cse5441 - parallel computing

 $https://cdn8.bigcommerce.com/s-o1ibh36yyf/images/stencil/1280x1280/products/18124/29821/hpgraphiccardsvideocardstesla_52934.1516995900.jpg?c=2\&imbypass=onality. The products of the product of the pro$

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-chttps://pictures.topspeed.com/IMG/jpg/201205/plymouth-hemi-cuda-p.jpg









© Copyright. Jeffrey S. Jones, 2018. May be used for educational purposes without written permission but with a citation to this source. May not be distributed electronically without written permission.

1

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

2

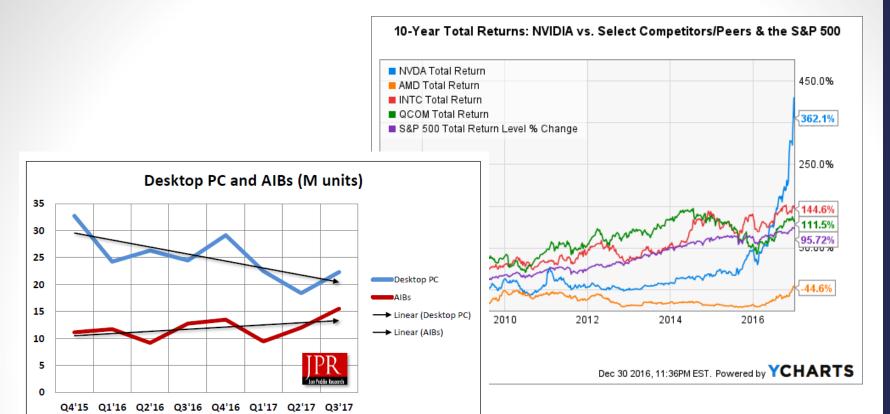
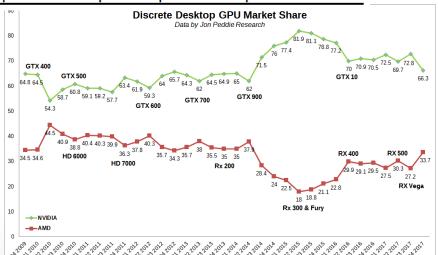


Figure 3: Shipments of desktop PCs compared to desktop AIBs over time



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

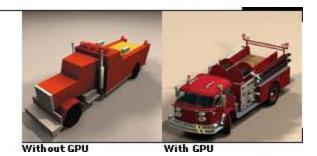
ADF

Amber Savant ArrayFire Semcad-X AxRecon TeraChem COSMO Theano

Desmond Torch7

Tsunami RTM **Empro**

UGENE Fluent Optistruct Vega ZZ **RADIOSS** 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

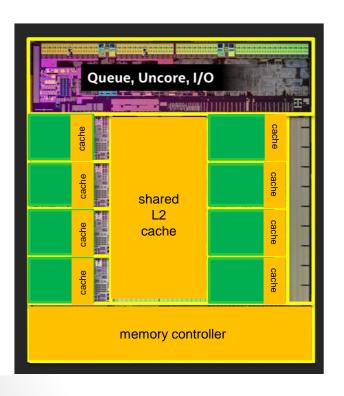
what is a gpu?



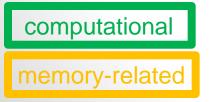
5

gpu / cpu comparison

"CPU" GPU







6

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

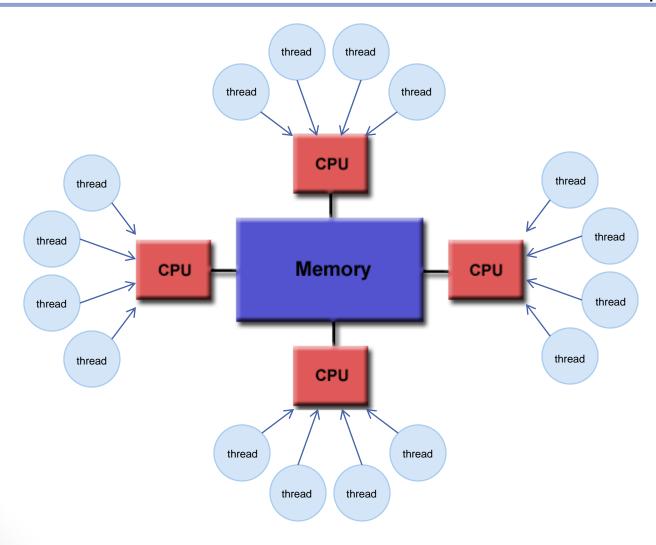
- rigorous analysis of variable scope and sharing
- device / host processing
- non-continguous, nonhomogenous memory
- hierarchical, synchronous thread teams

OSU CSE 5441

8

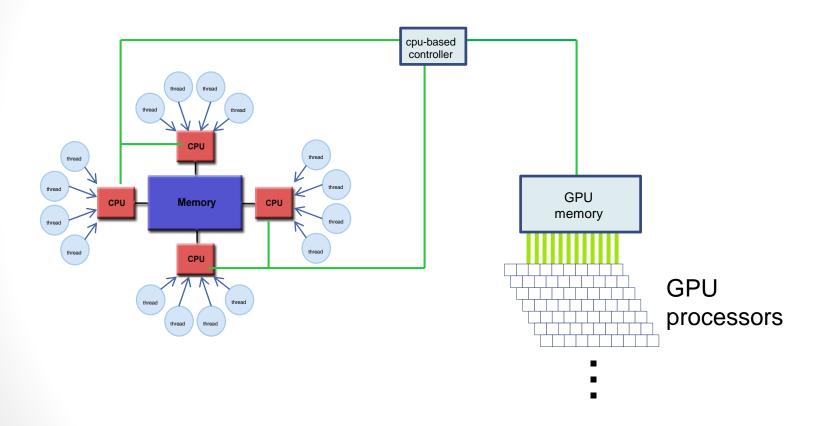
cpu global shared memory

review



9

cpu/gpu non-contiguous memory



-"memory speed"

"parallel-access memory speed"

OSU CSE 5441

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

10

JSU CSE 5441

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

unit of kernel scheduling, a group of threads warp

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

JSU CSE 5441

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:

13

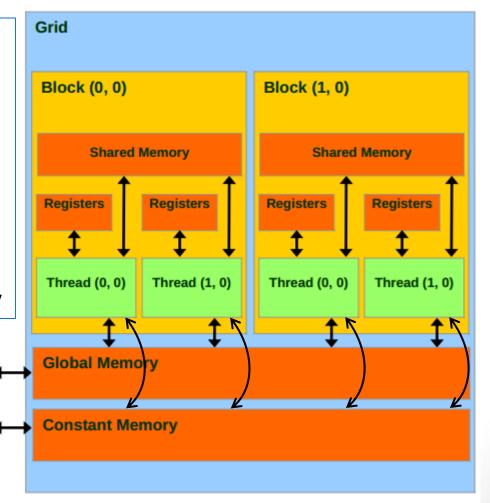
cuda memory transfers

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

Host



14

cuda memory transfers

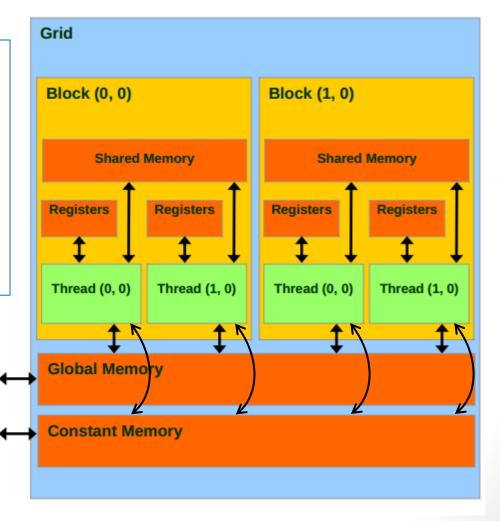
host side

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

Host



15

cuda memory transfers - allocation

cuda memory transfers - copy

cuda memory transfers - retrieve

```
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);
             = (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

```
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
                          //pointer for device memory
int *d_a;
// define thread hierarchy
int num blocks
int num th per blk = 8;
// allocate host and device memory
size t memSize;
  memSize = num_blocks * num_th_per_blk * sizeof(int);
            = (int*) malloc(memSize);
  h a
  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);
```

to the linearized thread_id

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

it's your turn ... reverseArray_singleBlock

```
// initialize an array using GPU
int main()
int *h a;
                          //pointer for host memory
                          //pointer for device memory
int *d_a;
// define thread hierarchy
int num blocks
int num th per blk = 8;
// allocate host and device memory
size t memSize;
  memSize = num blocks * num th per blk * sizeof(int);
            = (int*) malloc(memSize):
  h a
  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

 $https://cdn8.bigcommerce.com/s-o1ibh36yyf/images/stencii/1280x1280/products/18124/29821/hpgraphiccardsvideocardstesla_52934.1516995900.jpg?c=28imbypass=onality. The products of the product of the pro$

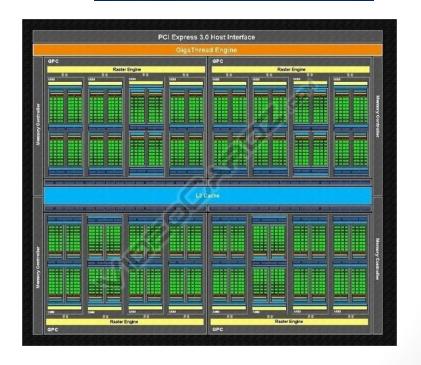
https://i0.wp.com/www.visitcocosisland.com/wp-content/uploads/2017/01/animalia-life-club.jpg?resize=640%2C492

https://qph.fs.quoracdn.net/main-qimg-257ea3b2d296c1605a50e5fd5db21cfa-chttps://pictures.topspeed.com/IMG/jpg/201205/plymouth-hemi-cuda-p.jpg









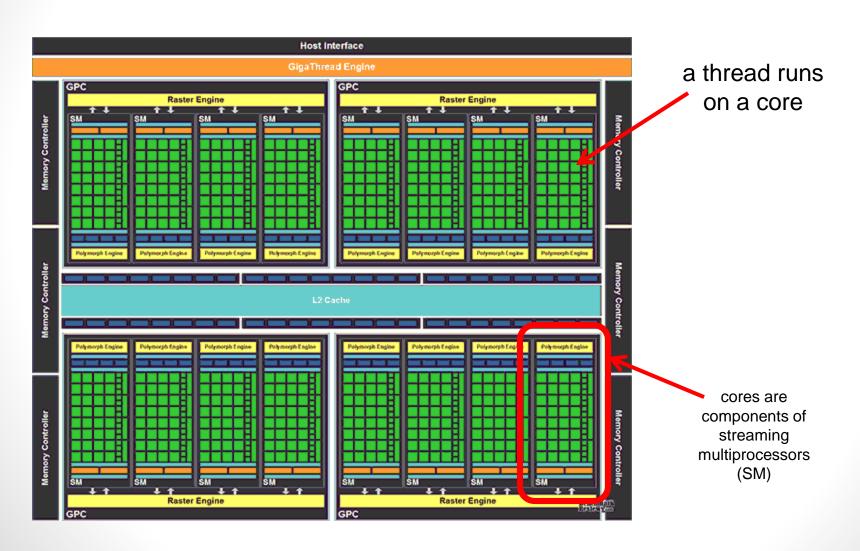
© Copyright. Jeffrey S. Jones, 2018. May be used for educational purposes without written permission but with a citation to this source. May not be distributed electronically without written permission.

24

nVidia GPU compute capability

DEVELONER I	NAME AND THE PARTY OF THE PARTY						
	WOA FOOLN'T DOCUMENTATION						
CONTRACTOR CO.	H. Compute Capabilities						
Diagramming Suite of Colombatiles	The general specifications and below as all a compute stocker depend on its compute copability (see <u>Compute Constitu</u> s).						
12. Drag arreing Health	No. 12 gives the features and technical specifications associated to each compute supplietly						
12. Degraming Market	Seek for Chink 12 contents or three complement with the 1888 Seek impossing all analogs.						
F.S. Handrick of Implementation F.S. Der terminals Dubbellen.	Section <u>County County Co</u> you was detailed	sis on the craft order of derivation of compute suggesting E.a. E.a. and F.a. respectively					
A CORPORATE STATE	H. J. Paul area, and Technical Described Law.						
*E.Compage biomies		Sales C. Paul on Engard per Compute Capability					
FC Depositio Drags	Periors Segret	Compute Capabilly					
10 C/O/Openie Resistan	(Add at helicine are suggested for all surplus suppliffs in)	10 13 15,17,16,13 13 4s 7s					
i E. Halfematikal Functions in F.O.C., Language Engage 6	Bank testion operating at 20th history rates in global memory (<u>Bank Foreigns</u>)	*					
+ D. Technic Policiting	at an influsir) against ing an 1246 flooting point values in global mannery (<u>at an influsion</u>)	*					
Q.A. Visconi Onto Language	Bank batter speaking at 246 https://das.insharet.nempy (<u>Bank Postlan</u>)	-					
0.3 User Filtering	standard operating at 200 facility politicatus. In shared memory (<u>standard</u>)	~					
0.5 Take Leeing	Stants brokken agerating or 64% integer values in global memory (<u>Stants Functions</u>)	~					
H. Grepte Ophilities -	Stanta Couldes agreeding an 8448 integer value. In shared memory (<u>Gentle Functions</u>)	~					
J. C. C. Surbenned Variables	Stanta still be againsting as 2000 facility point values in global and shared memory (<u>all amission</u>)						
E. Unified Hornery Drags genericy	Stants shift in age of the middle facility paid value. In global memory and shared memory (a) articles.)	~					
Separate	Viegosia ani latisi. Gratien (<u>Vientiala Fontian</u>)						
	_invalencypten(time: hou houling)						
	_spectrosit_modify						
	pyratir mak_prail;						
	_specific make_or () (Londo point in Facul into)	_					
							
	Sorties Sentites (<u>Sorties Footies</u>)	─					
	To big the continues.						
	Collectionary Programming						
	Furni shili (ser referense mensel)	-					
	Synamic Resiliellum	**					
	Halfy white facility point age of less with less said reality, multiplication, samperises, sur y shalle facilities, see						
	Transier	•					
		Table 14 Technical Specifical laws gar Compute Cogniting Compute Cogniting					
	Terinian Specifications	20 22 22 22 22 22 22 22 22 22 22 22 22 2					
	National series of recition grids per denies <u>General ed Seriel Bered in</u> d						
	Hashwar dimensionality of grid of throad blacks	1					
	Maximum undimension of a grid of the east blacks.	\$ ⁵⁴ 4					
	Hastman prior and mension of a grid of throad blacks.	NOTE:					
	Maximum dimensionality of thread Maxim	:					
	Maximum or or primers in reflections	-					
	Nazimum artimension of a block	64					
	Nation writer of threats per black	ğ					
	Hirgolae	=					
	Nation write alreaded black primally season						
	Nation write alreaded varyage multipleasure	¥ =					
	Nation write alreaded threat principle season	204 (2)					
	Number of 2041 registers per multipressure	ers cars ers					
	Sacroum number of 2046 registers per thread black	위도 교로 위도 의도 위도 교도 위도					
	Nation series of 1246 registers per thread	KE 122					
	Sacinar areas of the of meney ger multipressure	400 MB MB MB MB MB MB MB					
		AG NG NG					
	Maximum amount of shared memory gar throad black 🚾	48 98 98					
		26 NG HB					
	Nations are not of the element per broad limb : Surface of the element parts Joseph of the element per broad						
	Number of the set memory basis Americal Stand memory per 1 in set Constant memory site	5 204 84					
	Survice of shared memory banks. Invasid of least memory yet the real	20 20 H					
	Number of the set memory basis Americal Stand memory per 1 in set Constant memory site	5 204 84					
	Standar of the est memory basis. Jenus of stand memory gar bit read. Description memory that Description memory that Description of gar multi-processor for constant memory.	204					
	close of Arbeit American pairs. The of Arbeit American pairs from a Section of Arbeit American pairs from a Section of Arbeit American pairs from a Section on the pairs and parameter for annual and memory. Section on the pairs and parameter for the Arbeit American Section on the Section of Arbeit American pairs and Section on the Section of Arbeit American pairs and Section of Arbeit American pairs and Section Arbeit Americ						
	Totale of his of servey both. These of distance yets to each Standard revery yets Sta						
	Colore of Arther of Armery (Man.) Description of Armery (Man.) D						
	Comment of the eff enemy joins. These of leading recognition are provided in the comment of the						
	Colore of Arther of Armery (Man.) Description of Armery (Man.) D	20 204 204 204 204 204 204 204 204 204 2					
	Course of the eff energy betts. The class of bedder energy parts and The class of energy parts and The class of energy parts and The class of energy parts and the class of energy The class of energy parts and the foliation energy The class of energy parts and the foliation energy The class of energy parts and the class of energy The class of energy parts and the energy The class of energy parts a	20 204 204 204 204 204 204 204 204 204 2					
	Institute of facilities the second points. The second facilities recognize the consistent enemy. Desired continguity age and age assess for second enemy. Desire continguity age and age assess for second enemy. Desire continguity age and age assess for second enemy. Desire continguity age and age and age assess for second enemy. Desire continguity age and age and age and age and age and age	1					
	Counter Chief of energy betts. Transit allower areasy part to the control of energy part to the counter of energy counter of the fact of energy counter of the energy counter of the energy fact of energy counter of the energy fact of energy fact of energy fact of energy counter of the energy fact of energy fact of energy fact of energy counter of the energy fact of energy fact of energy fact of energy factors of energy fac	20 204 204 204 204 204 204 204 204 204 2					
	Colonie of Arther American Services (1998). The colonie of Arther American Services (1998) and the Colonie	1					
	Counter of the eff energy betts. The class of better energy to the counter of th	20					
	Interest in the extracting team. The of the extracting part to extract the extraction through the extraction of the ext						
	Counter of the eff energy betts. The classification are proportional. The classification are proportional. The classification are proportional are classification	1988 1988					
	Interest A the efficiency pairs in an interest pairs of the efficiency pairs and pairs of the efficiency pairs and pairs and pairs are pairs and pairs are recognized per analyse search to institute memory. Gets we entire pair and applications to the entire pairs and efficiency of the entire pairs and entire pai	201					
	Counter of the eff energy betts. Transit of both or every part treat. Transit of both or every part treat. Transit of both or every part treat. Transit of transit or every part treat. Transit or every counter or every part treat. Transit or every counter or every part treat or every part to every part to every part treat or every part tre	1988 1988					
	Content of the eff enemy joins. The of the eff enemy join in early content of enemy. Desired when you join in early content to enabled enemy. Carlo enemy join join and join early to enable enemy. Carlo enemy join join and join early to enable enemy. Carlo enemy join join and join early to enable enemy. Carlo enemy join join and join enemy join enemy. Carlo enemy join and join enemy join	201					
	Counter of the eff memory parts. Transit of limits are recognized by an advantage of the counter of the counte	2018 2018 2018 2018 2018 2018 2018 2018					
	Content of the eff enemy joins. The of the eff enemy join in early produced in the effect of enemy joins and produced in the effect of enemy joins. Desire descripting any end joy means for installed enemy. Carlo enemy joins joy multiple enemy for installed enemy. Carlo enemy joins joy multiple enemy for installed and proving joins and produced in the enemy joins and produced in the enemy joins and produced enemy j	201					
	Comment of the eff memory points. The contract of the effective payment of the contract of the effective payment of the contract of the effective payment of the contract of	10					
	Interest in the extracting terms or the control of	2018 1018					
	Comment of the eff memory points. The contract of the effective payment of the contract of the effective payment of the contract of the effective payment of the contract of	10					
	Interest in the extracting terms or the control of	2018 1018					

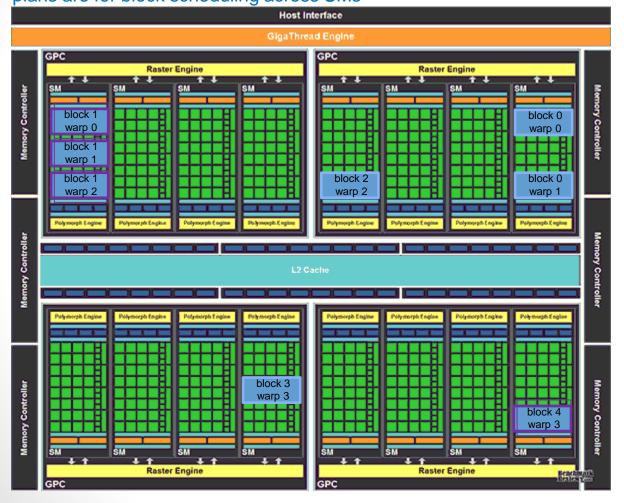
"massively parallel" thread organization: threads



26

"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)

27

"massively parallel" thread organization: grids

warps from 2 different grids running simultaneously on a single GPU

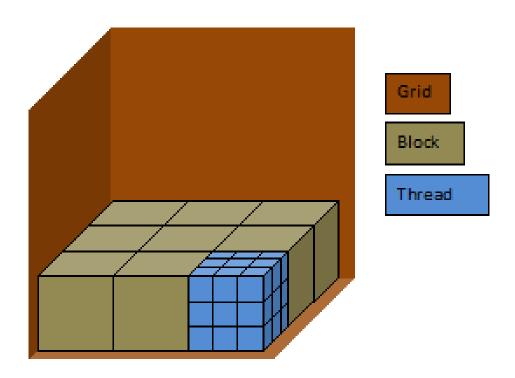


a group of blocks is a **grid**

> each kernel gets one grid of threads

a **GPU** may schedule multiple simultaneous grids

"massively parallel" thread organization



29

OSU CSE 5441

"massively parallel" thread organization:

threads in blocks, 1 block

1D: threadldx.x

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

2D: threadldx.x * threadldx.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: threadldx.x * threadldx.y * threadldx.z

```
0,0,2
                       0,0,3
                               0,0,4
                                      0.0.5
0,0,0
       0,0,1
                                              0,0,6
                                                      0,0,7
               0,1,2
                       0,1,3
                               0,1,4
                                      0,1,5
                                              0,1,6
                                                      0,1,7
0,1,0
       0,1,1
0.2.0
       0.2.1
               0.2.2
                       0.2.3
                              0.2.4
                                      0.2.5
                                              0,2,6
                                                      0.2.7
                         1,2,3
```

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

```
1-dimensional:
```

```
dim3 dimBlock(n_th_per_blk);
blk_t_id = threadldx.x
```

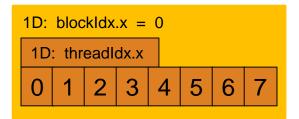
2-dimensional:

3-dimensional;

OSU CSE 5441

"massively parallel" thread organization:

threads in blocks, 1D x 1D



1D: blockldx.x = 1

1D: threadIdx.x

0 1 2 3 4 5 6 7

```
1D: blockldx.x = 2

1D: threadIdx.x

0 1 2 3 4 5 6 7
```

```
blockldx.x
block position in dimension x
blockldx.y
block position in dimension y
blockldx.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
```

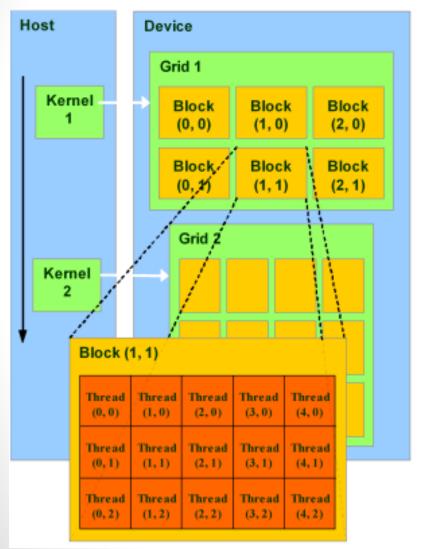
threadsPerBlock = blockDim.x threadNumInBlock = threadIdx.x

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

31

"massively parallel" thread organization:

threads in blocks, 2D x 2D



```
blockldx.x
block position in dimension x
blockldx.y
block position in dimension y
```

// if we had a third dimension
(blockldx.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
```

note: this illustration in column-major

32

it's your turn: reverseArray_multiBlock1

```
// reverse an array using GPU
int main()
int *h a;
                             //pointer for host memory
                             //pointer verification copy
int *h v;
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);
            = (int*) malloc(memSize);
  h a
            = (int*) malloc(memSize);
  h v
  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
```

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

33

reverseArrayMultiBlock1

IYT answer

it's your turn: reverseArray_Block2

```
// reverse an array using GPU
int main()
int *h a;
                             //pointer for host memory
                             //pointer verification copy
int *h v;
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);
            = (int*) malloc(memSize);
  h a
            = (int*) malloc(memSize);
  hν
  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
```

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

35

reverseArrayMultiBlock2

IYT answer

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[i * dim + k];
                 = B[k*dim + j];
          sum + = a * b;
       C[i * dim + j] = sum;
```

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[i * dim + k];
                 = 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
}
```

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[i * dim + k];
                = 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
}
```

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[i * dim + k];
                 = 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
}
```

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[i * dim + k];
                = 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;
              = 512;
int tpb
// 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
```

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[i * dim + k];
                 = 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
```

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[i * dim + k];
          а
                 = 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
```

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[i * dim + k];
          а
                 = 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;
    }
}</pre>
```

```
__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);
```

IYT answer

engineering cuda kernels

limits vary according to Nvidia "compute capability" level

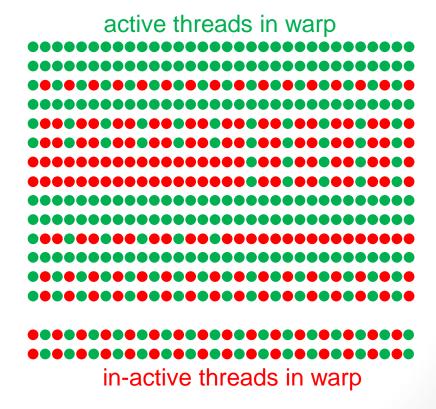
- do not exceed thread/block limit (512 or 1024)
- do not exceed thread/block dimensionality (512,512,64 or 1024,1024,64)
- do not exceed total registers (8k, 16k, 32k or 64k)
- do not exceed block shared memory (16kB or 48kB)

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
           = blockldx.x * blockDim.x + threadldx.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] \le 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;
```



JSU CSE 5441

SIMD serialization

arbitrarily-ordered data

blue heavy blue light light green red heavy medium blue blue light light green red heavy blue heavy blue heavy medium green red medium blue light light red light green medium green 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
```

JSU CSE 5441

SIMD serialization

SIMD-ordered data

blue	heavy
blue	heavy
blue	heavy
blue	mediun

blue medium blue medium blue medium

. . .

blue light blue light blue light

. . .

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

. . .

JSU CSE 5441

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

```
__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

 $https://cdn8.bigcommerce.com/s-o1ibh36yyf/images/stencil/1280x1280/products/18124/29821/hpgraphiccardsvideocardstesla_52934.1516995900.jpg?c=2\&imbypass=onality. The products of the product of the pro$

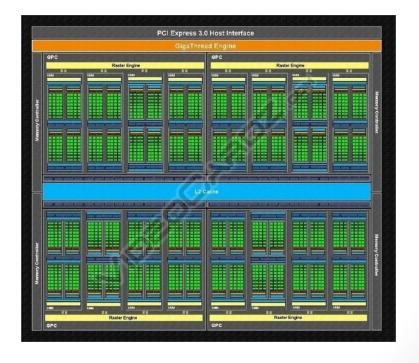
https://i0.wp.com/www.visitcocosisland.com/wp-content/uploads/2017/01/animalia-life-club.jpg?resize=640%2C492

https://qph.fs.quoracdn.net/main-qimg-257ea3b2d296c1605a50e5fd5db21cfa-chttps://pictures.topspeed.com/IMG/jpg/201205/plymouth-hemi-cuda-p.jpg









© Copyright. Jeffrey S. Jones, 2018. May be used for educational purposes without written permission but with a citation to this source. May not be distributed electronically without written permission.

56

J. S. Jones