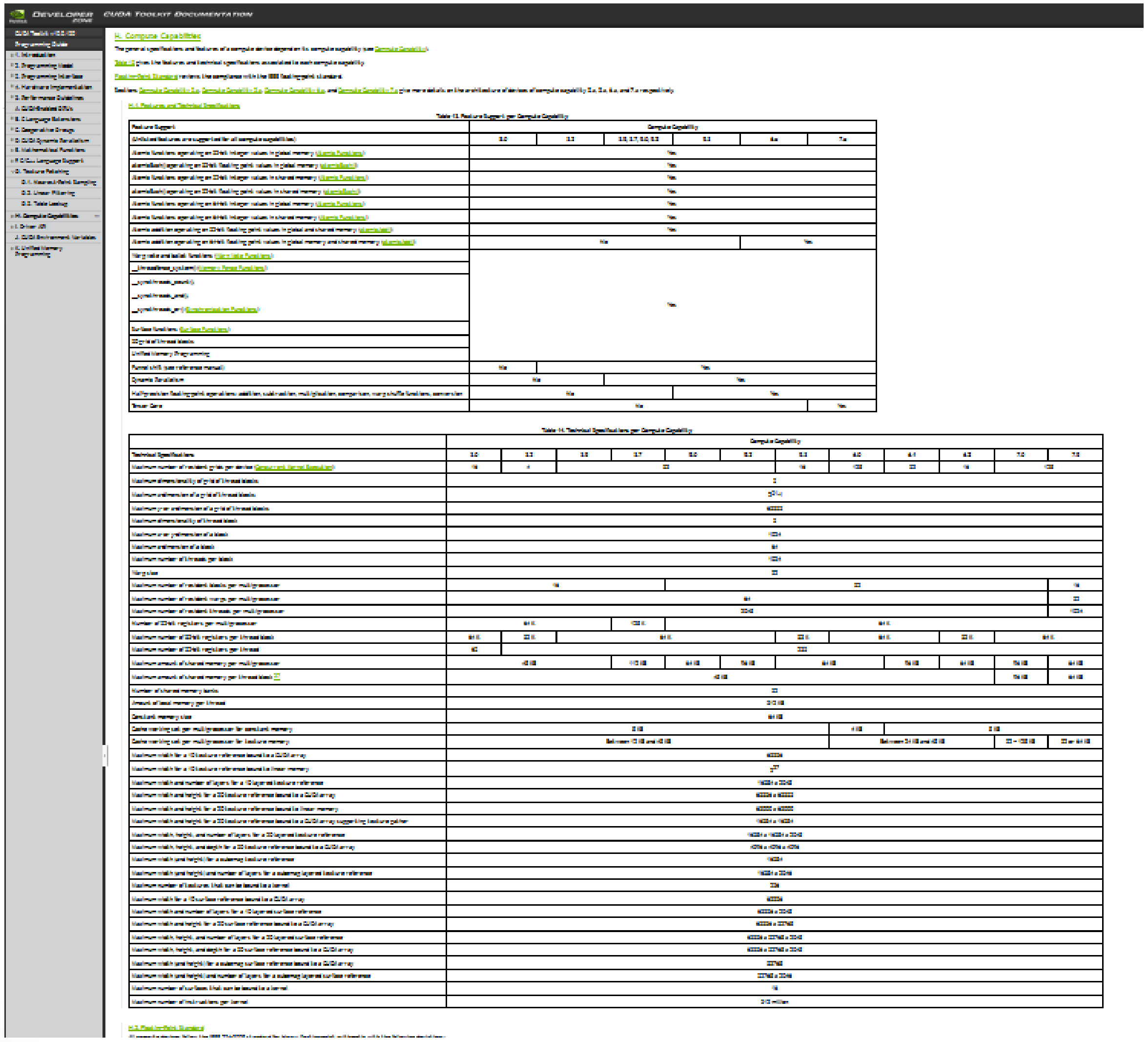
https://cdn8.bigcommerce.com/s-o1ibh36yyf/images/stencil/1280x1280/products/18124/29821/hpgraphiccardsvideocardstesla__52934.1516995900.jpg?c=2&imbypass=on
https://i0.wp.com/www.visitcocosisland.com/wp-content/uploads/2017/01/animalia-life-club.jpg?resize=640%2C492
https://qph.fs.quoracdn.net/main-qimg-257ea3b2d296c1605a50e5fd5db21cfa-c
https://pictures.topspeed.com/IMG/jpg/201205/plymouth-hemi-cuda-p.jpg



‘cuda

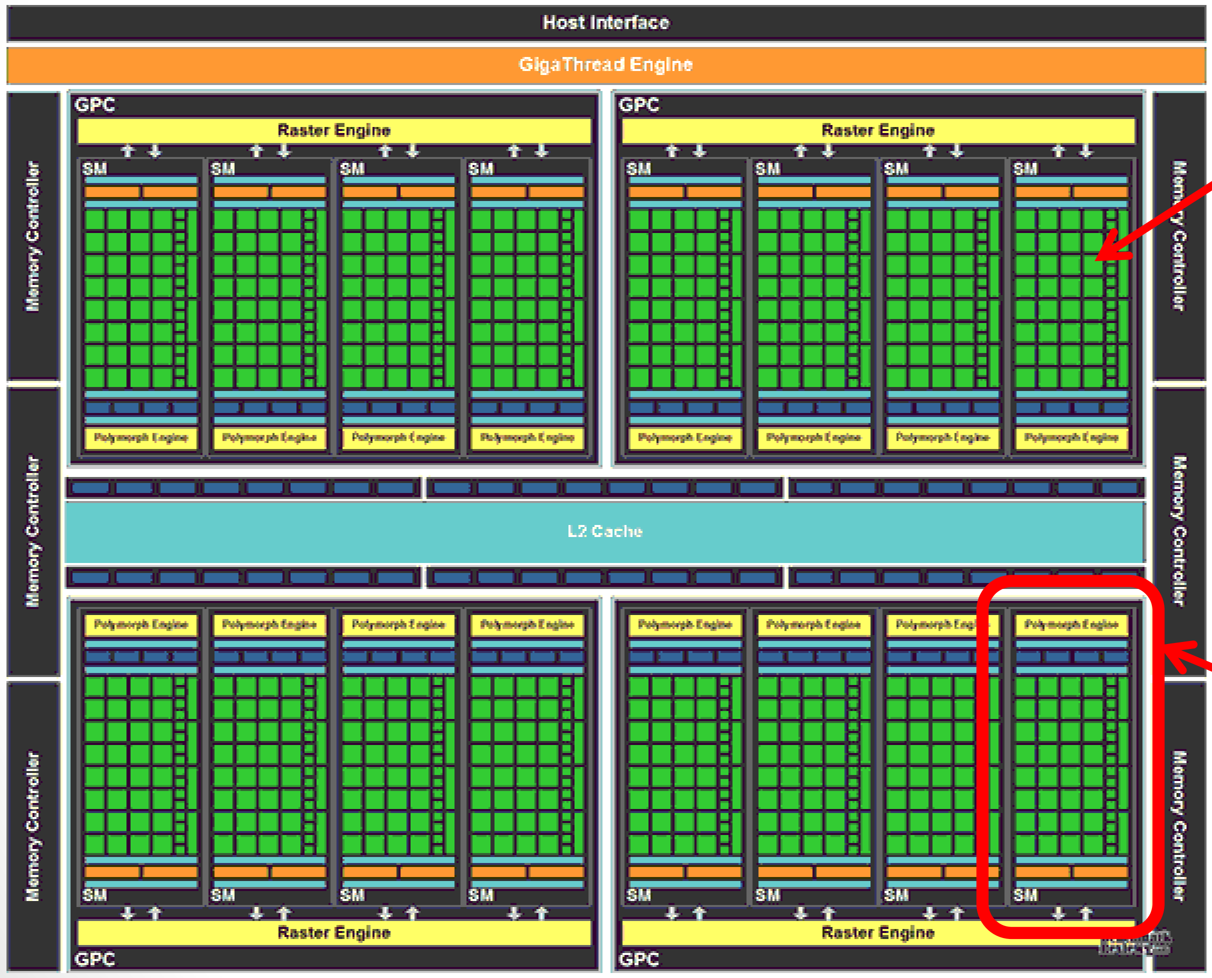


nVidia GPU compute capability



https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capabilities

“massively parallel” thread organization: threads



a thread runs on a core

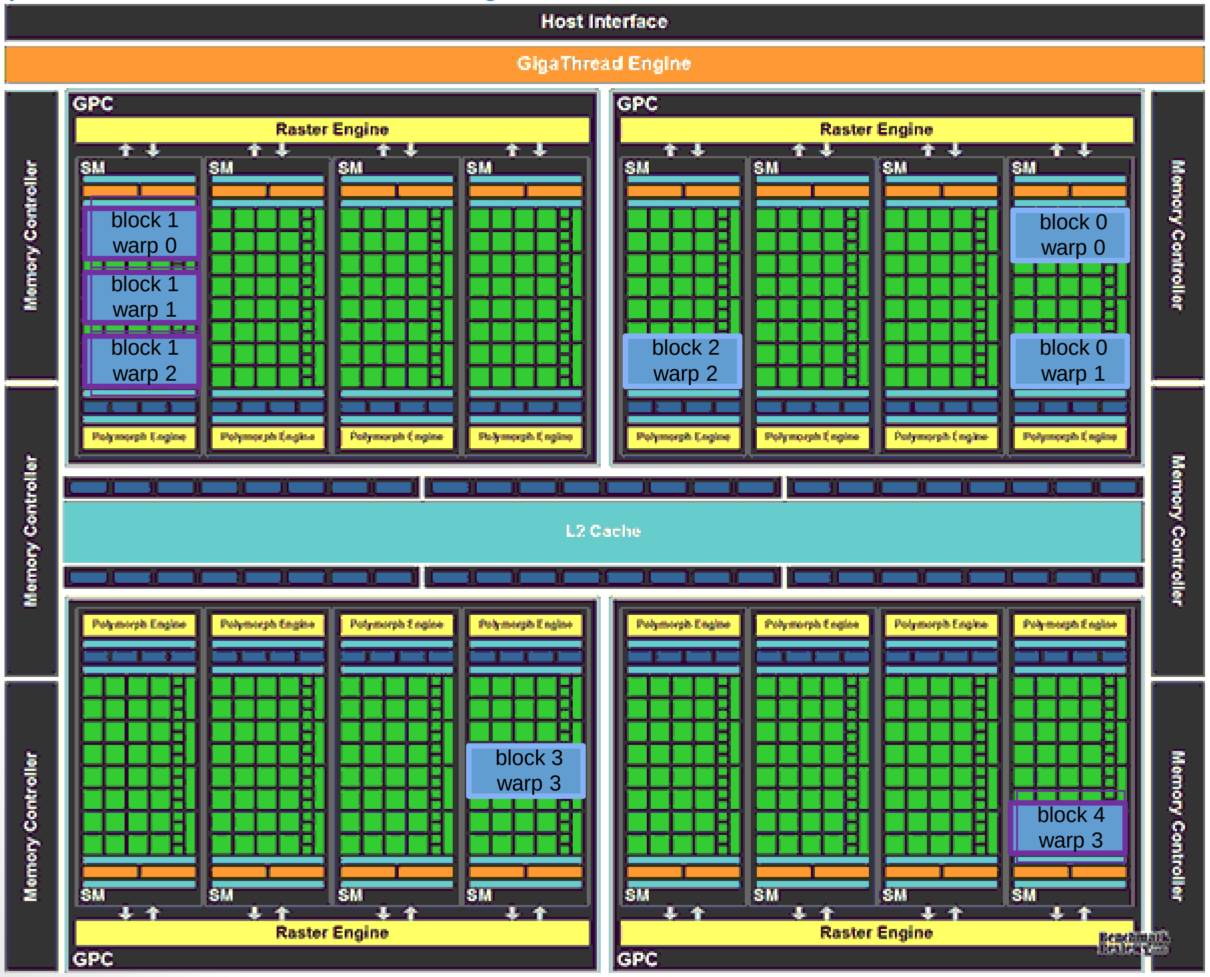
cores are components of streaming multiprocessors (SM)

http://archive.benchmarkreviews.com/images/reviews/processor/NVIDIA_Fermi/nvidia-fermi-gf100-gpu-block-diagram-benchmarkreviews-sm.png

http://archive.benchmarkreviews.com/images/reviews/processor/NVIDIA_Fermi/nvidia-fermi-gf100-gpu-block-diagram-benchmarkreviews-sm.png

"massively parallel" thread organization: blocks

currently all blocks from the same grid run on the same SM
plans are for block scheduling across SMs



threads are organized into **blocks**

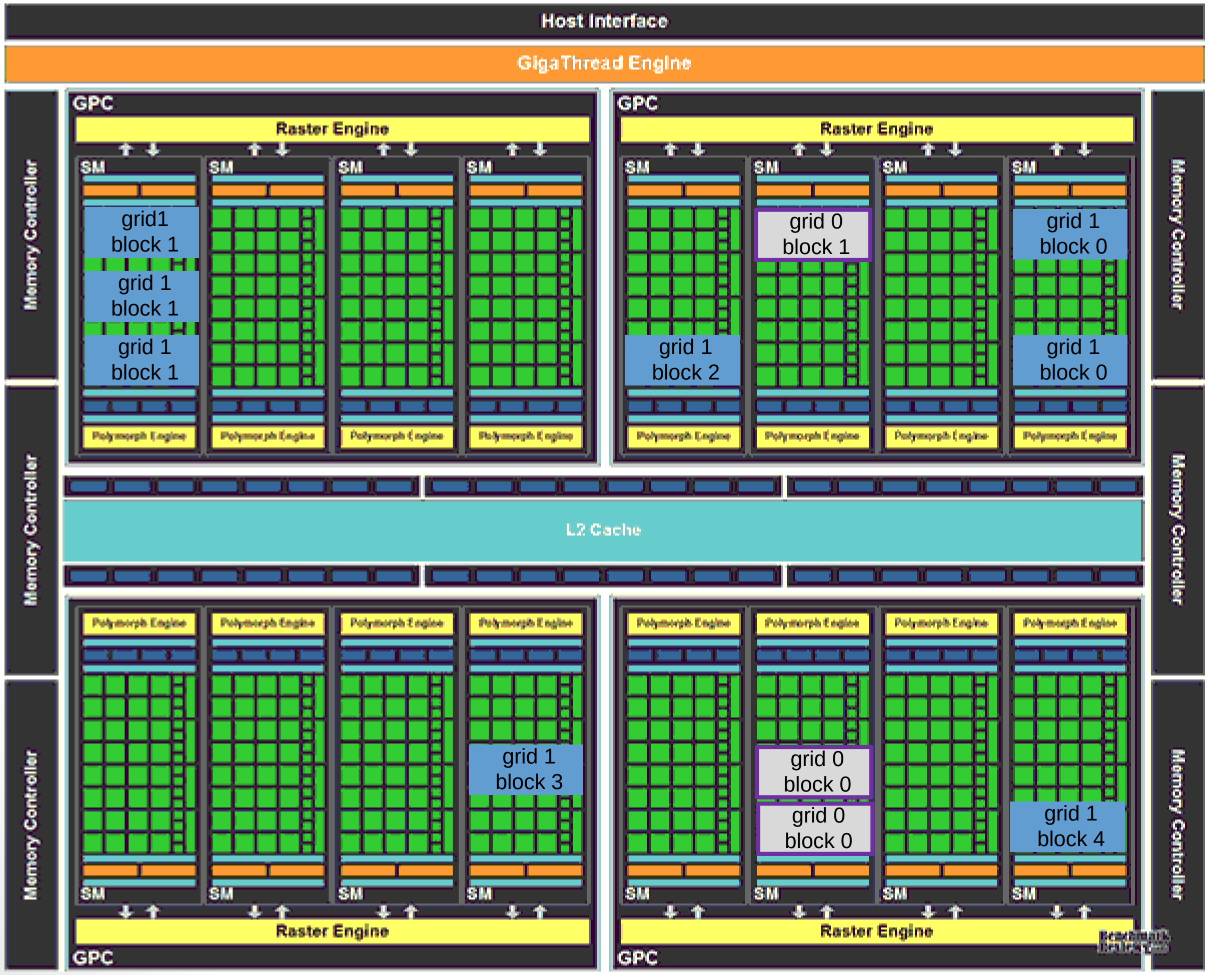
threads within a **block** are organized into **warps** of 32 threads

warps are scheduled atomically and all **block warps** run on the same **streaming multiprocessor** (SM)

note: warps are stylized for illustration purposes

“massively parallel” thread organization: grids

warps from 2 different grids running simultaneously on a single GPU



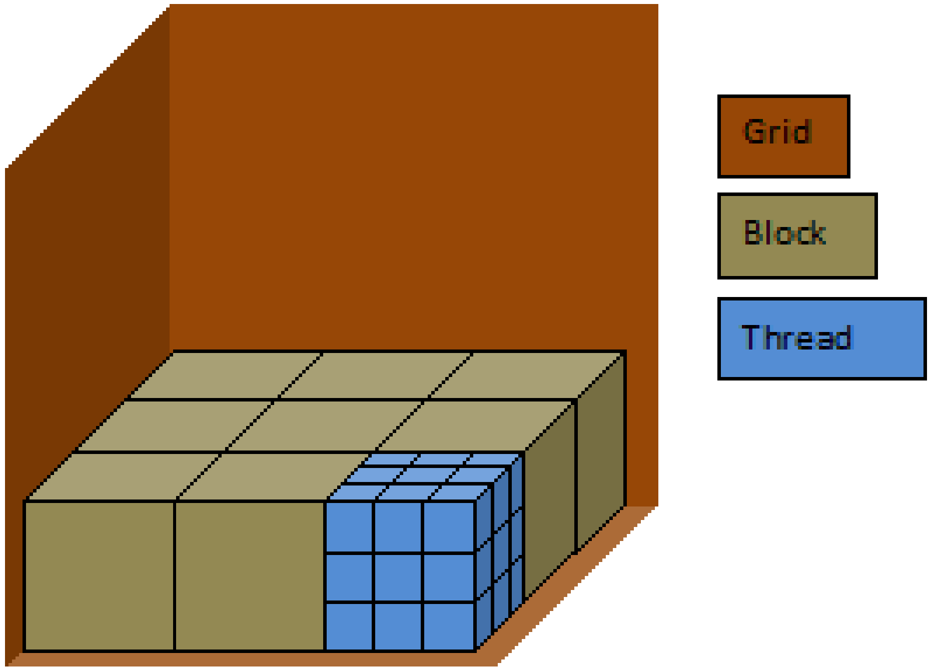
a group of **blocks** is a **grid**

each **kernel** gets one **grid** of **threads**

a **GPU** may schedule multiple simultaneous **grids**

http://archive.benchmarkreviews.com/images/reviews/processor/NVIDIA_Fermi/nvidia-fermi-gf100-gpu-block-diagram-benchmarkreviews-sm.png

“massively parallel” thread organization



http://ch.arrer.net/external/wordimg/102712_1431_MatrixMulti1.png

“massively parallel” thread organization:

threads in blocks, 1 block

threadIdx.x
thread position in dimension x
threadIdx.y
thread position in dimension y
threadIdx.z
thread position in dimension z

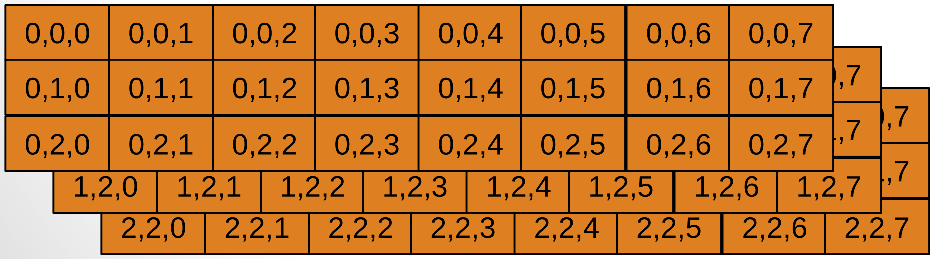
1D: threadIdx.x

0	1	2	3	4	5	6	7

2D: threadIdx.x * threadIdx.y

0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7
1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7
2,0	2,1	2,2	2,3	2,4	2,5	2,6	2,7

3D: threadIdx.x * threadIdx.y * threadIdx.z



0,0,0	0,0,1	0,0,2	0,0,3	0,0,4	0,0,5	0,0,6	0,0,7
0,1,0	0,1,1	0,1,2	0,1,3	0,1,4	0,1,5	0,1,6	0,1,7
0,2,0	0,2,1	0,2,2	0,2,3	0,2,4	0,2,5	0,2,6	0,2,7

1,2,0	1,2,1	1,2,2	1,2,3	1,2,4	1,2,5	1,2,6	1,2,7

2,2,0	2,2,1	2,2,2	2,2,3	2,2,4	2,2,5	2,2,6	2,2,7

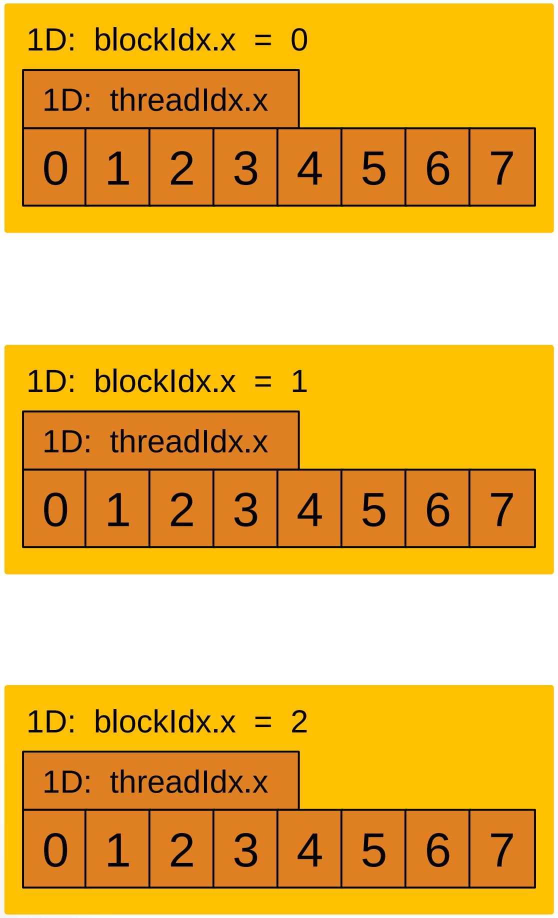
```
1-dimensional:
   dim3  dimBlock(n_th_per_blk);
   blk_t_id  = threadIdx.x

2-dimensional:
   dim3  dimBlock(nth_x, nth_y);
   blk_t_id  = blockDim.x * threadIdx.y
                    + threadIdx.x

3-dimensional;
   dim3  dimBlock(nth_x, nth_y, nth_z)
   blk_t_id  = threadIDx.z* blockDim.y *blockDim.x
                    + blockDim.x * threadIdx.y
                    + threadIdx.x
```

“massively parallel” thread organization:

threads in blocks, 1D x 1D



blockIdx.x	block position in dimension x
blockIdx.y	block position in dimension y
blockIdx.z	blockposition in dimension z

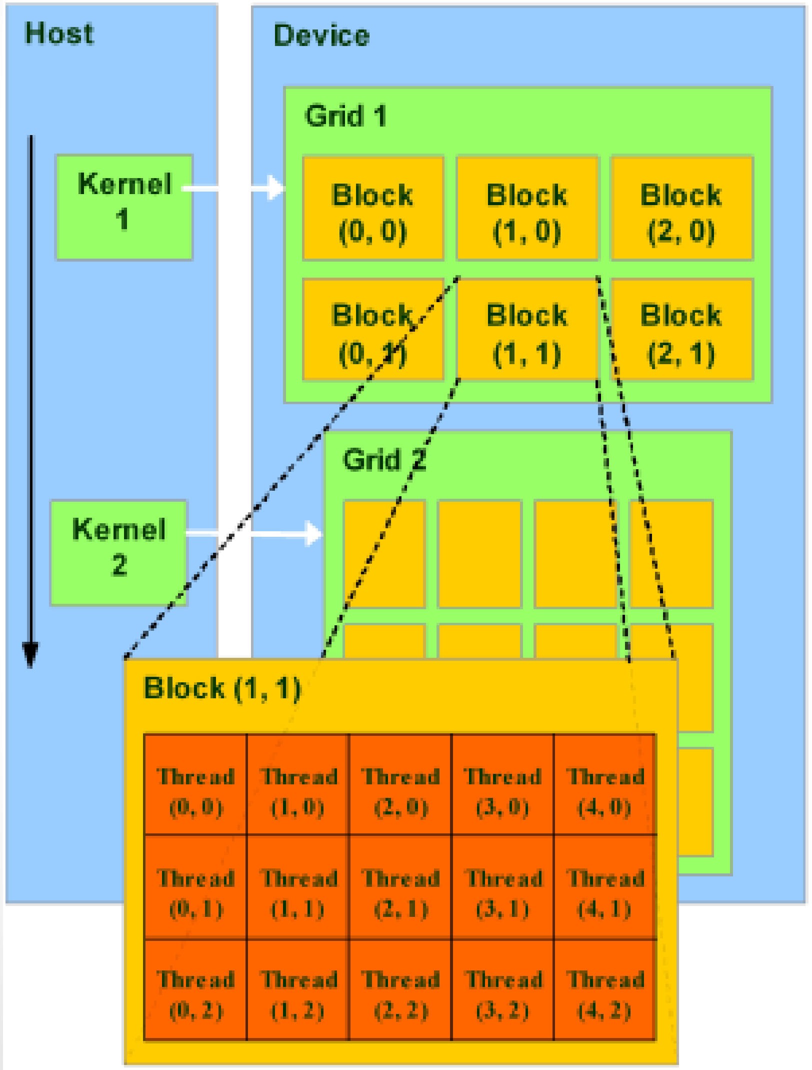
```
1-dimensional  x 1-dimensional:
   dim3  dimBlock(n_th_per_blk);
   dim3  dimGrid(n_blk_per_grid);

   blockNumInGrid     = blockIdx.x
   threadsPerBlock    = blockDim.x
   threadNumInBlock = threadIdx.x

   global_t_id  = blockidx.x * blockDim.x
                       + threadIdx.x
```

“massively parallel” thread organization:

threads in blocks, 2D x 2D



http://www.ixbt.com/video3/images/cuda/cuda3.png

blockIdx.x
block position in dimension x

blockIdx.y
block position in dimension y

// if we had a third dimension
(blockIdx.z block position in dimension z)

2-dimensional x 2-dimensional:

```
   dim3  dimBlock(tpb_x, tpb_y);
   dim3  dimGrid(bpg_x, bpg_y);

   blockNumInGrid      = blockIdx.x +
                         gridDim.x * blockIdx.y
   threadsPerBlock     = blockDim.x * blockDim.y
   threadNumInBlock = threadIdx.x +
                         blockDim.x * threadIdx.y

   global_t_id  = blockNumInGrid
                    * threadsPerBlock
                    + threadNumInBlock
```

note: this illustration in column-major

it’s your turn: reverseArray_multiBlock1

```
//  reverse an array using GPU

int main()
{
int  *h_a;                              //pointer for host memory
int  *h_v;                              //pointer verification copy
int  *d_a;                              //pointer for device input
int  *d_b;                              //pointer for device output

// define thread hierarchy
int  nblocks     = 1;
int  dimA        =  256;
int  tpb         = dimA;

// allocate host and device memory
size_t  memSize;
    memSize = nblocks * tpb * sizeof(int);
    h_a         = (int*) malloc(memSize);
    h_v         = (int*) malloc(memSize);
    cudaMalloc( (void**) &d_a, memSize);
    cudaMalloc( (void**) &d_b, memSize);

// initialize host arrays, copy to device
    for (int i = 0; i < dimA; i++)
    {
        h_a[i] = i;
        h_v[i] = i;
    }
    cudaMemcpy(d_a, h_a, memSize,
                        cudaMemcpyHostToDevice
```

```
// launch kernel
    dim3  dimGrid(nblocks);
    dim3  dimBlock(tpb);
    reverseArrayMBlock1<<< dimGrid, dimBlock >>>(d_a, d_b);

// get results
    cudaMemcpy( h_a, d_b, memSize,
                    cudaMemcpyDevice To Host);
. . .

//  kernel – reverse inArray into outArray
__global__ void reverseArrayMBlock1( int  *inArray, int *outArray)
{
int  tid;

    outArray[blockDim.x – 1 – threadIdx.x] = inArray[threadIdx.x]
```

modify host and kernel to:
- use 2 blocks of 1024 threads each
- use a linear array of 1M (1024x1024) integers
- on host, initialize the values of the array to equal the array offset
- in the kernel, reverse the contents of the array
- on the host, verify the kernel’s result

reverseArrayMultiBlock1

IYT answer

it's your turn: reverseArray_Block2

```
//  reverse an array using GPU

int main()
{
int  *h_a;                          //pointer for host memory
int  *h_v;                          //pointer verification copy
int  *d_a;                          //pointer for device input
int  *d_b;                          //pointer for device output
int  dimA    =  1024*1024;          //size of array


// define thread hierarchy
int  nblocks      = 2;
int  tpb          = 1024;

// allocate host and device memory
size_t  memSize;
   memSize = nblocks * tpb * sizeof(int);
   h_a          = (int*) malloc(memSize);
   h_v          = (int*) malloc(memSize);
   cudaMalloc( (void**) &d_a, memSize);
   cudaMalloc( (void**) &d_b, memSize);

// initialize host arrays, copy to device
   for (int i = 0; i < dimA; i++)
   {
      h_a[i] = i;
      h_v[i] = i;
   }
   cudaMemcpy(d_a, h_a, memSize,
                  cudaMemcpyHostToDevice
```

```
// launch kernel
   dim3  dimGrid(nblocks);
   dim3  dimBlock(tpb);
   reverseArrayBlock2<<< dimGrid, dimBlock >>>(d_a, d_b);

// get results
   cudaMemcpy( h_a, d_b, memSize,
                  cudaMemcpyDevice To Host);
. . .

//  kernel – reverse inArray into outArray
__global__ void reverseArray_Block2( int  *inArray, int *outArray)
{
int  tid;

   outArray[blockDim.x – 1 – threadIdx.x] = inArray[threadIdx.x]
```

in the kernel, reverse the contents of the array
as before, except:

- use a single block of 1024 threads
- use only a single on-device array

reverseArrayMultiBlock2

IYT answer

matrix multiply example

CPU implementation

```
int main()
{
float *A, *B, *C;  int dim = 512;
A = new float(dim*dim); B = new float(dim*dim);
   C = new float(dim*dim);
// I/O to load A and B .
   MatrixMulHost( A, B, C, dim);
}

void  MatrixMulHost (float *A, float *B,  float *C, int  dim)
{
float  a, b, sum;

   for (int i = 0;  i < dim;  i++)
   {
      for (int j = 0;  j < dim;  j++)
      {
         sum = 0;
         for (int k = 0;  k < dim;  k++)
         {
            a        = A[ i  * dim + k ];
            b        = B[ k * dim + j  ];
            sum + = a * b;
         }
         C[ i * dim + j ] = sum;
      }
   }
}
```

matrix multiply example

GPU skeleton

```
int main()
{
float *A, *B, *C;  int dim = 512;
A = new float(dim*dim); B = new float(dim*dim);
   C = new float(dim*dim);
// I/O to load A and B ...
   MatrixMulHost( A, B, C, dim);
}

void  MatrixMulHost (float *A, float *B,  float *C, int  dim)
{
float  a, b, sum;

   for (int i = 0;  i < dim;  i++)
   {
      for (int j = 0;  j < dim;  j++)
      {
         sum = 0;
         for (int k = 0;  k < dim;  k++)
         {
            a        = A[ i  * dim + k ];
            b        = B[ k * dim + j  ];
            sum + = a * b;
         }
         C[ i * dim + j ] = sum;
      }
   }
}
```

```
int main(void)
{
float *A, *B, *C;  int dim = 512;
A = new float(dim*dim); B = new float(dim*dim);
   C = new float(dim*dim);
// I/O to load A and B ...

// perform MatrixMul on Device

}
```

matrix multiply example

GPU skeleton

```
int main()
{
float *A, *B, *C;  int dim = 512;
A = new float(dim*dim); B = new float(dim*dim);
   C = new float(dim*dim);
// I/O to load A and B .
   MatrixMulHost( A, B, C, dim);
}

void  MatrixMulHost (float *A, float *B,  float *C, int  dim)
{
float  a, b, sum;

   for (int i = 0;  i < dim;  i++)
   {
      for (int j = 0;  j < dim;  j++)
      {
         sum = 0;
         for (int k = 0;  k < dim;  k++)
         {
            a        = A[ i  * dim + k ];
            b        = B[ k * dim + j  ];
            sum + = a * b;
         }
         C[ i * dim + j ] = sum;
      }
   }
}
```

```
int main(void)
{
float *A, *B, *C;  int dim = 512;
A = new float(dim*dim); B = new float(dim*dim);
   C = new float(dim*dim);
// I/O to load A and B .

// define thread hierarchy
// allocate device memory
// initialize device memory
// launch kernel
  // perform MatrixMul on Device
// retrieve results

}
```

matrix multiply example

GPU thread hierarchy

```
int main()
{
float *A, *B, *C;  int dim = 512;
A = new float(dim*dim); B = new float(dim*dim);
    C = new float(dim*dim);
// I/O to load A and B .
    MatrixMulHost( A, B, C, dim);
}

void  MatrixMulHost (float *A, float *B,  float *C, int  dim)
{
float  a, b, sum;

    for (int i = 0;  i < dim;  i++)
    {
        for (int j = 0;  j < dim;  j++)
        {
            sum = 0;
            for (int k = 0;  k < dim;  k++)
            {
                a         = A[ i  * dim + k ];
                b         = B[ k * dim + j  ];
                sum + = a * b;
            }
            C[ i * dim + j ] = sum;
        }
    }
}
```

```
int main(void)
{
float *A, *B, *C;  int dim = 512;
A = new float(dim*dim); B = new float(dim*dim);
    C = new float(dim*dim);
// I/O to load A and B .

// define thread hierarchy
int  nblocks      = 4;
int  tpb          = 512;

// allocate device memory
// initialize device memory
// launch kernel
   // perform MatrixMul on Device
// retrieve results

}
```

matrix multiply example

GPU memory allocation

```
int main()
{
float *A, *B, *C;  int dim = 512;
A = new float(dim*dim); B = new float(dim*dim);
    C = new float(dim*dim);
// I/O to load A and B .
    MatrixMulHost( A, B, C, dim);
}

void  MatrixMulHost (float *A, float *B,  float *C, int  dim)
{
float  a, b, sum;

    for (int i = 0;  i < dim;  i++)
    {
        for (int j = 0;  j < dim;  j++)
        {
            sum = 0;
            for (int k = 0;  k < dim;  k++)
            {
                a         = A[ i  * dim + k ];
                b         = B[ k * dim + j  ];
                sum + = a * b;
            }
            C[ i * dim + j ] = sum;
        }
    }
}
```

```
int main(void)
{
float *A, *B, *C;  int dim = 512;
A = new float(dim*dim); B = new float(dim*dim);
    C = new float(dim*dim);
// I/O to load A and B .

// define thread hierarchy
int  nblocks      = 4;
int  tpb          = 512;

// allocate device memory
size_t  memSize;
    memSize = dim * dim * sizeof(float);
    cudaMalloc( (void**) &d_A, memSize);
    cudaMalloc( (void**) &d_B, memSize);
    cudaMalloc( (void**) &d_C, memSize);


// initialize device memory
// launch kernel
   // perform MatrixMul on Device
// retrieve results

}
```

matrix multiply example

GPU memory initialization

```
int main()
{
float *A, *B, *C;  int dim = 512;
A = new float(dim*dim); B = new float(dim*dim);
   C = new float(dim*dim);
// I/O to load A and B .
   MatrixMulHost( A, B, C, dim);
}

void  MatrixMulHost (float *A, float *B,  float *C, int  dim)
{
float  a, b, sum;

   for (int i = 0;  i < dim;  i++)
   {
      for (int j = 0;  j < dim;  j++)
      {
         sum = 0;
         for (int k = 0;  k < dim;  k++)
         {
            a        = A[ i  * dim + k ];
            b        = B[ k * dim + j  ];
            sum + = a * b;
         }
         C[ i * dim + j ] = sum;
      }
   }
}
```

```
int main(void)
{
float *A, *B, *C;  int dim = 512;
A = new float(dim*dim); B = new float(dim*dim);
   C = new float(dim*dim);
// I/O to load A and B .
// define thread hierarchy
int  nblocks     = 4;
int  tpb         = 512;
// allocate device memory
size_t  memSize;
   memSize = dim * dim * sizeof(float);
   cudaMalloc( (void**) &d_A, memSize);
   cudaMalloc( (void**) &d_B, memSize);
   cudaMalloc( (void**) &d_C, memSize);

// initialize device memory
   cudaMemcpy(d_A, A, memSize, cudaMemcpy HostToDevice);
   cudaMemcpy(d_B, B, memSize, cudaMemcpy HostToDevice);

// launch kernel
  // perform MatrixMul on Device
// retrieve results

}
```

matrix multiply example

GPU kernel launch

```
int main()
{
float *A, *B, *C;  int dim = 512;
A = new float(dim*dim); B = new float(dim*dim);
   C = new float(dim*dim);
// I/O to load A and B .
   MatrixMulHost( A, B, C, dim);
}

void  MatrixMulHost (float *A, float *B,  float *C, int  dim)
{
float  a, b, sum;

   for (int i = 0;  i < dim;  i++)
   {
      for (int j = 0;  j < dim;  j++)
      {
         sum = 0;
         for (int k = 0;  k < dim;  k++)
         {
            a        = A[ i  * dim + k ];
            b        = B[ k * dim + j  ];
            sum + = a * b;
         }
         C[ i * dim + j ] = sum;
      }
   }
}
```

```
int main(void)
{
float *A, *B, *C;  int dim = 512;
A = new float(dim*dim); B = new float(dim*dim);
   C = new float(dim*dim);
// I/O to load A and B .
// define thread hierarchy
int  nblocks      = 4;
int  tpb          = 512;
// allocate device memory
size_t  memSize;
   memSize = dim * dim * sizeof(float);
   cudaMalloc( (void**) &d_A, memSize);
   cudaMalloc( (void**) &d_B, memSize);
   cudaMalloc( (void**) &d_C, memSize);
// initialize device memory
   cudaMemcpy(d_A, A, memSize, cudaMemcpy HostToDevice);
   cudaMemcpy(d_B, B, memSize, cudaMemcpy HostToDevice);

// launch kernel
   dim3  dimGrid(nblocks);
   dim3  dimBlock(tpb);
   MatrixMulDevice<<< dimGrid, dimBlock >>>(d_A, d_B, d_C, dim);
   // perform MatrixMul on Device

// retrieve results

}
```

matrix multiply example

fetch results

```
int main()
{
float *A, *B, *C;  int dim = 512;
A = new float(dim*dim); B = new float(dim*dim);
   C = new float(dim*dim);
// I/O to load A and B .
   MatrixMulHost( A, B, C, dim);
}

void  MatrixMulHost (float *A, float *B,  float *C, int  dim)
{
float  a, b, sum;

   for (int i = 0;  i < dim;  i++)
   {
      for (int j = 0;  j < dim;  j++)
      {
         sum = 0;
         for (int k = 0;  k < dim;  k++)
         {
            a        = A[ i  * dim + k ];
            b        = B[ k * dim + j  ];
            sum + = a * b;
         }
         C[ i * dim + j ] = sum;
      }
   }
}
```

```
int main(void)
{
float *A, *B, *C;  int dim = 512;
A = new float(dim*dim); B = new float(dim*dim);
   C = new float(dim*dim);
// I/O to load A and B .
// define thread hierarchy
int  nblocks      = 4;
int  tpb             = 512;
// allocate device memory
size_t  memSize;
   memSize = dim * dim * sizeof(float);
   cudaMalloc( (void**) &d_A, memSize);
   cudaMalloc( (void**) &d_B, memSize);
   cudaMalloc( (void**) &d_C, memSize);
// initialize device memory
   cudaMemcpy(d_A, A, memSize, cudaMemcpy HostToDevice);
   cudaMemcpy(d_B, B, memSize, cudaMemcpy HostToDevice);

// launch kernel
   dim3  dimGrid(nblocks);
   dim3  dimBlock(tpb);
   MatrixMulDevice<<< dimGrid, dimBlock >>>(d_A, d_B, d_C, dim);
   // perform MatrixMul on Device

// retrieve results
   cudaMemcpy( C, d_C, memSize, cudaMemcpyDevice To Host);
}
```

it's your turn: matrix multiply example

kernel

```
void  MatrixMulHost (float *A, float *B,  float *C, int  dim)
{
float  a, b, sum;

   for (int i = 0;  i < dim;  i++)
   {
      for (int j = 0;  j < dim;  j++)
      {
         sum = 0;
         for (int k = 0;  k < dim;  k++)
         {
            a        = A[ i  * dim + k ];
            b        = B[ k * dim + j  ];
            sum + = a * b;
         }
         C[ i * dim + j ] = sum;
      }
   }
}
```

```
__global__ void  MatrixMulDevice( float *A, float *B, float *C))
{
// perform MatrixMul on Device

               provide the kernel

}
```

```
int main(void)
{
float *A, *B, *C;  int dim = 512;
A = new float(dim*dim); B = new float(dim*dim);
   C = new float(dim*dim);
// I/O to load A and B .
// define thread hierarchy
int  nblocks      = 4;
int  tpb          = 512;
// allocate device memory
size_t  memSize;
   memSize = dim * dim * sizeof(float);
   cudaMalloc( (void**) &d_A, memSize);
   cudaMalloc( (void**) &d_B, memSize);
   cudaMalloc( (void**) &d_C, memSize);
// initialize device memory
   cudaMemcpy(d_A, A, memSize, cudaMemcpy HostToDevice);
   cudaMemcpy(d_B, B, memSize, cudaMemcpy HostToDevice);

// launch kernel
   dim3  dimGrid(nblocks);
   dim3  dimBlock(tpb);
// perform MatrixMul on Device
   MatrixMulDevice<<< dimGrid, dimBlock >>>(d_A, d_B, d_C, dim);

// retrieve results
   cudaMemcpy( C, d_C, memSize, cudaMemcpyDevice To Host);
}
```

matrix multiply example

IYT answer

engineering cuda kernels

limits vary according to Nvidia "compute capability" level

- do not exceed thread/block limit (512 or 1024)
- do not exceed thread/block dimensionality (512,512,64 or 1024,1024,64)
- do not exceed total registers (8k, 16k, 32k or 64k)
- do not exceed block shared memory (16kB or 48kB)

top performance determined by benchmarking

- depends upon specific GPU hardware
- depends upon application characteristics
- threads execute in 32-thread warps
- typical sweet spot in the 128 – 512 threads/block range

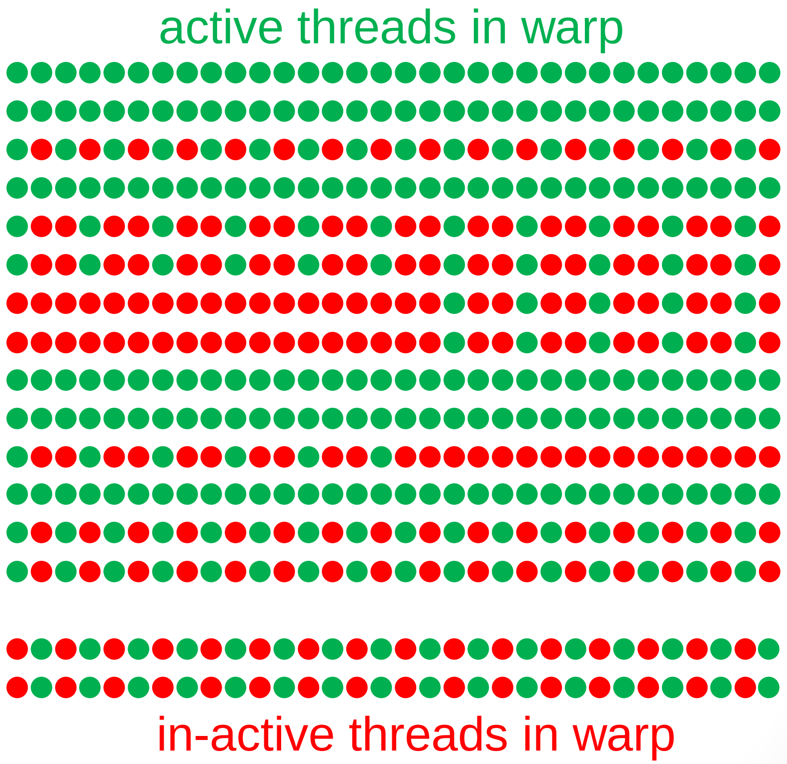
thread “divergence”

```
//  kernel – do various thigs to array A

__global__ void doLotsOfStuff( float *A)
{
int  tid;

    tid        =  blockIdx.x * blockDim.x + threadIdx.x;
    if ( tid % 2 == 0 )
        A[tid] = 42.0;
    if ( tid % 3 == 0 )
        A[tid] = 2.0;
        if ( threadIdx.x > ( th_per_warp / 2) )
            A[tid] = pow(A[tid], 3);
            A[tid] = A[tid] / A[tid-1] + A[tid-2];
    A[tid] - - ;
    if ( A[tid] <= 2 )
        A[tid] = abs(A[tid]) + 1;
    if ( tid % 2 == 0 )
       A[tid] = A[tid] + 12.0;
       A[tid] = A[tid] * PI;
    else
       A[tid] = abs(A[tid] - 12.0);
       A[tid] = A[tid] / PI;

}
```



SIMD serialization

arbitrarily-ordered data

blue	heavy
blue	light
green	light
red	heavy
blue	medium
blue	light
green	light
red	heavy
blue	heavy
blue	heavy
green	medium
red	medium
blue	light
red	light
green	light
green	medium
red	medium
. . .	

naturally lends itself to a branching structure ...

```
if blue
    if heavy
        do blue-heavy work
    else if medium
        do blue-medium work
    else if light
        do blue-light work
else if green
    if heavy
        do green-heavy work
    else if medium
        do green-medium work
    else if light
        do green-light work
else if red
    . . .
```

SIMD serialization

SIMD-ordered data

blue	heavy
blue	heavy
blue	heavy
. . .	
blue	medium
blue	medium
blue	medium
. . .	
blue	light
blue	light
blue	light
. . .	
green	heavy
green	heavy
green	heavy
. . .	
green	medium
. . .	

naturally lends itself to a branching structure ...

```
while blue heavy
        do blue-heavy work
while blue medium
        do blue-medium work
while blue light
        do blue-light work
while green heavy
        do green-heavy work
while green medium
        do green-medium work
while green light
        do green-light work
while red heavy
        do red-heavy work
while red medium
        . . .
```

giga flops (GFlops) benchmarking

given that all variables are single- or double-precision floating point numbers,

the following statements perform "1 flop" worth of work:

- A = B + C
- A[k] = A[j] * A[i]
- A[k-1] += B[m]
- A[k] = B[m] / C[n]

while the following would be "3 flops" worth of work:

- A[m] = (B[m] + C[m]) / (D[m] + E[m])

to compute the performance of a code segment:

- count the number of flops performed in the segment
- divide by the execution time in seconds

results are typically reported in "GFlops/sec"

cuda function types

__global__ void kernel callable from non-cuda function

example:

```
        __global__   void   CellLoop1(struct cuda_params *p,  float *anbs,  float *aps,  float *resid,  float *scs)
```

__host__ void function (non-kernel) using cuda functions callable from non-cuda function

example:

```
        __host__ void    disord_simd(int tindex,  int oiter)
        //useful for including operations like ...
                cudaMemcpy(anbs,  d_anbs,  ncells*nf_max*sizeof(float),  cudaMemcpyDeviceToHost);
```

__device__ float function callable from a kernel

example:

```
        __device__ float  d_iwalls(int pol,  int band,  float degrees,  float delta_la,  int nla,  float *xi,  float *wi);
```

cuda functions reside in a ".cu" program file

debugging: gpuAssert()

host-side debugging

a helpful macro ...

```
#define gpuErrchk(ans) { gpuAssert((ans), __FILE__, __LINE__); }
inline void gpuAssert(cudaError_t code, const char *file, int line, bool abort=true)
{
   if (code != cudaSuccess)
   {
      fprintf(stderr,"GPUassert: %s %s %d\n", cudaGetErrorString(code), file, line);
      if (abort) exit(code);
   }
}
```

verify copy successful

```
cudaMemcpy(r,    d_resid,ncells*sizeof(float), cudaMemcpyDeviceToHost);
gpuErrchk( cudaMemcpy(r, d_resid,ncells*sizeof(float), cudaMemcpyDeviceToHost) );
```

ensure kernel ran to completion

```
CellLoop3<<<dimGrid, dimBlock>>>(d_p, d_anbs, d_aps, d_intensity, d_resid, d_scs);
gpuErrchk( cudaPeekAtLastError() );            // stop if kernel did not launch
gpuErrchk( cudaDeviceSynchronize() );          // for debugging – ensure kernel finished
```

debugging: printf()

device-side debugging

"printf" from within a kernel is supported on selected cuda implementations

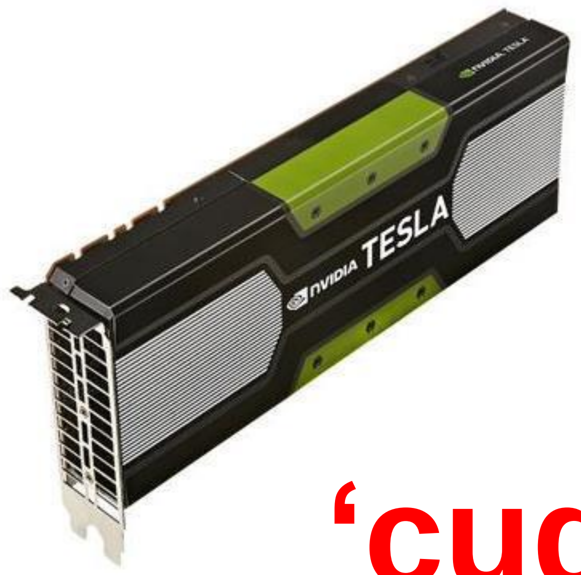
- must have compute capability of 2.0 or higher
- must have cuda developer version 3.1 or higher
- output is buffered and returned to host at kernel completion
- output buffer overflow is lost
- host program must run long enough for output buffers to transfer and flush (think seg fault behavior)

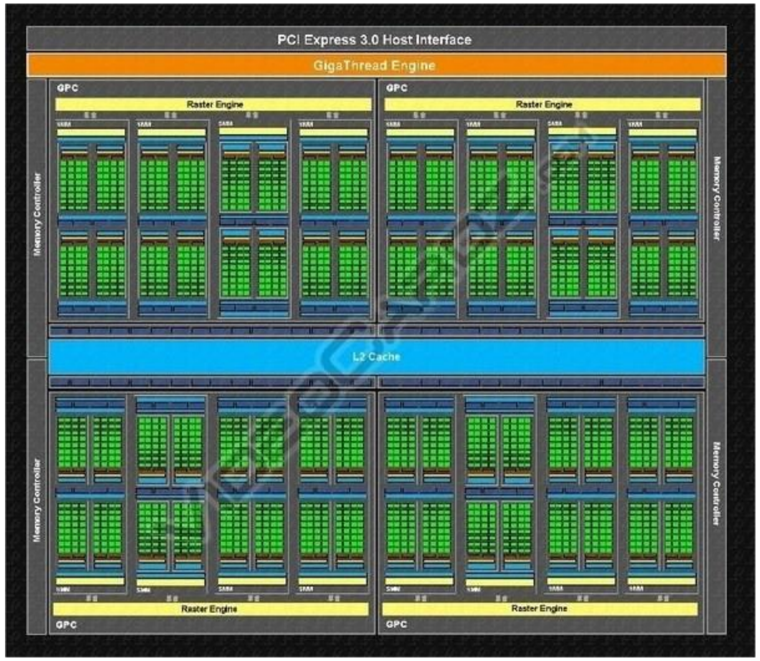
debugging

- cudagrind
- cuda-gdb

cse5441 - parallel computing

https://cdn8.bigcommerce.com/s-o1ibh36yyf/images/stencil/1280x1280/products/18124/29821/hpgraphiccardsvideocardstesla__52934.1516995900.jpg?c=2&imbypass=on
https://i0.wp.com/www.visitcocosisland.com/wp-content/uploads/2017/01/animalia-life-club.jpg?resize=640%2C492
https://qph.fs.quoracdn.net/main-qimg-257ea3b2d296c1605a50e5fd5db21cfa-c
https://pictures.topspeed.com/IMG/jpg/201205/plymouth-hemi-cuda-p.jpg





‘cuda