GPU CUDA Programming

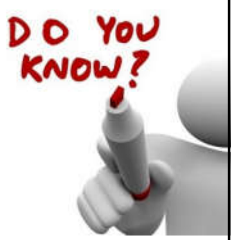
이 정 근 (Jeong-Gun Lee)

한림대학교 소프트웨어융합대학, 임베디드 SoC 연구실

www.onchip.net

Email: **Jeonggun.Lee@hallym.ac.kr**

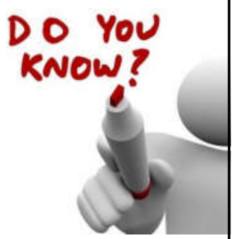
차례



- **Introduction**
  - Multicore/Manycore and GPU
  - GPU on Medical Applications
- **Parallel Programming on GPUs: Basics**
  - Conceptual Introduction
- **GPU Architecture Review**
- **Parallel Programming on GPUs: Practice**
  - Real programming
- **Conclusion**

차례

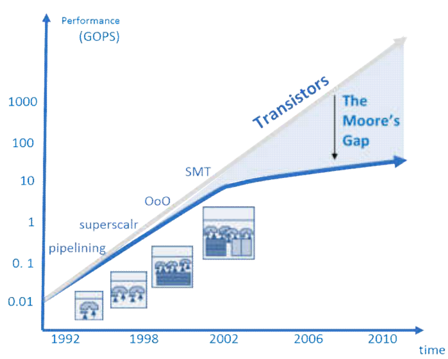
- **Introduction**
  - Multicore/Manycore and GPU
  - GPU on Medical Applications
- **Parallel Programming on GPUs: Basics**
  - Conceptual Introduction
- **GPU Architecture Review**
- **Parallel Programming on GPUs: Practice**
  - Real programming
- **Conclusion**

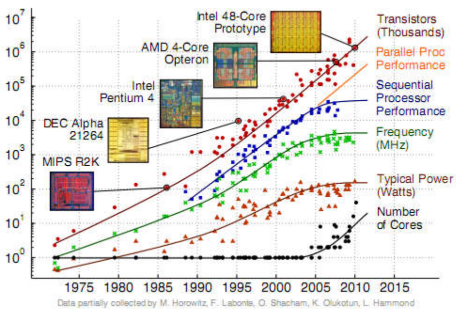


Introduction

- Multicore/Manycore and GPU

> A **multi-core** CPU (or chip-level multiprocessor, CMP) combines two or more independent cores into a single package composed of a single integrated circuit (IC), called a die, or more dies packaged together.
>
> \- Wikipedia





Prepared by C. Batten - School of Electrical and Computer Engineering - Cornell University - 2005 - retrieved Dec 12 2012 - http://www.csl.cornell.edu/courses/ece5950/handouts/ece5950-overview.pdf

Introduction

- Alpha-Go ?

구글의 DeepMind가 이렇게 강력한 인공지능을 개발할 수 있었던 바탕은 풍부한 계산자원에 있다. 판후이와 대국을 한 분산 AlphaGo의 경우 **1202개의 CPU와 176개의 GPU**가 사용됐다.

Deep learning is **HOT** !

Why now ?

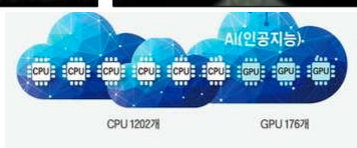
Introduction

- Alpha-Go ?

구글의 DeepMind가 이렇게 강력한

The Big Bang in Machine Learning

Why now ?



Introduction: GPU ? Manycore ?

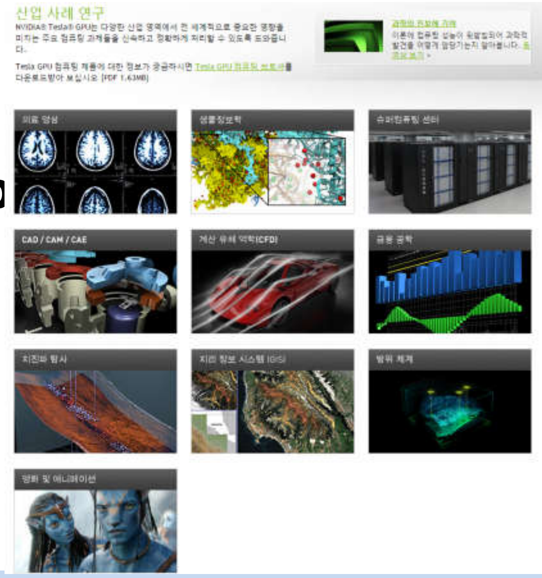
Introduction: GPU ? Manycore ?

GPU ?

Graphics only Processing Unit ?

Introduction: GPU ? Manycore ?

Grap



nit ?

Introduction: GPU ? Manycore ?

Grap

nit ?

GPGPU

General Purpose GPU

http://www.nvidia.co.kr/object/tesla-case-studies-kr.html

Introduction: GPU ? Manycore ?

SIMT

GPU = Multi-threaded Vector Processing on Massively Parallel Many Cores

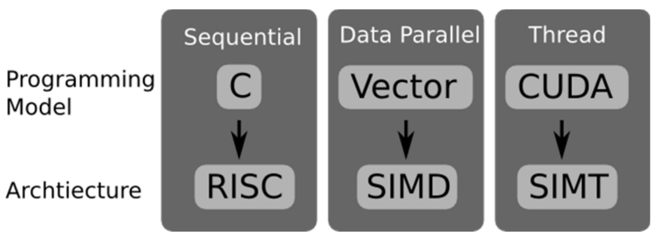
SIMT was introduced by Nvidia:
[The G80 Nvidia GPU architecture, Tesla] introduced the ***single-instruction multiple-thread (SIMT)*** execution model where multiple independent threads execute concurrently using a single instruction.

From https://en.wikipedia.org/wiki/Single_instruction,_multiple_threads

Pixel shaders : Vertex shaders → Unified Shader (GPGPU)

GPU and SIMT ?

- Multicore/Manycore and GPU

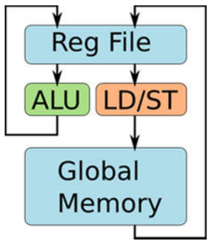


Seq. vs Data Parallel vs Thread

```
int A[2][4];
for(i=0;i<2;i++){
    for(j=0;j<4;j++){
        A[i][j]++;
    }
}
```

Assembly code of inner-loop

lw r0, 4(r1)
addi r0, r0, 1
sw r0, 4(r1)

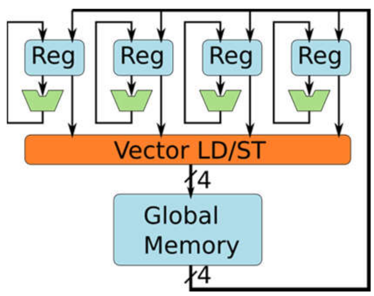


Programmer's view of RISC

참조: http://www.es.ele.tue.nl/~heco/courses/EmbeddedComputerArchitecture/

Most CPUs Have Vector SIMD Units

- Programmer's view of a vector SIMD, e.g. SSE.



참조: http://www.es.ele.tue.nl/~heco/courses/EmbeddedComputerArchitecture/

Let's Program the Vector SIMD

- Unroll inner-loop


```
int A[2][4];
for(i=0;i<2;i++){
    for(j=0;j<4;j++){
        A[i][j]++;
    }
}
```

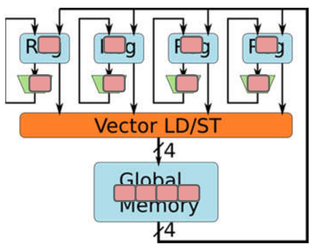
```
int A[2][4];
for(i=0;i<2;i++){
    movups  xmm0,        [ &A[i][0] ]  // load
    addps   xmm0,        xmm1          // add 1
    movups  [ &A[i][0] ], xmm0         // store
}
```

Looks like the previous example,
but each SSE instruction executes on 4 ALUs.

참조:
http://www.es.ele.tue.nl/~heco/courses/EmbeddedComputerArchitecture/

How Do Vector Programs Run?

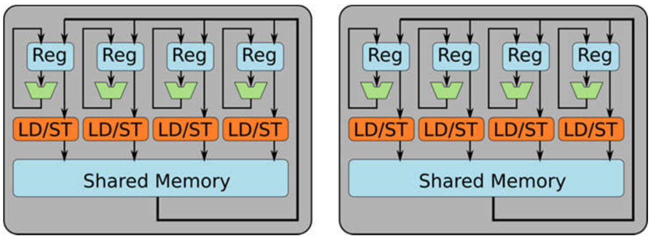
```
int A[2][4];
for(i=0;i<2;i++){
    movups xmm0,         [ &A[i][0] ]  // load
    addps  xmm0,         xmm1          // add 1
    movups [ &A[i][0] ], xmm0          // store
}
```



참조:
http://www.es.ele.tue.nl/~heco/courses/EmbeddedComputerArchitecture/

CUDA Programmer's View of GPUs

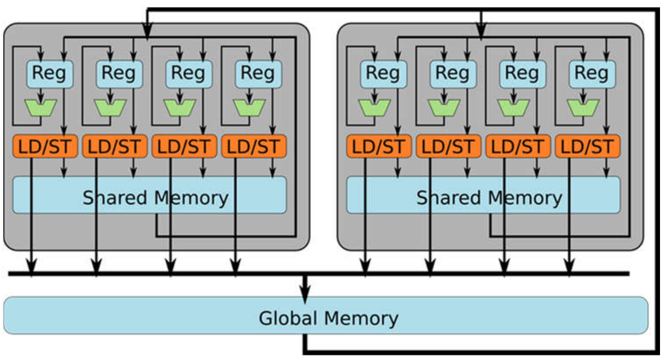
- A GPU contains multiple SIMD Units.



참조:
http://www.es.ele.tue.nl/~heco/courses/EmbeddedComputerArchitecture/

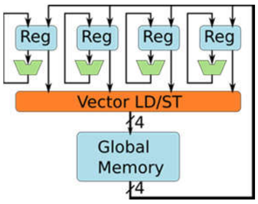
CUDA Programmer's View of GPUs

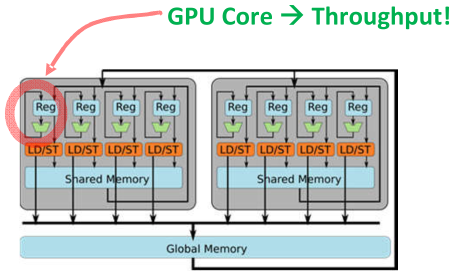
- A GPU contains multiple SIMD Units. All of them can access global memory.



참조:
http://www.es.ele.tue.nl/~heco/courses/EmbeddedComputerArchitecture/

What Are the Differences?





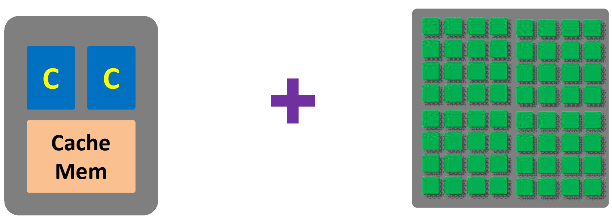
Let's start with two important differences:

1. GPUs use threads instead of vectors
2. GPUs have the "Shared Memory" spaces

참조:
http://www.es.ele.tue.nl/~heco/courses/EmbeddedComputerArchitecture/

CPU vs. GPU

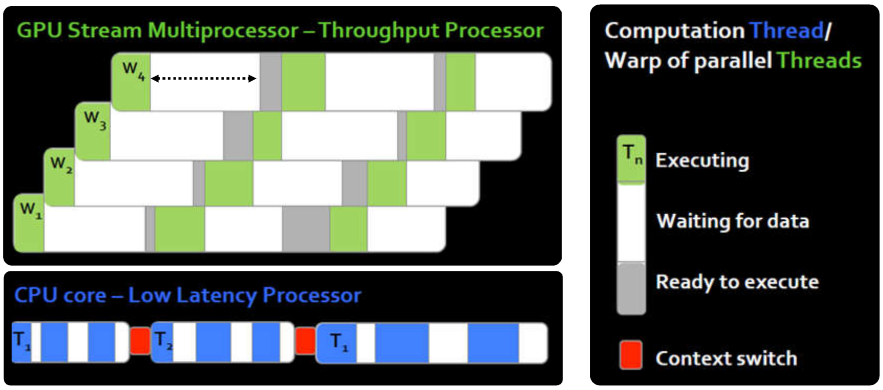
***Latency* Processor + *Throughput* Processor**



Heterogeneous Computing

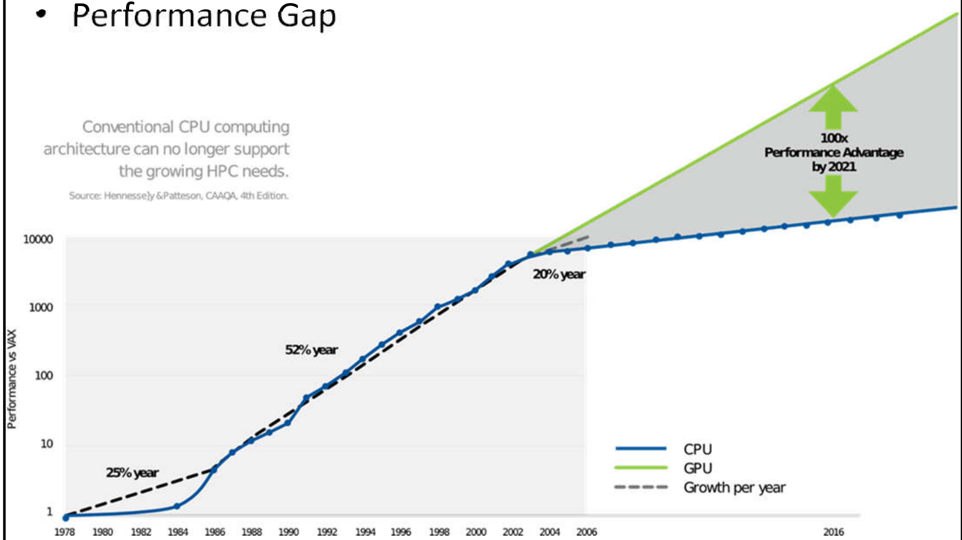
CPU vs. GPU

- CPU architecture attempts to minimize latency within each thread with the help of CACHING
- GPU architecture hides latency with computation from other thread warps with the help of THREADING



Introduction

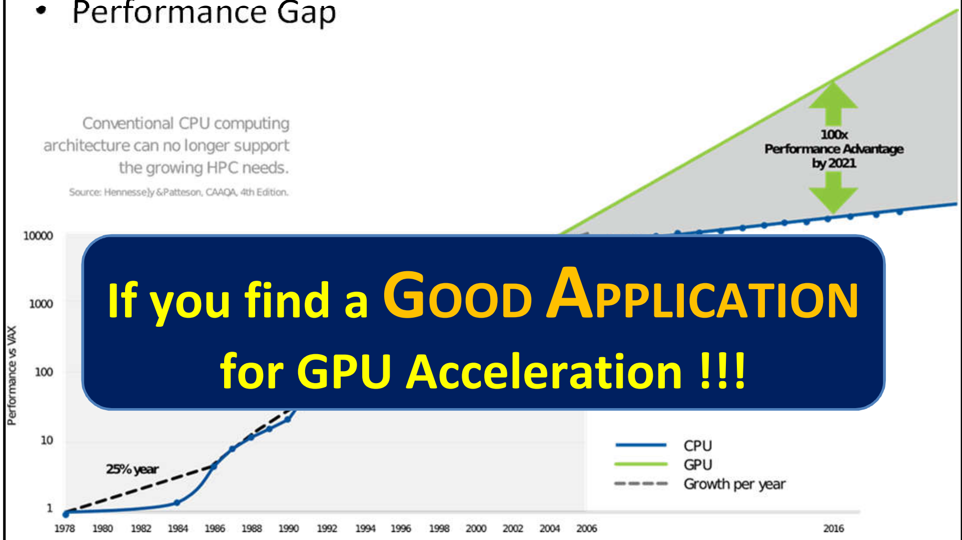
- Performance Gap



Ref: Tesla GPU Computing Brochure

Introduction

- Performance Gap

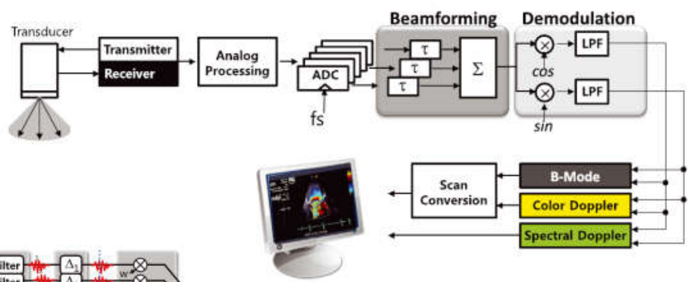


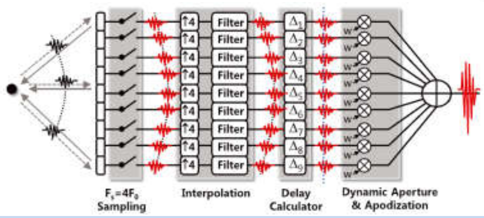
If you find a GOOD APPLICATION for GPU Acceleration !!!

Ref: [Tesla GPU Computing Brochure]

Introduction

- GPU on ***Medical Imaging*** Applications
  - One of the **GOOD GPU APPLICATIONs**

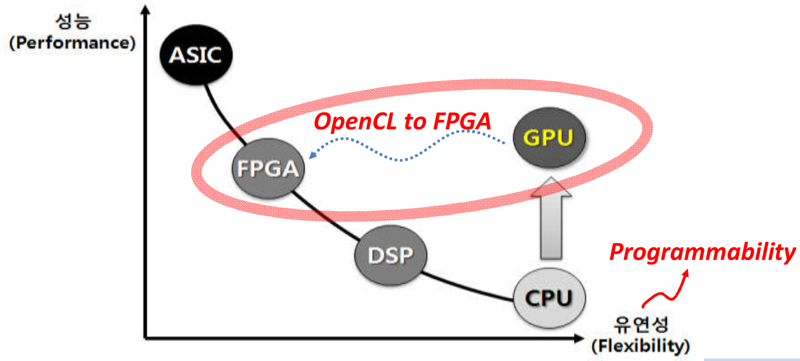




Real-Time
&
Data Level Parallelism

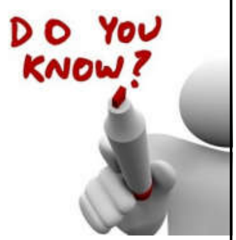
Introduction: ***Programmable!***

- Implementation styles of conventional ***Medical Imaging*** Applications
  - ASIC / FPGA or DSP for high performance real-time computing



차례

- **Introduction**
  - Multicore/Manycore and GPU
  - GPU on Medical Applications
- **Parallel Programming on GPUs: Basics**
  - Conceptual Introduction
- **GPU Architecture Review**
- **Parallel Programming on GPUs: Practice**
  - Real programming
- **Conclusion**



Parallel Programming on a GPU

We have one code part …

```
for(i=0; i < 100; i++)
        C[i] = A[i] + B[i];
```

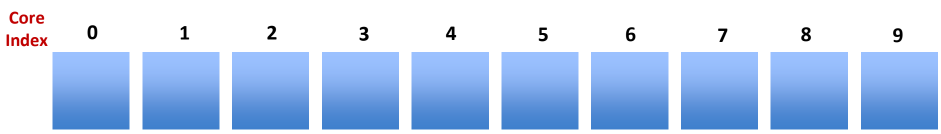
Yes! It is simple !

But what about on ***10 cores*** on-chip processor ?

Parallel Programming on a GPU

But what about on 10 cores on-chip processor ?

```
for(i=0; i < 100; i++)
   C[i] = A[i] + B[i];
```



Parallel Programming on a GPU

But what about on 10 cores on-chip processor ?

```
for(i=0; i < 100; i++)
  C[i] = A[i] + B[i];
```

Core Index

0 1 2 3 4 5 6 7 8 9

For each core

```
for(i=0; i < 10; i++)
  C[    ?    ] = A[    ?    ] + B[    ?    ];
```

Parallel Programming on a GPU

But what about on 10 cores on-chip processor ?

```
for(i=0; i < 100; i++)
  C[i] = A[i] + B[i];
```

Core Index

0 1 2 3 4 5 6 7 8 9

For each core

```
for(i=0; i < 10; i++)
  C[CoreIndex*10+i] = A[CoreIndex*10+i] + B[CoreIndex*10+i];
```

Parallel Programming on a GPU

What about Threading ?

```
for(i=0; i < 100; i++)
  C[i] = A[i] + B[i];
```

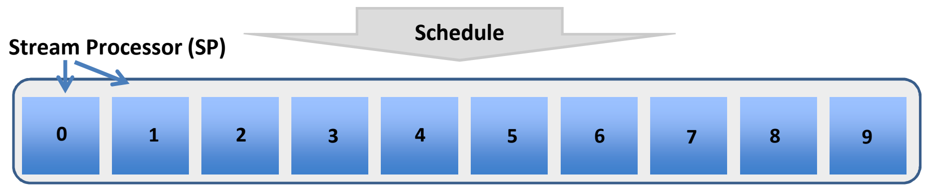
Thread 0: C[0] = A[0] + B[0];	Thread 1: C[1] = A[1] + B[1];	…	Thread 98: C[98] = A[98] + B[98];	Thread 99: C[99] = A[99] + B[99];

Parallel Programming on a GPU

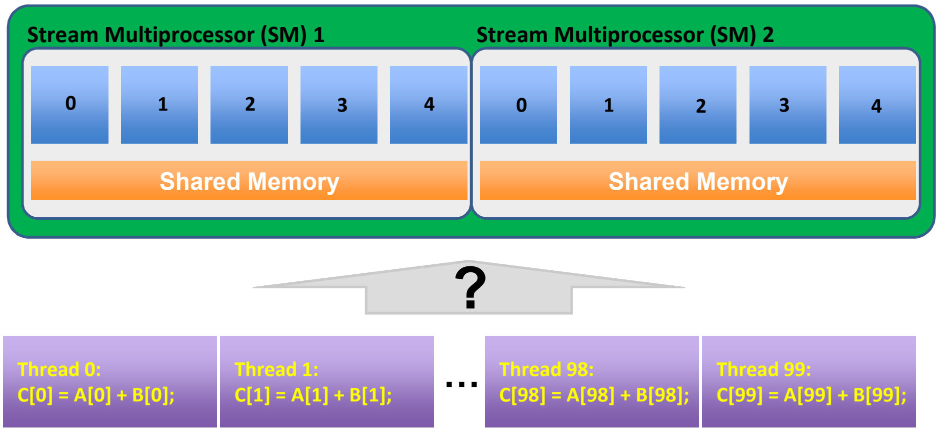
What about Threading ?

```
for(i=0; i < 100; i++)
  C[i] = A[i] + B[i];
```

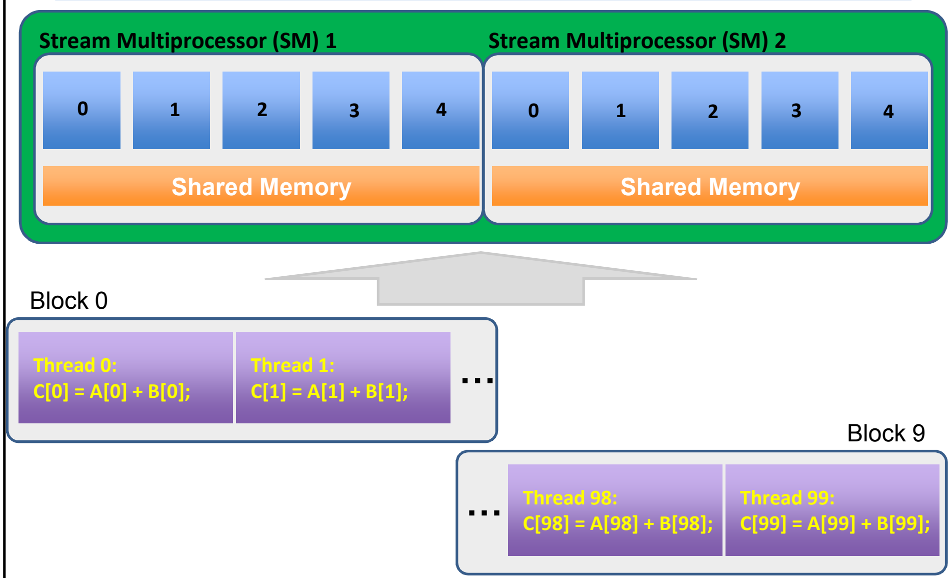
Thread 0: C[0] = A[0] + B[0];	Thread 1: C[1] = A[1] + B[1];	…	Thread 98: C[98] = A[98] + B[98];	Thread 99: C[99] = A[99] + B[99];



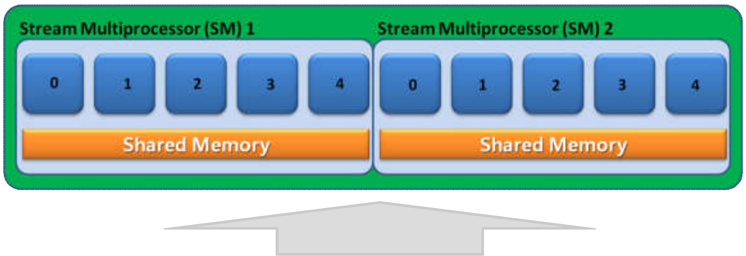
Parallel Programming on a GPU

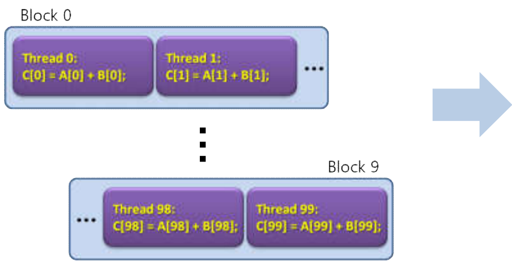


Parallel Programming on a GPU



Parallel Programming on a GPU





1 Thread per block & 100 Blocks
Or
5 Thread per block & 20 Blocks
Or
10 Thread per block & 10 Blocks
Or
20 Thread per block & 5 Blocks
...

Parallel Programming on a GPU

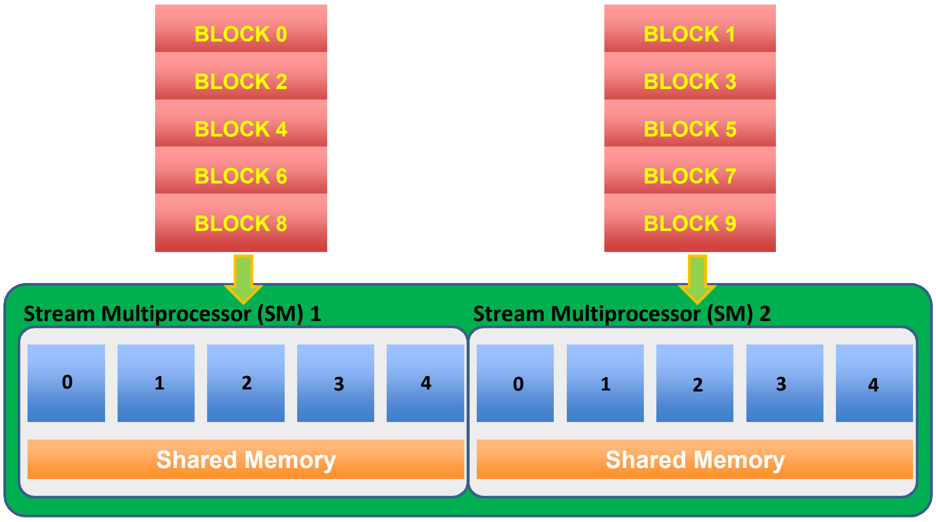
We have one code part …

```
for(i=0; i < 100; i++)
        C[i] = A[i] + B[i];
```

Now, **10 blocks** and **10 threads** per a block !

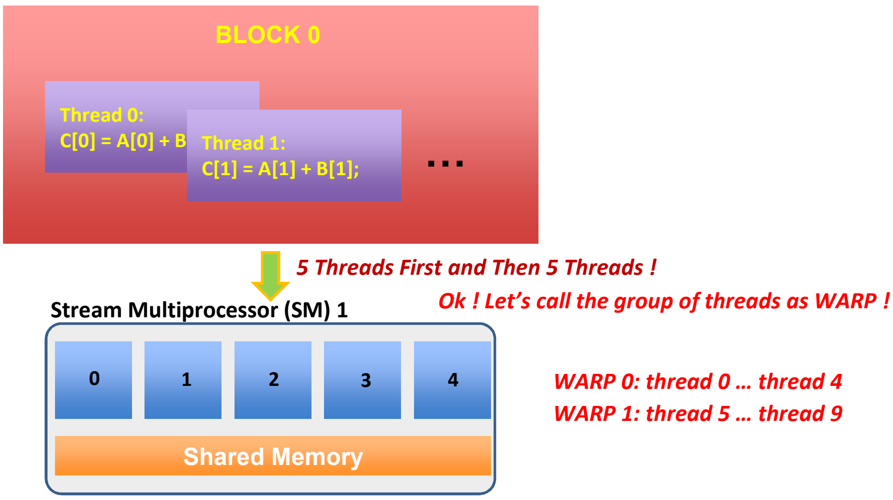
Parallel Programming on a GPU

Now, **10 blocks** and **10 threads** per a block !



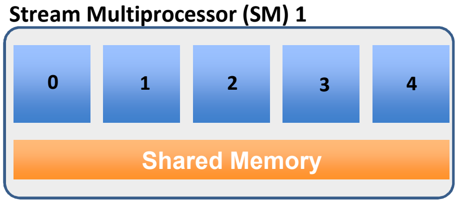
Parallel Programming on a GPU

Now, **10 blocks** and **10 threads** per a block !



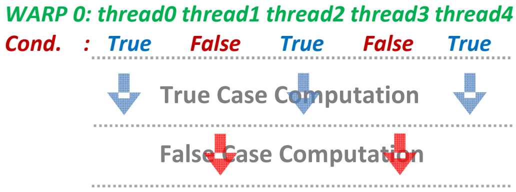
Parallel Programming on a GPU

Note: Warp divergence



WARP 0: thread 0 … thread 4
WARP 1: thread 5 … thread 9

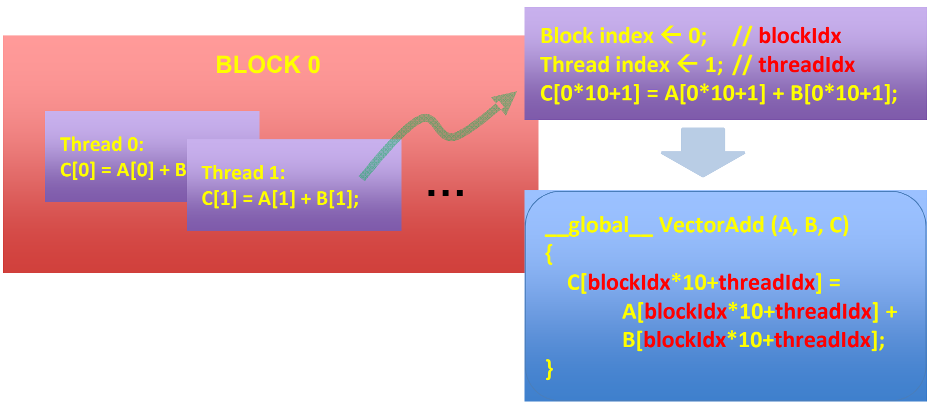
```
__global__ warp_divergence( … )
{
    …
    if ( condition )
    { True Case Computation; }
    else
    { False Case Computation; }
}
```



Parallel Programming on a GPU

Now, **10 blocks** and **10 threads** per a block !

→ We have two indexes: ***block index*** & ***thread index***



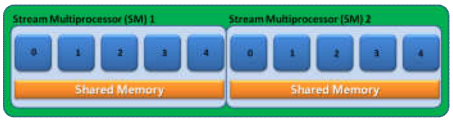
Parallel Programming on a GPU

Hm… Where are A, B, C arrays in a system ?

Parallel Programming on a Manycore

Hm… Where are A, B, C arrays in a system ?

A system ?

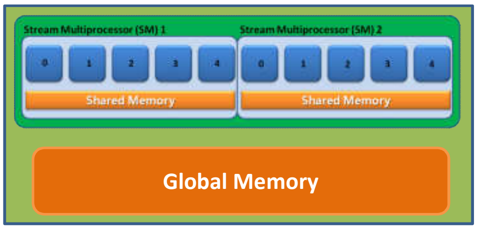


Parallel Programming on a Manycore

Hm… Where are A, B, C arrays in a system ?

A system ?

GPU



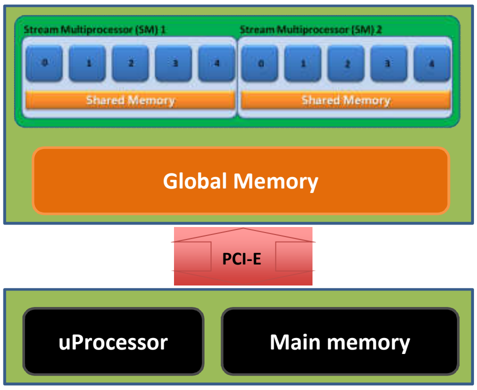
Parallel Programming on a Manycore

Hm… Where are A, B, C arrays in a system ?

A system ?

GPU

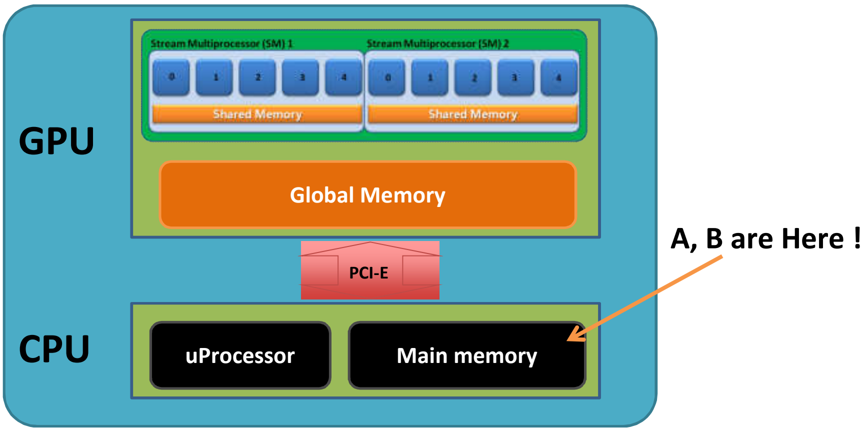
CPU



Parallel Programming on a Manycore

Hm… Where are A, B, C arrays in a system ?

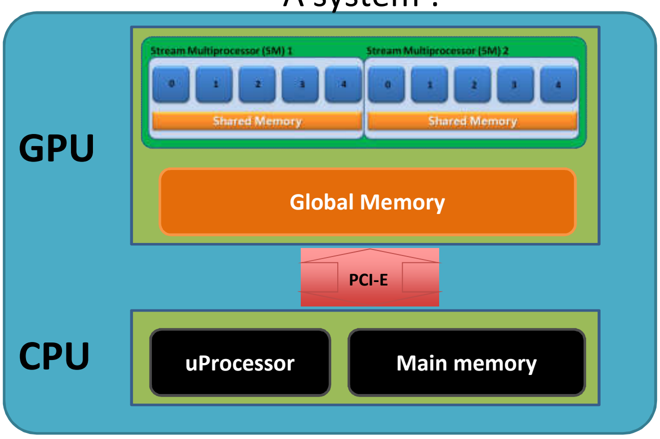
A system !



Parallel Programming on a Manycore

Hm… Where are A, B, C arrays in a system ?

A system !

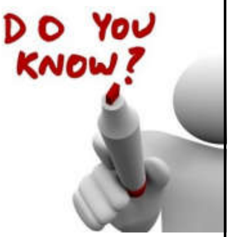


1. cudaMalloc
2. cudaMemcpy
3. Kernel launch
4. cudaMemcpy

https://github.com/jeonggunlee/CUDATeaching/blob/master/01_cuda_lab/04_helloCUDA.ipynb

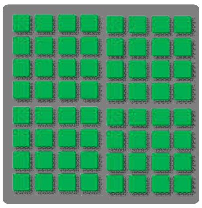
차례

- **Introduction**
  - Multicore/Manycore and GPU
  - GPU on Medical Applications
- **Parallel Programming on GPUs: Basics**
  - Conceptual Introduction
- **GPU Architecture Review**
- **Parallel Programming on GPUs: Practice**
  - Real programming
- **Conclusion**

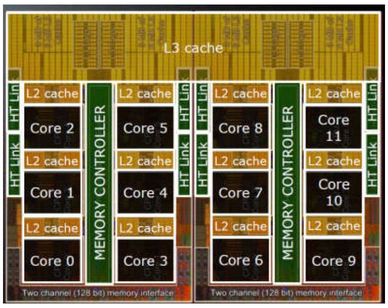


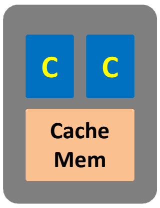
GPU Architecture

- The architecture of GPUs
  - Single Instruction Multiple Thread (SIMT)
  - Manycore Vector Processing
  - *Throughput*-Oriented Accelerating Architecture



- First See a **CPU Case**: AMD 12 Core CPU



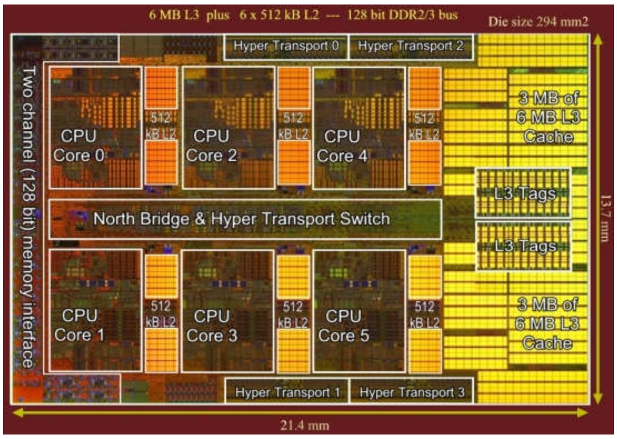


Many on-chip Cache: L1/L2/L3

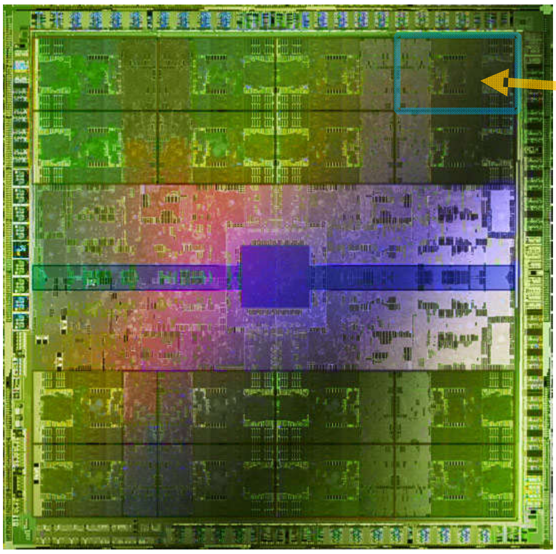
* HT: HyperTransport - low-latency point-to-point link - https://en.wikipedia.org/wiki/HyperTransport

One More ?

- Another AMD 6-core CPU



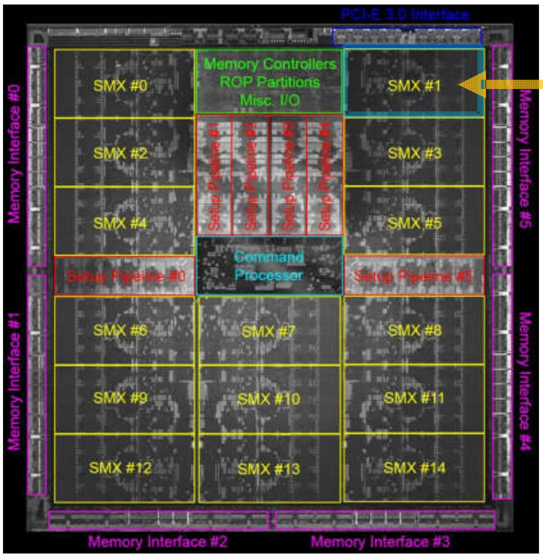
Then, Fermi GPU ?



Compute Unit
(SM = 32 CUDA cores)

16 SM = **512** Cores

Then, Kepler GPU ?

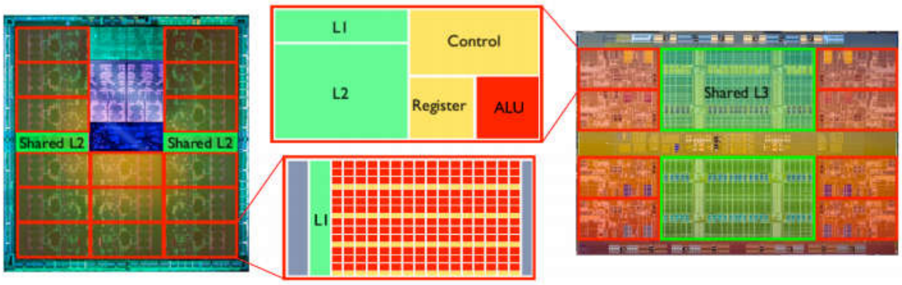


Compute Unit
(SMX = 192 CUDA cores)

16 SMX = **2880** Cores

GK110 Packs 2880 CUDA Cores, 384-bit Memory Interface

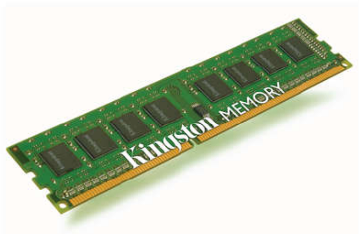
Cache in CPU & GPU



- Die shots of **NVIDIA's GK110 GPU** (left) and **Intel's Nehalem Beckton 8 core CPU** (right) with block diagrams for the GPU streaming multiprocessor and the CPU core.

One more thing in GPU

- Special Memory in GPU
  - Graphics memory: much higher bandwidth than standard CPU memory (QDR)



CPUs use DRAM

GPUs use Graphics DRAM

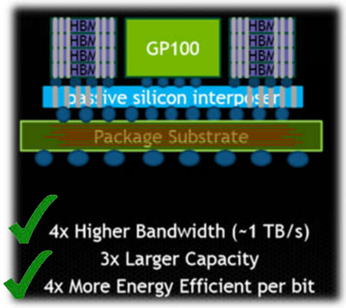
One more thing in GPU

- Special Memory in GPU
  - Graphics memory: much higher bandwidth than standard CPU memory (QDR)

CPUs use DRAM

GPUs use Graphics DRAM

Stacked Memory in Pascal



HBM2
High Bandwidth Memory

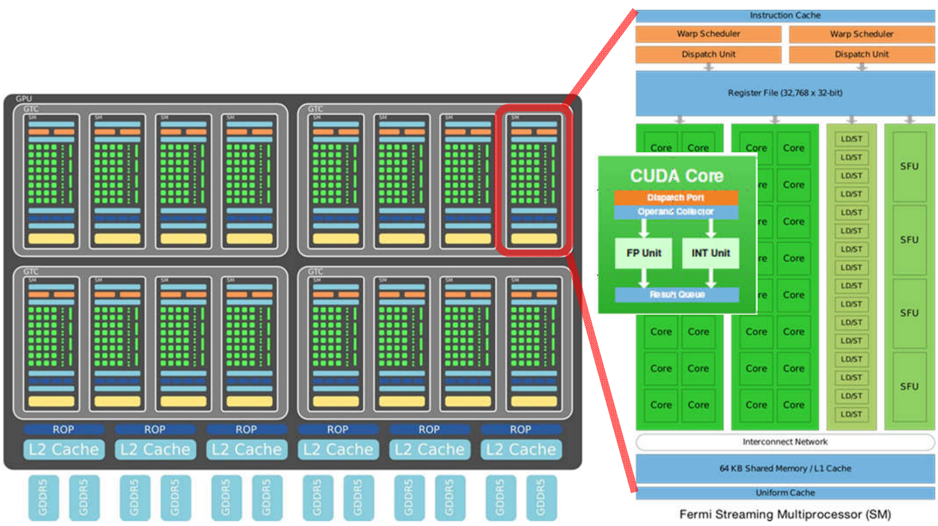
Simple Comparison

- GPU vs CPU: Theoretical Peak capabilities

	NVIDIA Fermi	AMD Magny-Cours (6172)
Cores	448 (1.15GHz)	12 (2.1GHz)
Operations/cycle	1	4
DP Performance (peak)	515 GFlops	101 GFlops
Memory Bandwidth (peak)	144 GB/s	27.5 GB/s

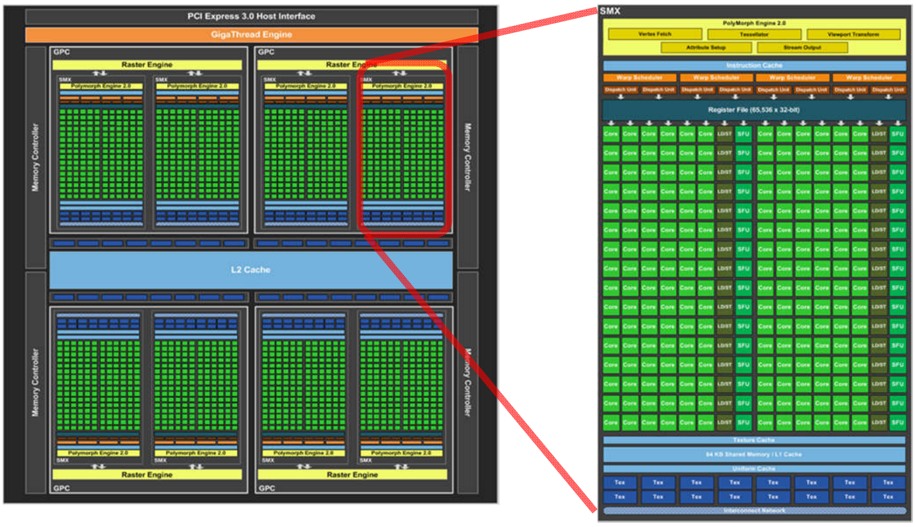
- For these particular example, GPU's theoretical advantage is ~5x for both compute and main memory bandwidth
- Performance very much depends on applications
  - Depends on how well a target application is suited to/tuned for architecture

Nvidia GPU: Fermi (2009)



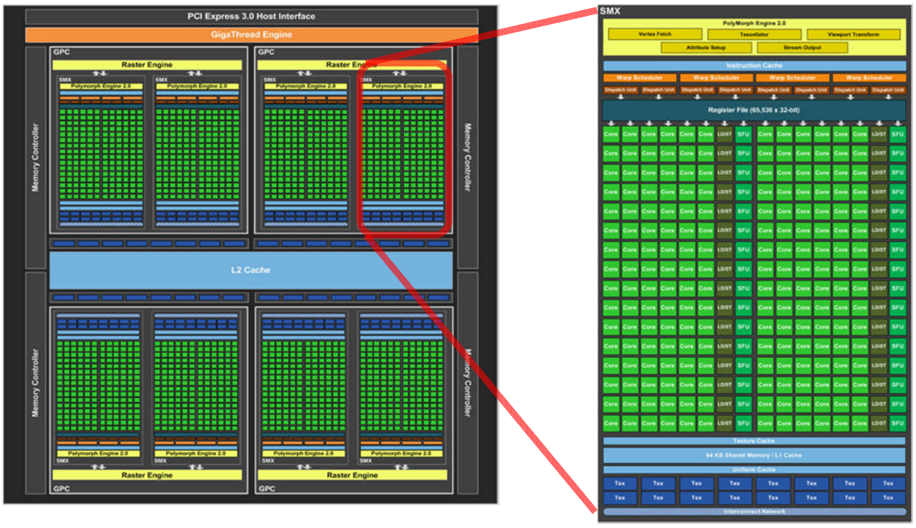
ref: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

Nvidia GPU: Kepler (2012)



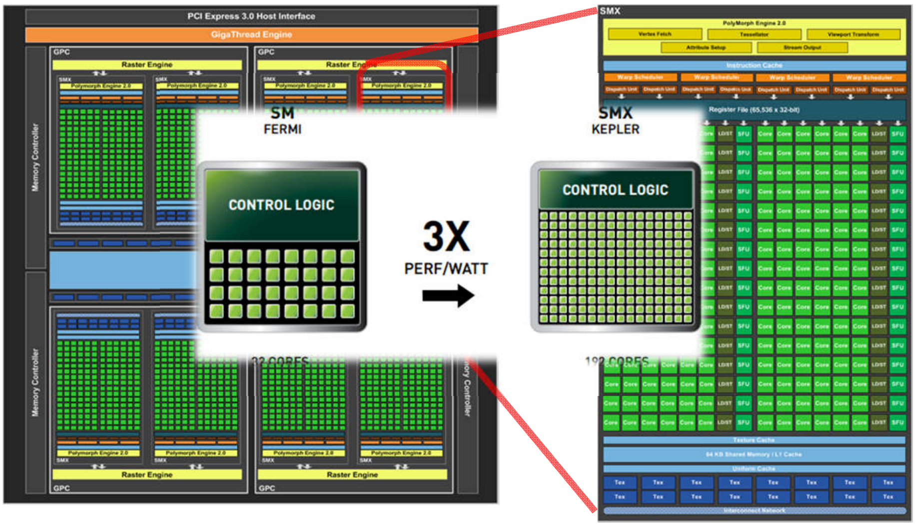
ref: http://www.nvidia.com/object/nvidia-kepler.html

Nvidia GPU: Kepler (2012)



ref: http://www.nvidia.com/object/nvidia-kepler.html

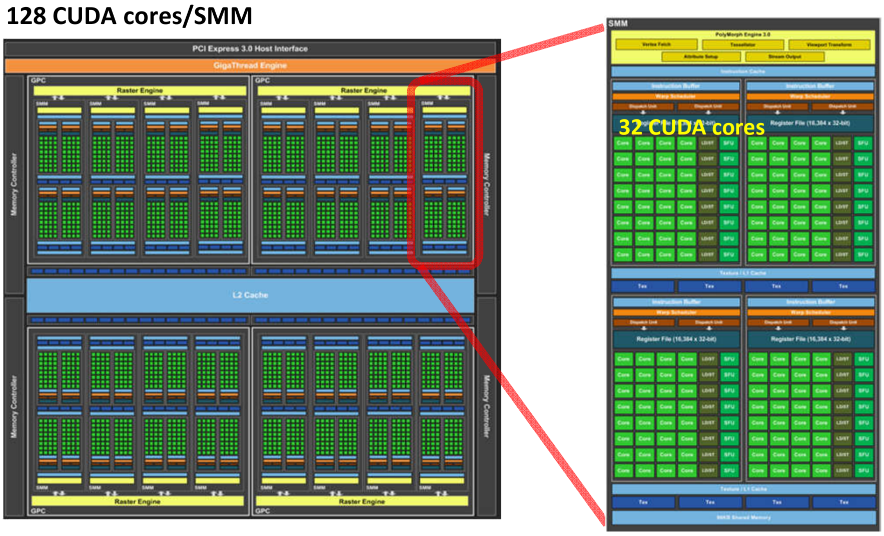
Nvidia GPU: Kepler (2012)



ref: http://www.nvidia.com/object/nvidia-kepler.html

Nvidia GPU: Maxwell (2014)

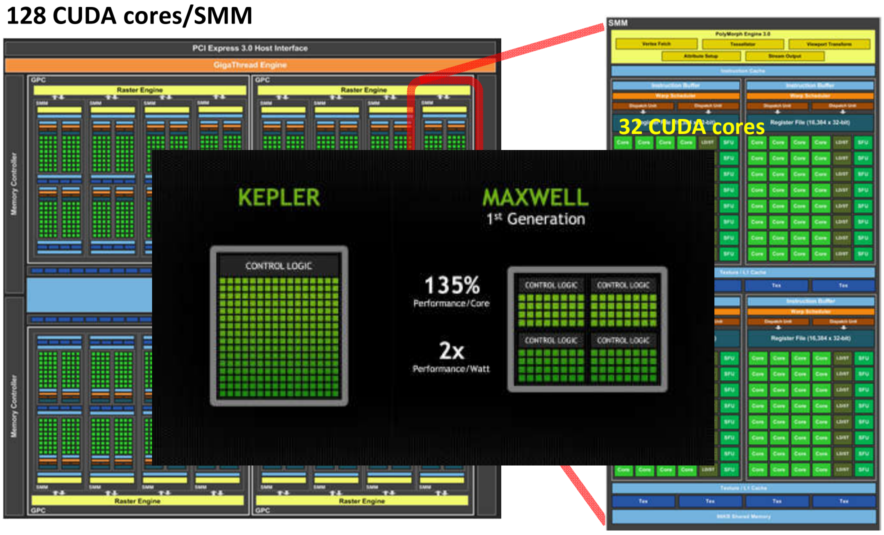
128 CUDA cores/SMM



ref: http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF

Nvidia GPU: Maxwell (2014)

128 CUDA cores/SMM



ref: http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF

Nvidia GPU: Pascal (2016)

TESLA P100 GPU: GP100

56 SMs

3584 CUDA Cores

5.3 TF Double Precision

10.6 TF Single Precision

21.2 TF Half Precision

16 GB HBM2

720 GB/s Bandwidth

Looking at an individual SM, there are 64 CUDA cores, and each SM has a 256K register file, which is four times the size of the shared L2 cache size. In total, the GP100 has 14,336K of register file space. Compared to Maxwell, each core in Pascal has twice as many registers, 1.33 times more shared memory

Nvidia GPU: Pascal (2016)

GP100 SM

	GP100
CUDA Cores	64
Register File	256 KB
Shared Memory	64 KB
Active Threads	2048
Active Blocks	32

Looking at an individual SM, there are 64 CUDA cores, and each SM has a 256K register file, which is four times the size of the shared L2 cache size. In total, the GP100 has 14,336K of register file space. Compared to Maxwell, each core in Pascal has twice as many registers, 1.33 times more shared memory

Nvidia GPU: Pascal (2016)

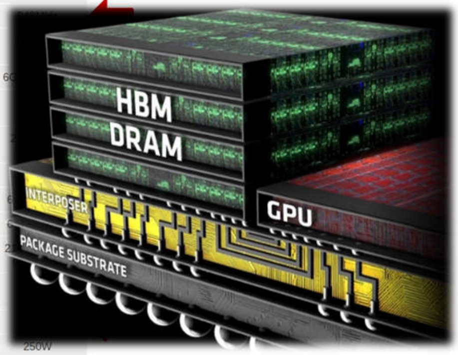
NVIDIA Tesla Family Specification Comparison

	Tesla P100	Tesla K80	Tesla K40	Tesla M40
Stream Processors	3584	2 x 2496	2880	3072
Core Clock	1328MHz	562MHz	745MHz	948MHz
Boost Clock(s)	1480MHz	875MHz	810MHz, 875MHz	1114MHz
Memory Clock	1.4Gbps HBM2	5GHz GDDR5	6GHz GDDR5	6GHz GDDR5
Memory Bus Width	4096-bit	2 x 384-bit	384-bit	384-bit
Memory Bandwidth	720GB/sec	2 x 240GB/sec	288GB/sec	288GB/sec
VRAM	16GB	2 x 12GB	12GB	12GB
Half Precision	21.2 TFLOPS	8.74 TFLOPS	4.29 TFLOPS	6.8 TFLOPS
Single Precision	10.6 TFLOPS	8.74 TFLOPS	4.29 TFLOPS	6.8 TFLOPS
Double Precision	5.3 TFLOPS (1/2 rate)	2.91 TFLOPS (1/3 rate)	1.43 TFLOPS (1/3 rate)	213 GFLOPS (1/32 rate)
GPU	GP100 (610mm2)	GK210	GK110B	GM200
Transistor Count	15.3B	2 x 7.1B(?)	7.1B	8B
TDP	300W	300W	235W	250W
Cooling	N/A	Passive	Active/Passive	Passive
Manufacturing Process	TSMC 16nm FinFET	TSMC 28nm	TSMC 28nm	TSMC 28nm
Architecture	Pascal	Kepler	Kepler	Maxwell 2

Nvidia GPU: Pascal (2016)

NVIDIA Tesla Family Specification Comparison

	Tesla P100	Tesla K80	Tesla K40	Tesla M40
Stream Processors	3584	2 x 2496	2880	3072
Core Clock	1328MHz	562MHz	745MHz	[illegible]
Boost Clock(s)	1480MHz	875MHz	810MHz, 875MHz	[illegible]
Memory Clock	1.4Gbps HBM2	5GHz GDDR5	6GHz GDDR5	[illegible]
Memory Bus Width	4096-bit	2 x 384-bit	384-bit	[illegible]
Memory Bandwidth	720GB/sec	2 x 240GB/sec	288GB/sec	[illegible]
VRAM	16GB	2 x 12GB	12GB	[illegible]
Half Precision	21.2 TFLOPS	8.74 TFLOPS	4.29 TFLOPS	[illegible]
Single Precision	10.6 TFLOPS	8.74 TFLOPS	4.29 TFLOPS	[illegible]
Double Precision	5.3 TFLOPS (1/2 rate)	2.91 TFLOPS (1/3 rate)	1.43 TFLOPS (1/3 rate)	[illegible]
GPU	GP100 (610mm2)	GK210	GK110B	[illegible]
Transistor Count	15.3B	2 x 7.1B(?)	7.1B	[illegible]
TDP	300W	300W	235W	250W
Cooling	N/A	Passive	Active/Passive	Passive
Manufacturing Process	TSMC 16nm FinFET	TSMC 28nm	TSMC 28nm	TSMC 28nm
Architecture	Pascal	Kepler	Kepler	Maxwell 2



Nvidia GPU: Volta (2017)

Nvidia GPU: Volta (2017)

Volta GV100 Full GPU with 84 SM Units

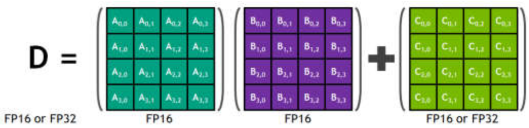
Nvidia GPU: Volta (2017)

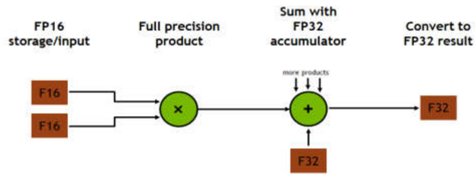
- Volta uses a ***12 nm FinFET*** process.
- Architectural improvements of the Volta architecture include the following:
  - ***CUDA Compute Capability 7.0 ~~> 9.0***
  - ***High Bandwidth Memory 2*** https://en.wikipedia.org/wiki/CUDA
  - ***NVLink 2.0***: a high-bandwidth bus between the CPU and GPU, and between multiple GPUs. Allows much higher transfer speeds than those achievable by using PCI Express; estimated to provide 25 Gbit/s per lane.[7] (Disabled for Titan V)
  - ***Tensor cores***: A tensor core is a unit that multiplies two 4×4 FP16 matrices, and then adds a third FP16 or FP32 matrix to the result by using fused multiply–add (fma) operations, and obtains an FP32 result that could be optionally demoted to an FP16 result. Tensor cores are intended to speed up the training of neural networks.

Volta: Tensor Core

- ***Tensor cores***: A tensor core is a unit that multiplies two 4×4 FP16 matrices, and then adds a third FP16 or FP32 matrix to the result by using **fused multiply–add (fma)** operations, and obtains an FP32 result that could be optionally demoted to an FP16 result. Tensor cores are intended to speed up the training of neural networks.

D = A×B + C





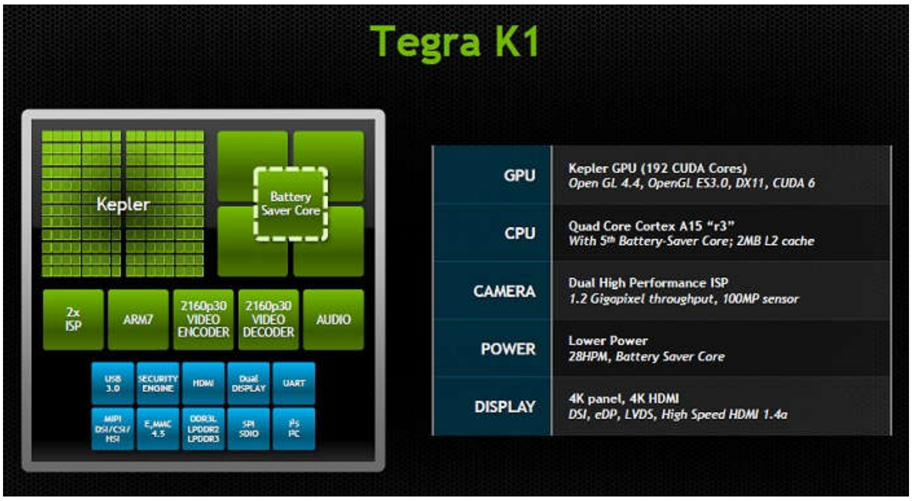
Nvidia GPU: Volta (2017)

Table 1. Comparison of NVIDIA Tesla GPUs

Tesla Product	Tesla K40	Tesla M40	Tesla P100	Tesla V100
GPU	GK180 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)	GV100 (Volta)
SMs	15	24	56	80
TPCs	15	24	28	40
FP32 Cores / SM	192	128	64	64
FP32 Cores / GPU	2880	3072	3584	5120
FP64 Cores / SM	64	4	32	32
FP64 Cores / GPU	960	96	1792	2560
Tensor Cores / SM	NA	NA	NA	8
Tensor Cores / GPU	NA	NA	NA	640
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz	1530 MHz
Peak FP32 TFLOPS[1]	5	6.8	10.6	15.7
Peak FP64 TFLOPS[1]	1.7	.21	5.3	7.8
Peak Tensor TFLOPS[1]	NA	NA	NA	125
Texture Units	240	192	224	320
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2	4096-bit HBM2
Memory Size	Up to 12 GB	Up to 24 GB	16 GB	16 GB
L2 Cache Size	1536 KB	3072 KB	4096 KB	6144 KB
Shared Memory Size / SM	16 KB/32 KB/48 KB	96 KB	64 KB	Configurable up to 96 KB
Register File Size / SM	256 KB	256 KB	256 KB	256KB
Register File Size / GPU	3840 KB	6144 KB	14336 KB	20480 KB
TDP	235 Watts	250 Watts	300 Watts	300 Watts
Transistors	7.1 billion	8 billion	15.3 billion	21.1 billion
GPU Die Size	551 mm²	601 mm²	610 mm²	815 mm²
Manufacturing Process	28 nm	28 nm	16 nm FinFET+	12 nm FFN

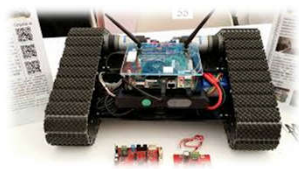
[1] Peak TFLOPS rates are based on GPU Boost Clock

Embedded Mobile ?

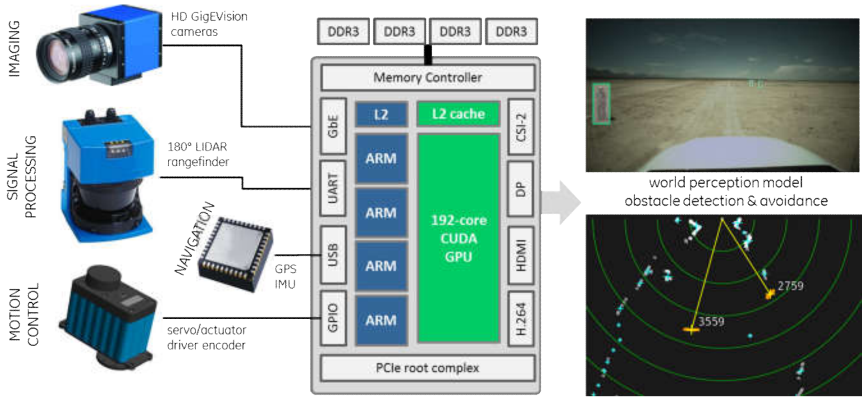


Embedded Mobile ?

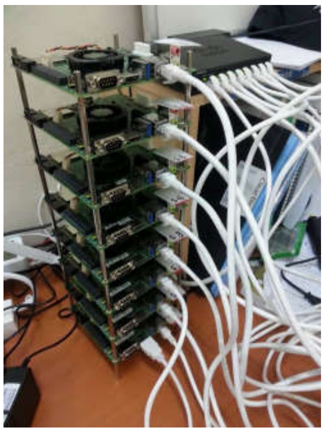




Embedded Mobile ?



Jetson TK1 Micro Server Cluster



Our Lab: Jetson TK1

```
ubuntu@tegra-ubuntu: ~/NVIDIA_CUDA-6.5_Samples/1_Utilities/deviceQuery
ubuntu@tegra-ubuntu:~/NVIDIA_CUDA-6.5_Samples/1_Utilities/deviceQuery$ ./deviceQuery
./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GK20A"
  CUDA Driver Version / Runtime Version          6.5 / 6.5
  CUDA Capability Major/Minor version number:    3.2
  Total amount of global memory:                 1892 MBytes (1984385024 bytes)
  ( 1) Multiprocessors, (192) CUDA Cores/MP:     192 CUDA Cores
  GPU Clock rate:                                852 MHz (0.85 GHz)
  Memory Clock rate:                             924 Mhz
  Memory Bus Width:                              64-bit
  L2 Cache Size:                                 131072 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            Yes
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device PCI Bus ID / PCI location ID:           0 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 6.5, CUDA Runtime Version = 6.5, NumDevs = 1, Device0 = GK20A
Result = PASS
ubuntu@tegra-ubuntu:~/NVIDIA_CUDA-6.5_Samples/1_Utilities/deviceQuery$
```

Demos on Jetson TK1

- Vision

Face detection:

http://elinux.org/Jetson/Tutorials/Vision-controlled_GPIO

Bgfg_segm:

http://pages.hmc.edu/jspjut/class/s2015/e190u/lab/lab5.html
GPIO:

http://www.jetsonhacks.com/2015/09/17/gpio-nvidia-jetson-tk1/

Demos on Jetson TK1

- Machine Learning
  - Try **YOLO** (You only look once) on your Jetson TK1!

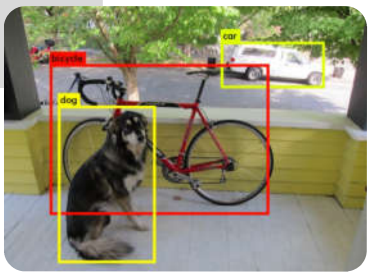
- Follow Steps:
  - git clone https://github.com/pjreddie/darknet
  - cd darknet
  - make
  - wget https://pjreddie.com/media/files/yolo.weights
  - ./darknet detect cfg/yolo.cfg yolo.weights data/dog.jpg

https://devtalk.nvidia.com/default/topic/975387/fastest-framework-for-object-detection-on-jetson-tk1-/

Demos on Jetson TK1

- Machine Learning

```
layer     filters    size              input                output
    0 conv     32  3 x 3 / 1   416 x 416 x   3   ->   416 x 416 x  32
    1 max          2 x 2 / 2   416 x 416 x  32   ->   208 x 208 x  32
   .......
   29 conv    425  1 x 1 / 1    13 x  13 x1024   ->    13 x  13 x 425
   30 detection
Loading weights from yolo.weights...Done!
data/dog.jpg: Predicted in 0.016287 seconds.
car: 54%
bicycle: 51%
dog: 56%
```



  - **Try data/eagle.jpg**
    - **data/dog.jpg**
    - **data/person.jpg**
    - **data/horses.jpg !**

https://devtalk.nvidia.com/default/topic/975387/fastest-framework-for-object-detection-on-jetson-tk1-/

Demos on Jetson TK1

- Machine Learning
- Real-Time Detection on a Webcam
  - Install Tiny YOLO
    - Tiny YOLO is based off of the Darknet reference network and is much faster but less accurate than the normal YOLO model.

```
wget https://pjreddie.com/media/files/tiny-yolo-voc.weights

./darknet detector test cfg/voc.data cfg/tiny-yolo-voc.cfg tiny-yolo-voc.weights data/dog.jpg
```

https://devtalk.nvidia.com/default/topic/975387/fastest-framework-for-object-detection-on-jetson-tk1-/

Demos on Jetson TK1

- Machine Learning
- Real-Time Detection on a Webcam
  - Install Tiny YOLO
    - Tiny YOLO is based off of the Darknet reference network and is much faster but less accurate than the normal YOLO model.

```
wget https://pjreddie.com/media/files/tiny-yolo-voc.weights

./darknet detector demo cfg/voc.data cfg/tiny-yolo-voc.cfg tiny-yolo-voc.weights
```

https://devtalk.nvidia.com/default/topic/975387/fastest-framework-for-object-detection-on-jetson-tk1-/
https://groups.google.com/forum/#!topic/darknet/of0cKVPF7gM

Demos on Jetson TK1

- Machine Learning
- Real-Time Detection on a Webcam
  - Install Tiny YOLO
    - Tiny YOLO is based off of the Darknet reference network and is much faster but less accurate than the normal YOLO model.

```
wget https://pjreddie.com/media/files/tiny-yolo-voc.weights

./darknet detector demo cfg/voc.data cfg/tiny-yolo-voc.cfg tiny-yolo-voc.weights
```

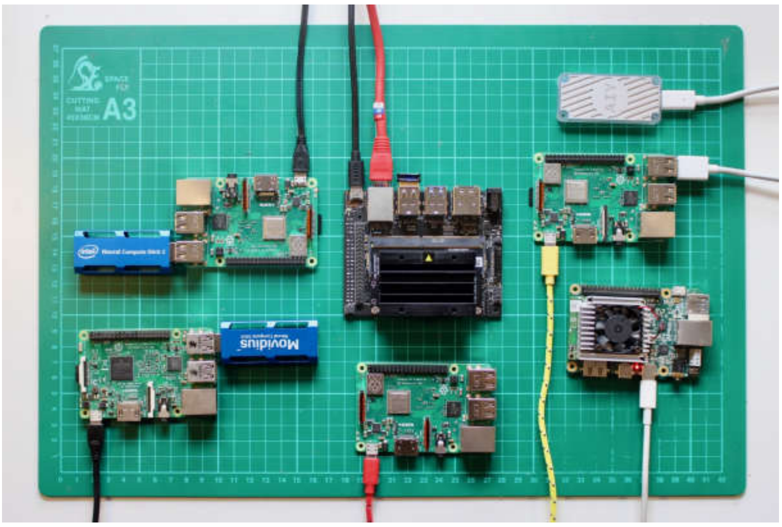
https://devtalk.nvidia.com/default/topic/975387/fastest-framework-for-object-detection-on-jetson-tk1-/
https://groups.google.com/forum/#!topic/darknet/of0cKVPF7gM

Jetson TX1 and TX2

Jetson TX1 and TX2

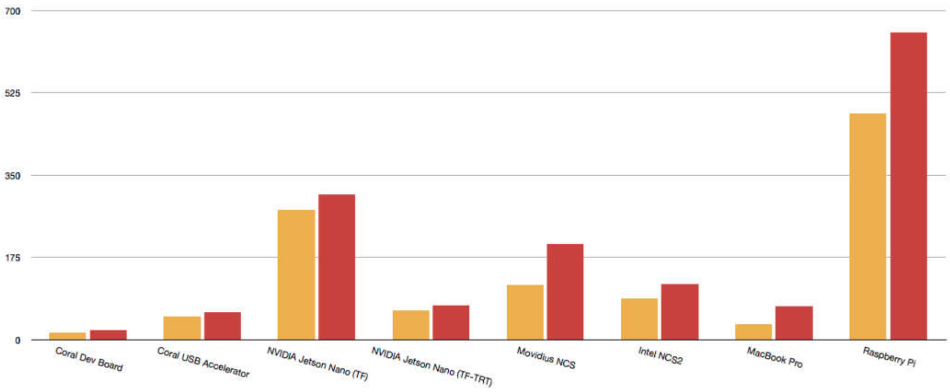
	JETSON TX1	JETSON TX2
GPU	Maxwell	Pascal
CPU	64-bit A57 CPUs	64-bit Denver 2 and A57 CPUs
Memory	4 GB 64 bit LPDDR4 25.6 GB/s	8 GB 128 bit LPDDR4 58.4 GB/s
Storage	16 GB eMMC	32 GB eMMC
Wi-Fi/BT	802.11 2x2 ac/BT Ready	802.11 2x2 ac/BT Ready
Video Encode	2160p @ 30	2160p @ 60
Video Decode	2160p @ 60	2160p @ 60 12 bit support for H.265, VP9
Camera	1.4Gpix/s Up to 1.5Gbps per lane	1.4Gpix/s Up to 2.5Gbps per lane
Mechanical	50mm x 87mm 400-pin Compatible Board to Board Connector	

Deep Learning Inference Devices



https://medium.com/@aallan/benchmarking-edge-computing-ce3f13942245?fbclid=IwAR1C97UY76uEM2Sq-uMZiqfIfYwEld_AxQcl2UiUYvoswK-oBnt5_C6VFlc

Deep Learning Inference Devices



Inferencing speeds in milli-seconds for MobileNet SSD V1 (orange) and MobileNet SSD V2 (red) across all tested platforms. Low numbers are good!

https://medium.com/@aallan/benchmarking-edge-computing-ce3f13942245?fbclid=IwAR1C97UY76uEM2Sq-uMZiqfIfYwEld_AxQcl2UiUYvoswK-oBnt5_C6VFlc

Deep Learning Inference Devices

Board	MobileNet v1 (ms)	MobileNet v2 (ms)	Idle Current (mA)	Peak Current (mA)	Price (US$)
Coral Dev Board	15.7	20.9	600	960	$149.00
Coral USB Accelerator	49.3	58.1	470	880	$74.99+$35.00
NVIDIA Jetson Nano (TF)	276.0	309.3	450	1220	$99.00
NVIDIA Jetson Nano (TF-TRT)	61.6	72.3			
Movidius NCS	115.7	204.5	500	860	$79.00+$35.00
Intel NCS2	87.2	118.6	480	910	$79.00+$35.00
MacBook Pro[1]	33.0	71.0	1570	1950	>$3,000
Raspberry Pi	480.3	654.0	410	1050	$35.00

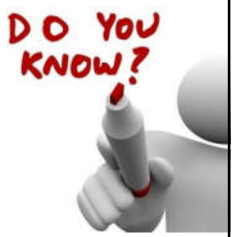
[1] The MacBook Pro takes a +20V supply, all other platforms take a +5V supply.

Benchmarking results in milli-seconds for MobileNet v1 SSD 0.75 depth model and the MobileNet v2 SSD model both trained using the Common Objects in Context (COCO) dataset with an input size of 300×300, alongside idle and peak current consumption for the platforms before and during extended testing.

https://medium.com/@aallan/benchmarking-edge-computing-ce3f13942245?fbclid=IwAR1C97UY76uEM2Sq-uMZiqfIfYwEld_AxQcl2UiUYvoswK-oBnt5_C6VFlc

차례

- **Introduction**
  - Multicore/Manycore and GPU
  - GPU on Medical Applications
- **Parallel Programming on GPUs: Basics**
  - Conceptual Introduction
- **GPU Architecture Review**
- **Parallel Programming on GPUs: Practice**
  - Real programming
- **Conclusion**



CUDA Kernels

- Parallel portion of application: execute as a **kernel**
  - Entire GPU executes kernel
  - Kernel launches create thousands of CUDA threads efficiently

CPU	Host	Executes functions
GPU	Device	Executes kernels

- CUDA threads
  - Lightweight
  - Fast switching: Hardware
  - 1000s execute simultaneously

Kernel Thread !
C[i] = A[i] + B[i];

- Kernel launches create ***hierarchical*** groups of threads
  - **Threads** are grouped into **Blocks**, and Blocks into **Grids**
  - Threads and Blocks represent different levels of parallelism

CUDA C : C with a few keywords

- **Kernel**: function that executes on device (GPU) and can be called from host (CPU)
  - Can only access GPU memory
  - No variable number of arguments
  - No static variables
- Functions must be declared with a qualifier

  __global__: GPU kernel function launched by CPU, must return void

  __device__: can be called from GPU functions

  __host__: can be called from CPU functions (default)
- Qualifiers determines how functions are compiled
  - Controls which compilers are used to compile functions

Kernel Thread !
C[i] = A[i] + B[i];

__device__ add(…)
{ …
C[i] = A[i] + B[i];

CUDA C : C with a few keywords

- **Kernel**: function that executes on device (GPU) and can be called from host (CPU)
- Functions must be declared with a qualifier

  __global__: GPU kernel function launched by CPU, must return void
  __device__: can be called from GPU functions
  __host__: can be called from CPU functions (default)

```
#include <stdio.h>

__device__ void hiDeviceFunction(void)
{ printf("Hello! This is in hiDeviceFunction. \n");}

__global__ void helloCUDA(void)
{
  printf("Hello thread %d\n", threadIdx.x);
  hiDeviceFunction();
}

int main()
{
  helloCUDA<<<1, 1>>>();
  return 0;
}
```

```
ubuntu@tegra-ubuntu:~/NVIDIA_CUDA-6.5_Samples/MYCODE/lab1$ nvcc hello.cu
ubuntu@tegra-ubuntu:~/NVIDIA_CUDA-6.5_Samples/MYCODE/lab1$ ./a.out
Hello thread 0
Hello! This is in hiDeviceFunction.
ubuntu@tegra-ubuntu:~/NVIDIA_CUDA-6.5_Samples/MYCODE/lab1$
```

https://github.com/jeonggunlee/CUDATeaching/blob/master/01_cuda_lab/04_helloCUDA.ipynb

CUDA C : C with a few keywords

- Functions must be declared with a qualifier
  - __global__: GPU kernel function launched by CPU, must return void
  - __device__: can be called from GPU functions
  - __host__: can be called from CPU functions (default)

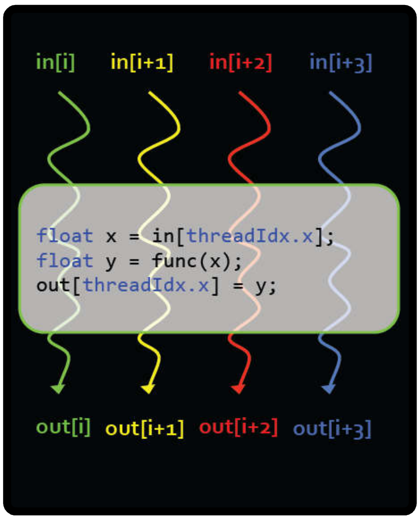
```
#include <stdio.h>

__global__ void helloCUDA(void)
{
  printf("Hello thread %d\n", threadIdx.x);
}

int main()
{
  helloCUDA<<<1, 4>>>();
  cudaDeviceReset();
  return 0;
}
```

```
ubuntu@tegra-ubuntu:~/NVIDIA_CUDA-6.5_Samples/MYCODE/lab1$ nvcc hello.cu
ubuntu@tegra-ubuntu:~/NVIDIA_CUDA-6.5_Samples/MYCODE/lab1$ ./a.out
Hello thread 0
Hello thread 1
Hello thread 2
Hello thread 3
ubuntu@tegra-ubuntu:~/NVIDIA_CUDA-6.5_Samples/MYCODE/lab1$
```

CUDA Kernels : Parallel Threads

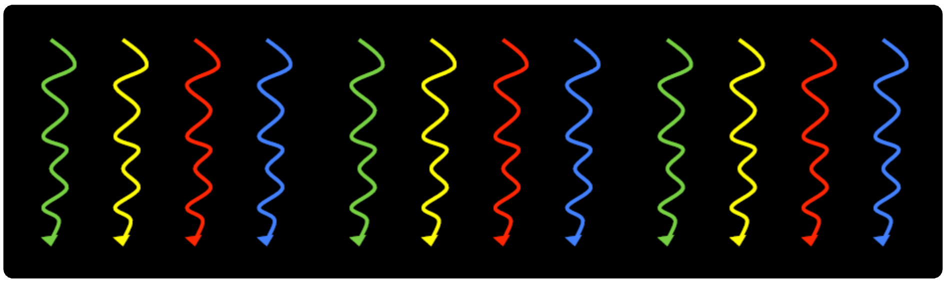
- A kernel is a function executed on a GPU as an array of parallel threads
- All threads execute the same kernel code, but can take different paths
- Each thread has an ID
  - Select input/output data
  - Control decisions

```
__device__ add(...)
{   ...
    i = threadIdx.x;
    C[i] = A[i] + B[i];
    ...
```



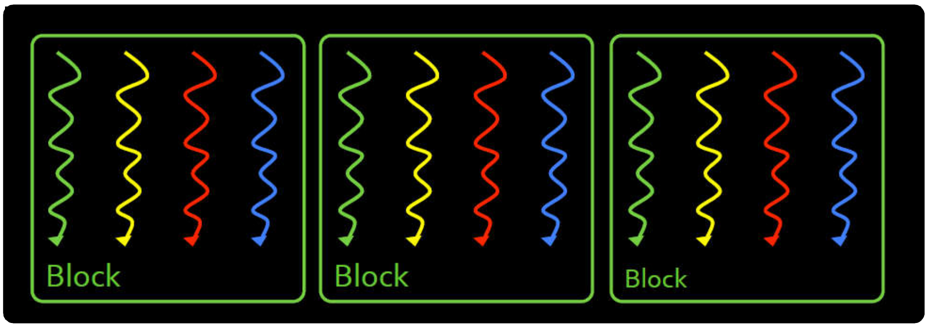
CUDA Thread Organization

- GPUs can handle thousands of concurrent threads
- CUDA programming model supports even more
  - Allows a kernel launch to specify more threads than the GPU can execute concurrently
  - Helps to amortize kernel launch times



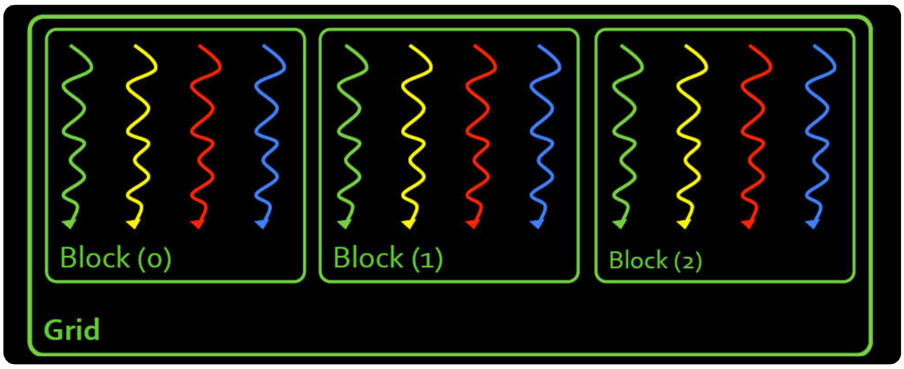
Blocks of threads

- Threads are grouped into blocks

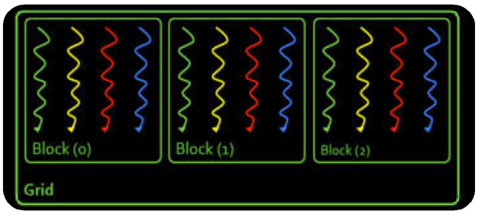


Grids of blocks

- Threads are grouped into blocks
- Blocks are grouped into a grid
- A kernel is executed as a grid of blocks of threads



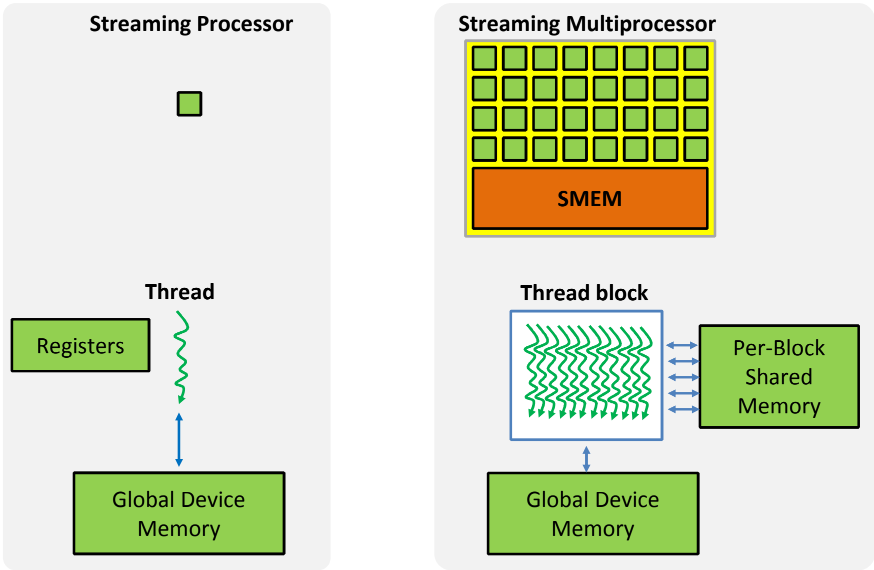
Grids of blocks



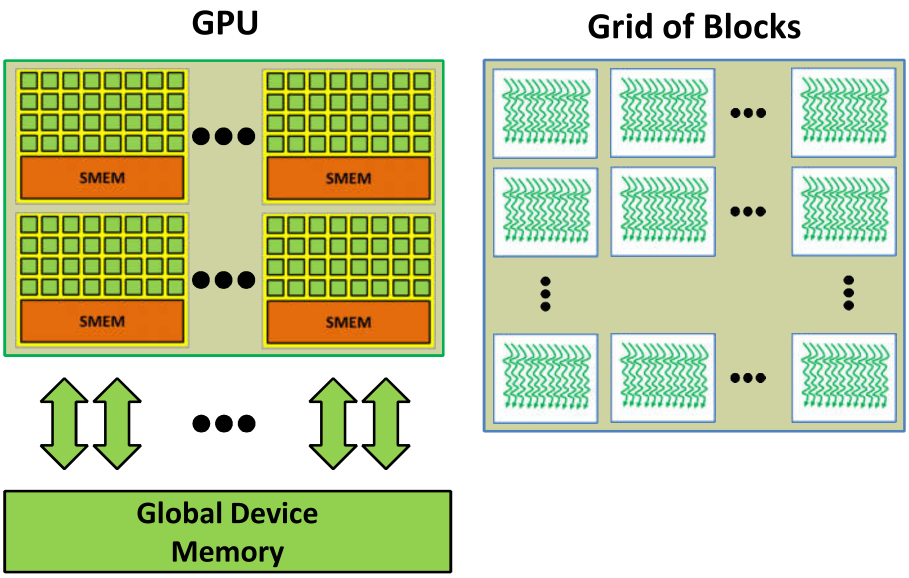

```
#include <stdio.h>

__global__ void helloCUDA(void)
{
  printf("Hello thread %d in block %d\n",
            threadIdx.x, blockIdx.x);
}

int main()
{
  helloCUDA<<<3, 4>>>();
  cudaDeviceReset();
  return 0;
}
```

```
ubuntu@tegra-ubuntu:~/NVIDIA_CUDA-6.5_Samples/MYCODE/lab1$ nvcc hello.cu
ubuntu@tegra-ubuntu:~/NVIDIA_CUDA-6.5_Samples/MYCODE/lab1$ ./a.out
Hello thread 0 in block 1
Hello thread 1 in block 1
Hello thread 2 in block 1
Hello thread 3 in block 1
Hello thread 0 in block 2
Hello thread 1 in block 2
Hello thread 2 in block 2
Hello thread 3 in block 2
Hello thread 0 in block 0
Hello thread 1 in block 0
Hello thread 2 in block 0
Hello thread 3 in block 0
```

Blocks execute on SM



Grids of blocks executes across GPU

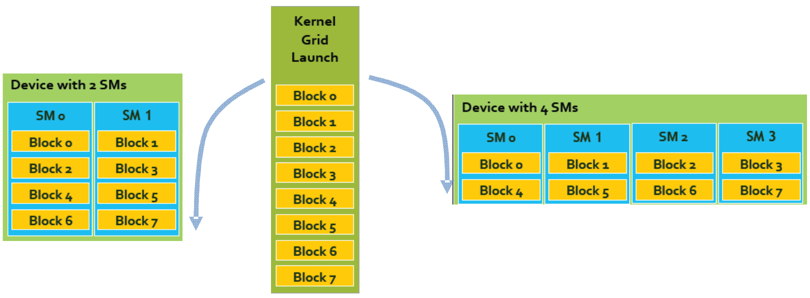


Kernel Execution

- A thread executes on a single streaming processor
  - Allows use of a familiar scalar code within a kernel
- A block executes on a single streaming multiprocessor
  - Threads and blocks do not migrate to different SMs
  - All threads within block execute in concurrently, in parallel
- A streaming multiprocessor may execute multiple blocks
  - Must be able to satisfy aggregate register and memory demands

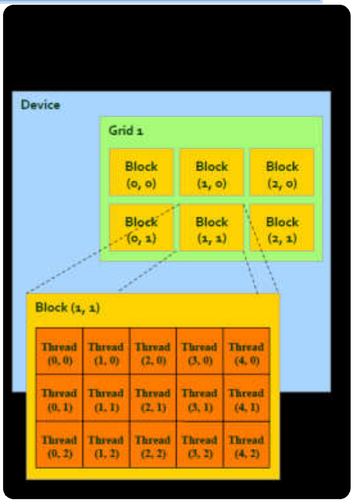
Block abstraction provides scalability

- Blocks may execute in arbitrary order, concurrently or sequentially and parallelism increases with resources
  - Depends on when execution resources become available
- Independent execution of blocks provides scalability
  - Blocks can be distributed across any number of SMs



Thread and Block ID and Dimensions

- Threads
  - 3D IDs, unique within a block
- Thread Blocks
  - 2D IDs, unique within a grid
- Build-in variables
  - **threadIdx**
  - **blockIdx**
  - **blockDim**
  - **gridDim**



- Programmers usually select dimensions that simplify the mapping of the application data to CUDA threads

Examples of Indexes and Indexing

```
__global__ void kernel( int *a )          4
{
  int idx = blockIdx.x*blockDim.x + threadIdx.x;
  a[idx] = 7;             // output: 7 7 7 7  7 7 7 7  7 7 7 7  7 7 7 7
}

__global__ void kernel( int *a )
{
 int idx = blockIdx.x*blockDim.x + threadIdx.x;
  a[idx] = blockIdx.x;    // output: 0 0 0 0  1 1 1 1  2 2 2 2  3 3 3 3
}

__global__ void kernel( int *a )
{
 int idx = blockIdx.x*blockDim.x + threadIdx.x;
  a[idx] = threadIdx.x;   // output: 0 1 2 3  1 2 3 4  0 1 2 3  0 1 2 3
}
```

https://github.com/jeonggunlee/CUDATeaching/blob/master/01_cuda_lab/04_helloCUDA.ipynb

Example of 2D indexing

```
__global__ void kernel( int *a )
{
   int ix = blockIdx.x*blockDim.x + threadIdx.x;
   int iy = blockIdx.y*blockDim.y + threadIdx.y;
   int idx = iy * dimx + ix;

   a[idx] = a[idx] + 1;
}
```

https://github.com/jeonggunlee/CUDATeaching/blob/master/01_cuda_lab/06_2DIndex.ipynb

Let's Start Again from C

OPTIONAL SKIP

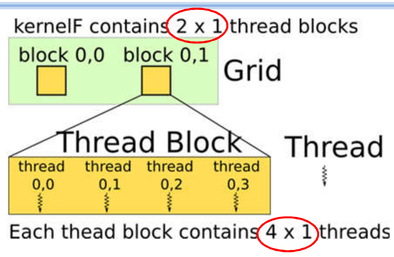
```
int A[2][4];
for(i=0;i<2;i++)
   for(j=0;j<4;j++)
      A[i][j]++;
```

convert into CUDA

```
int A[2*4];
kernelF<<<(2,1),(4,1)>>>(A);   // define 2x4=8 threads
__device__    kernelF(A){      // all threads run same kernel
   i = blockIdx.x;             // each thread block has its id
   j = threadIdx.x;            // each thread has its id
   A[i*2+j]++;                 // each thread has different i and j
}
```

참조:
http://www.es.ele.tue.nl/~heco/courses/EmbeddedComputerArchitecture/

Thread Hierarchy

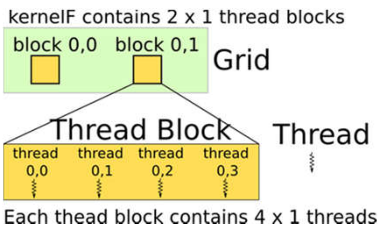


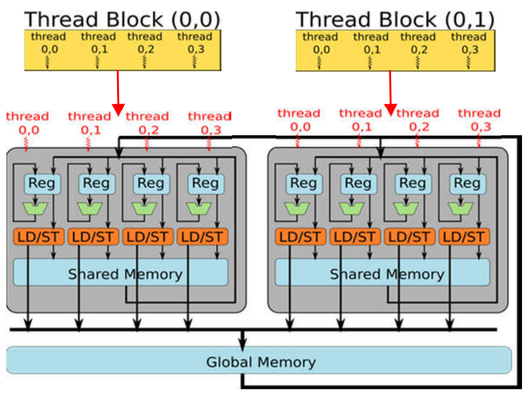
Example:
thread 3 of block 1 operates
on element A[1][3]

```
int A[2*4];
kernelF<<<(2,1),(4,1)>>>(A);   // define 2x4=8 threads
__device__    kernelF(A){      // all threads run same kernel
    i = blockIdx.x;            // each thread block has its id
    j = threadIdx.x;           // each thread has its id
    A[i*2+j]++;                // each thread has different i and j
}
```

참조:
http://www.es.ele.tue.nl/~heco/courses/EmbeddedComputerArchitecture/

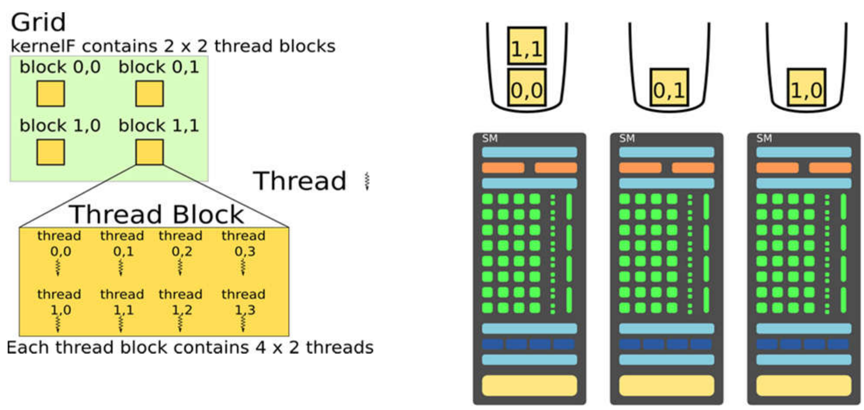
How Are Threads Scheduled?





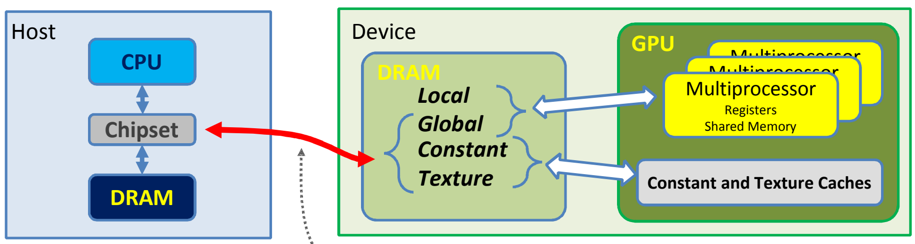
참조:
http://www.es.ele.tue.nl/~heco/courses/EmbeddedComputerArchitecture/

Blocks Are Dynamically Scheduled



참조:
http://www.es.ele.tue.nl/~heco/courses/EmbeddedComputerArchitecture/

Memory Architecture



PCI Express link performance[27][30]

PCI Express version	Line code	Transfer rate[i]	Throughput[i]			
			×1	×4	×8	×16
1.0	8b/10b	2.5 GT/s	250 MB/s	1 GB/s	2 GB/s	4 GB/s
2.0	8b/10b	5 GT/s	500 MB/s	2 GB/s	4 GB/s	8 GB/s
3.0	128b/130b	8 GT/s	984.6 MB/s	3.938 GB/s	7.877 GB/s	15.754 GB/s
4.0 (*expected in 2017*)	128b/130b	16 GT/s	1.969 GB/s	7.877 GB/s	15.754 GB/s	31.508 GB/s
5.0 (*far future*)[28][29]	128b/130b	32 or 25 GT/s[ii]	3.9, or 3.08 GB/s	15.8, or 12.3 GB/s	31.5, or 24.6 GB/s	63.0, or 49.2 GB/s

https://en.wikipedia.org/wiki/PCI_Express

Memory Architecture

PCI Express link performance[27][30]

PCI Express version	Line code	Transfer rate[i]	Throughput[i]			
			×1	×4	×8	×16
1.0	8b/10b	2.5 GT/s	250 MB/s	1 GB/s	2 GB/s	4 GB/s
2.0	8b/10b	5 GT/s	500 MB/s	2 GB/s	4 GB/s	8 GB/s
3.0	128b/130b	8 GT/s	984.6 MB/s	3.938 GB/s	7.877 GB/s	15.754 GB/s
4.0 (*expected in 2017*)	128b/130b	16 GT/s	1.969 GB/s	7.877 GB/s	15.754 GB/s	31.508 GB/s
5.0 (*far future*)[28][29]	128b/130b	32 or 25 GT/s[ii]	3.9, or 3.08 GB/s	15.8, or 12.3 GB/s	31.5, or 24.6 GB/s	63.0, or 49.2 GB/s

https://en.wikipedia.org/wiki/PCI_Express

```
ubuntu@tegra-ubuntu:~/NVIDIA
bandwidthTest      bandwidthT
bandwidthTest.cu  Makefile
ubuntu@tegra-ubuntu:~/NVIDIA
[CUDA Bandwidth Test] - Star
Running on...

 Device 0: GK20A
 Quick Mode

 Host to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     988.7

 Device to Host Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     3793.5

 Device to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     11805.1

Result = PASS
ubuntu@tegra-ubuntu:~/NVIDIA_CUDA-6.5_Samples/1_Utilities/bandwidthTest$
```

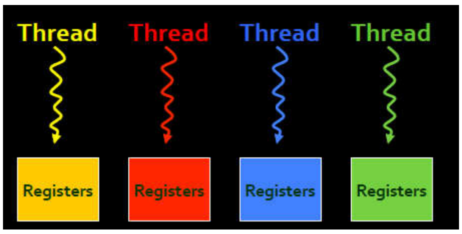
https://github.com/jeonggunlee/CUDATeaching/blob/master/01_cuda_lab/08_DeviceQuery_Bandwidth.ipynb

CUDA Memory Hierarchy

- Thread
  - Registers

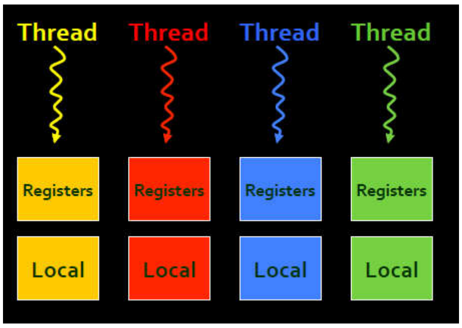
OPTIONAL

SKIP



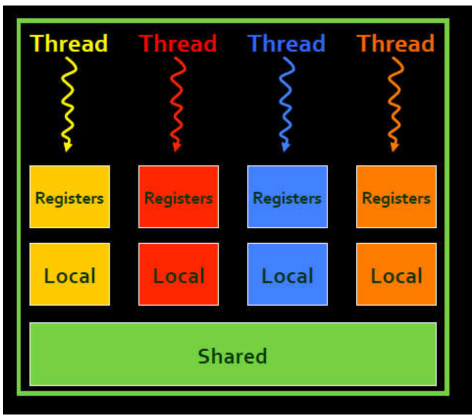
CUDA Memory Hierarchy

- Thread
  - Registers
  - Local memory



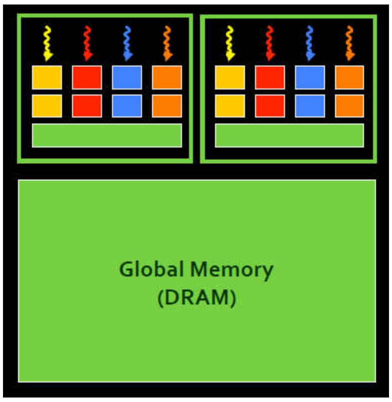
CUDA Memory Hierarchy

- Thread
  - Registers
  - Local memory
- Thread Block
  - Shared memory



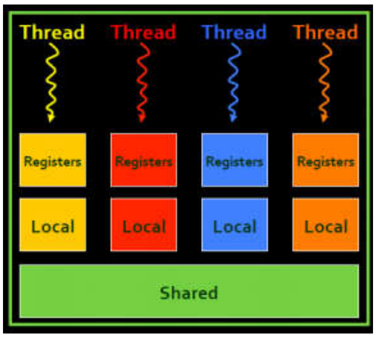
CUDA Memory Hierarchy

- Thread
  - Registers
  - Local memory
- Thread Block
  - Shared memory
- All Thread Blocks
  - Global memory



Shared Memory

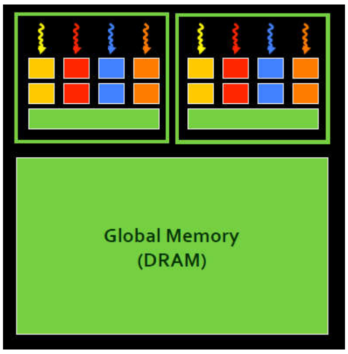
__shared__ <type> x[<elements>];



- Allocated per thread block
- Scope: threads in block
- Data lifetime: same as block
- Capacity: small (about 48kB)
- Latency: a few cycles
- Bandwidth: very high
  - SM:32*4B*1.15GHz/2 = 73.6 GB/s
  - GPU: 14*32*4B*1.15GHz/2 = 1.03 TB/s
- Common uses
  - Sharing data among threads in a block
  - User-managed cache (to reduce global memory accesses)

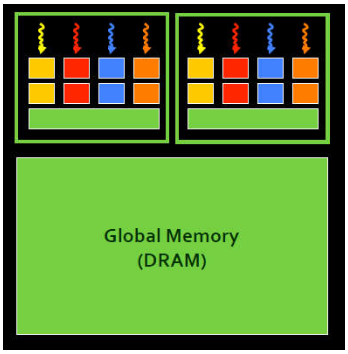
Global Memory

- Allocated explicitly by host (CPU) thread
- Scope: all threads of all kernels
- Data lifetime: determin by host (CPU) thread
  - **cudaMalloc(void **pointer, size_t nbytes)**
  - **cudaFree(void* pointer)**
- Capacity: large (1-12GB)
- Latency: 400-800 cycles
- Bandwidth: 156 GB/s, → 1TB/s
  - Data access patterns will limit bandwidth achieved in practice
- Common uses
  - Staging data transfers to/from CPU
  - Staging data between kernel launches



Global Memory

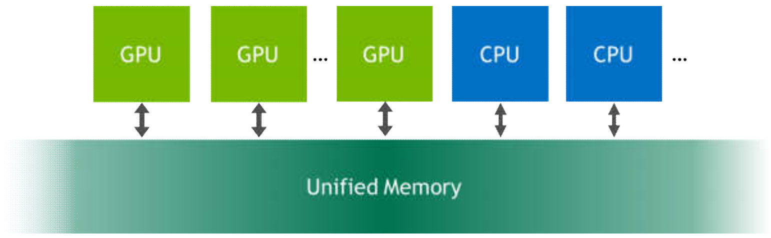
```
ubuntu@tegra-ubuntu:~/NVIDIA_CUDA-6.5_Samples/1_Utilities/bandwidthTest$ ls
bandwidthTest      bandwidthTest.o   NsightEclipse.xml
bandwidthTest.cu   Makefile          readme.txt
ubuntu@tegra-ubuntu:~/NVIDIA_CUDA-6.5_Samples/1_Utilities/bandwidthTest$ ./bandwidthTest
[CUDA Bandwidth Test] - Starting...
Running on...

 Device 0: GK20A
 Quick Mode

 Host to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     988.7

 Device to Host Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     3793.5

 Device to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(MB/s)
   33554432                     11805.1

Result = PASS
ubuntu@tegra-ubuntu:~/NVIDIA_CUDA-6.5_Samples/1_Utilities/bandwidthTest$
```



https://github.com/jeonggunlee/CUDATeaching/blob/master/01_cuda_lab/08_DeviceQuery_Bandwidth.ipynb

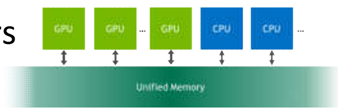
Unified Memory

- Unified Memory for CUDA Beginners



Unified Memory

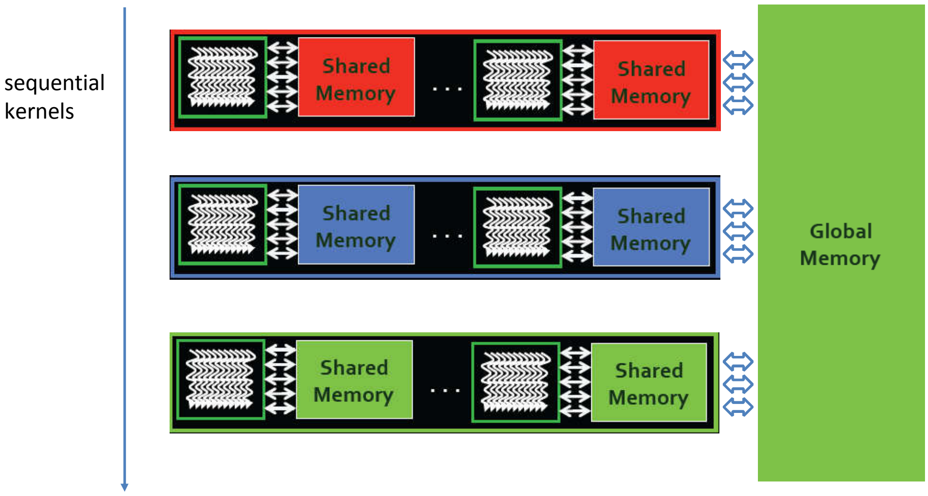
- Unified Memory for CUDA Beginners



On-Demand Paging

```
__global__
void setValue(int *ptr, int index, int val)
{
  ptr[index] = val;
}

void foo(int size) {
  char *data;
  cudaMallocManaged(&data, size);      <- Unified Memory allocation

  memset(data, 0, size);               <- Access all values on CPU

  setValue<<<...>>>(data, size/2, 5);  <- Access one value on GPU
  cudaDeviceSynchronize();

  useData(data);

  cudaFree(data);
}
```

Communication and Data Persistence



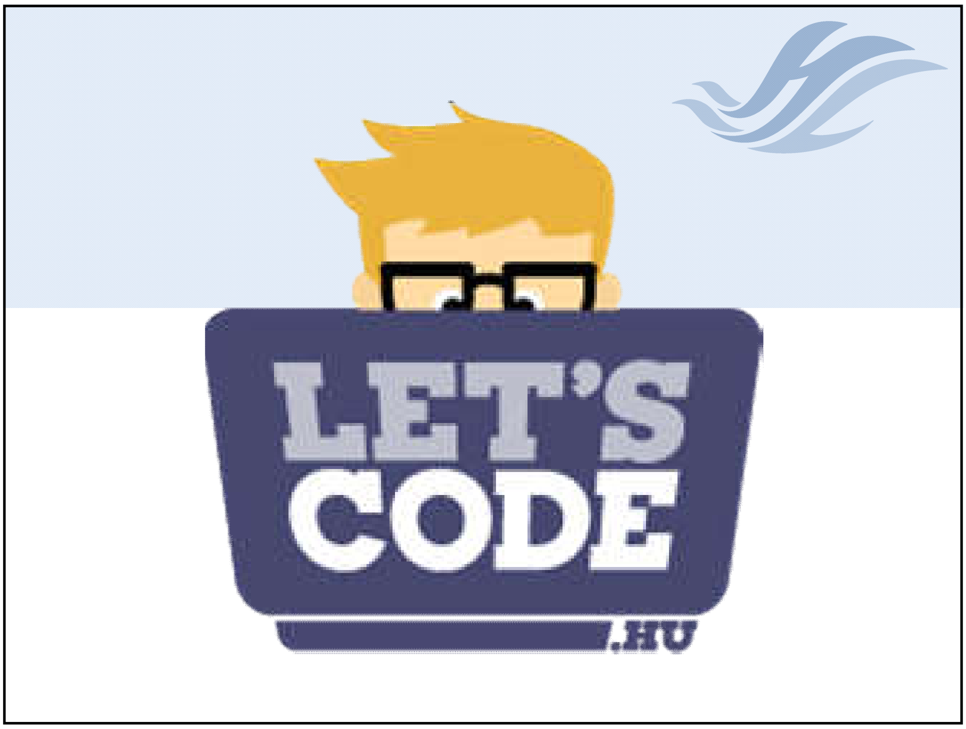
Managing Device (GPU) Memory

- Host (CPU) manage device (GPU) memory
  - cudaMalloc(void** pointer, size_t num_bytes)
  - cudaMemset(void** pointer, int value, size_t count)
  - cudaFree(void* pointer)
- Ex.: allocate and initialize array of 1024 ints on device

```
// allocate and initialize int x[1024] on device
int n = 1024;
int num_bytes = 1024*sizeof(int);
int* d_x = 0; // holds device pointer
cudaMalloc((void**)&d_x, num_bytes);
cudaMemset(d_x, 0, num_bytes);
cudaFree(d_x);
```

Transferring Data

- **cudaMemcpy**(void* dst, void* src, size_t num_bytes, enum cudaMemcpyKind direction);
  - Returns to host thread after the copy completes
  - Block CPU thread until all bytes have been copied
  - Doesn't start copying until previous CUDA calls complete
- Direction controlled by num cudaMemcpyKind
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice
- CUDA also provides non-blocking
  - Allows program to overlap data transfer with concurrent computation on host and device



Example: SAXPY Kernel

```
// [compute] for(i=0; i<n; i++) y[i] = a*x[i] + y[i];
// Each thread processes on element
__global__ void saxpy(int n, float a, float* x, float* y)

?

int main()

?
```

Example: SAXPY Kernel

```
// [compute] for(i=0; i<n; i++) y[i] = a*x[i] + y[i];
// Each thread processes on element
__global__ void saxpy(int n, float a, float* x, float* y)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  if( i<n ) y[i] = a*x[i] + y[i];
}

int main()

?
```

Example: SAXPY Kernel

```
// [compute] for(i=0; i<n; i++) y[i] = a*x[i] + y[i];
// Each thread processes on element
__global__ void saxpy(int n, float a, float* x, float* y)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  if( i<n ) y[i] = a*x[i] + y[i];
}

int main()
{
  ...
  // invoke parallel SAXPY kernel with 256 threads / block
  int nblocks = (n + 255)/256;
  saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);
}
```

Example: SAXPY Kernel

```
// [compute] for(i=0; i<n; i++) y[i] = a*x[i] + y[i];
// Each thread processes on element
__global__ void saxpy(int n, float a, float* x, float* y)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  if( i<n ) y[i] = a*x[i] + y[i];
}
```

Device Code

```
int main()
{
  ...
  // invoke parallel SAXPY kernel with 256 threads / block
  int nblocks = (n + 255)/256’
  saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);
}
```

Example: SAXPY Kernel

```
// [compute] for(i=0; i<n; i++) y[i] = a*x[i] + y[i];
// Each thread processes on element
__global__ void saxpy(int n, float a, float* x, float* y)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  if( i<n ) y[i] = a*x[i] + y[i];
}
```

```
int main()
{
  …
  // invoke parallel SAXPY kernel with 256 threads / block
  int nblocks = (n + 255)/256’
  saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);
}
```

Host Code

Example: SAXPY Kernel

```
int main()
{
  // allocate and initialize host (CPU) memory
  float* x = …;
  float* y = …;

  // allocate device (GPU) memory
  float *d_x, *d_y;
  cudaMalloc((void**) &d_x, n*sizeof(float));
  cudaMalloc((void**) &d_y, n*sizeof(float));
  // copy x and y from host memory to device memory
  cudaMemcpy(d_x, x, n*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_x, x, n*sizeof(float), cudaMemcpyHostToDevice);
  // invoke parallel SAXPY kernel with 256 threads / block
  int nblocks = (n + 255)/256’
  saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);
```

Example: SAXPY Kernel

```
int main()
{
  // allocate and initialize host (CPU) memory
  float* x = ...;
  float* y = ...;
  // allocate device (GPU) memory
  float *d_x, *d_y;
  cudaMalloc((void**) &d_x, n*sizeof(float));
  cudaMalloc((void**) &d_y, n*sizeof(float));
  // copy x and y from host memory to device memory
  cudaMemcpy(d_x, x, n*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_x, x, n*sizeof(float), cudaMemcpyHostToDevice);
  // invoke parallel SAXPY kernel with 256 threads / block
  int nblocks = (n + 255)/256'
  saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);
```

Example: SAXPY Kernel

```
  // invoke parallel SAXPY kernel with 256 threads / block
  int nblocks = (n + 255)/256;
  saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);

  // copy y from device (GPU) memory to host (CPU) memory
  cudaMemcpy(d_y, y, n*sizeof(float), cudaMemcpyDeviceToHost);

  // do something with the result ...

  // free device (GPU) memory
  cudaFree(d_x);
  cudaFree(d_y);

  return 0;
}
```

Example: Check the Differences

```
void saxpy_serial(int n, float a, float* x, float* y)
{
  for(int i = 0; i < n; i++)
    y[i] = a*x[i] + y[i];
}
  // invoke host SAXPY function
  saxpy_serial(n, 2.0, x, y);
}
```

O(n)

Standard C Code

```
__global__ void saxpy(int n, float a, float* x, float* y)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  if( i<n ) y[i] = a*x[i] + y[i];
}
  // invoke parallel SAXPY kernel with 256 threads / block
  int nblocks = (n + 255)/256;
  saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);
}
```

O(1)

CUDA C Code

One More: multiplication table

- Parallel construction of multiplication table

```
//  multiplication table using CUDA
...
#include <cuda_runtime.h>

#define BLOCK_SIZE 8
#define THREAD_SIZE 9

int main()
{
 int *host_Result; //Save result data of host
 int *device_Result; //Save result data of device

 int i=0, j=0;

 //Allocate host memory
 host_Result = (int *)malloc( BLOCK_SIZE * THREAD_SIZE * sizeof(int) );

 //Allocate device memory
 cudaMalloc( (void**) &device_Result, sizeof(int) * BLOCK_SIZE * THREAD_SIZE);

 //Function name <<BLOCK_SIZE, THREAD_SIZE>>> parameters
 test <<<BLOCK_SIZE, THREAD_SIZE>>>(device_Result); //Execute Device code
```

One More: multiplication table

- Parallel construction of multiplication table

```
//Function name <<BLOCK_SIZE, THREAD_SIZE>>> parameters
test <<<BLOCK_SIZE, THREAD_SIZE>>>(device_Result); //Execute Device code

//Copy device result to host result
cudaMemcpy( host_Result,
            device_Result, sizeof(int) * BLOCK_SIZE * THREAD_SIZE,
            cudaMemcpyDeviceToHost );

//Print result
for(j=0; j<BLOCK_SIZE; j++)
{
 printf("%3d step\n", (j + 2));
 for(i=0; i<THREAD_SIZE; i++)
 {   printf("%3d X %3d = %3d\n", j+2, i+1, host_Result[j * THREAD_SIZE + i]);  }
 printf("\n");
}

free(host_Result); //Free host memory
cudaFree(device_Result); //Free device memory

return 1;
}
```

One More: multiplication table

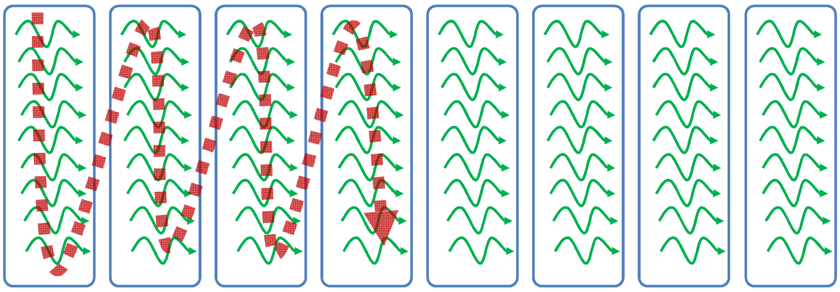
- Parallel construction of multiplication table

```
// Device code
__global__ void test(int *result)
{
    int tidx, bidx;
    tidx = threadIdx.x;        //x-coordinate of thread
    bidx = blockIdx.x;         //x-coordinate of block

    result[                    ] = 
}
```

One More: multiplication table

- Parallel construction of multiplication table

Block 0 Block 7



2단 9단

One More: multiplication table

- Parallel construction of multiplication table

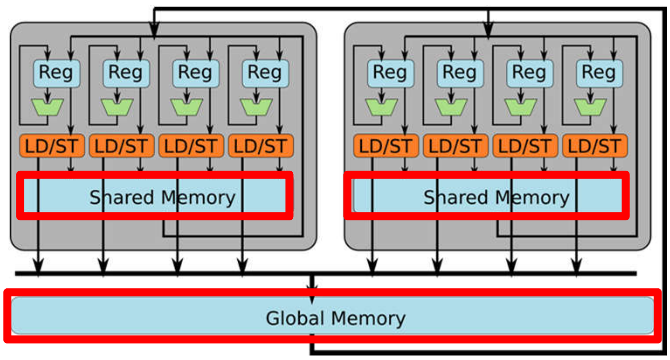
```
// Device code
__global__ void test(int *result)
{
    int tidx, bidx;
    tidx = threadIdx.x;        //x-coordinate of thread
    bidx = blockIdx.x;         //x-coordinate of block

    result[THREAD_SIZE * bidx + tidx] = (bidx + 2) * (tidx + 1);
}
```

One more: Utilizing Memory Hierarchy

Memory access latency

several cycles

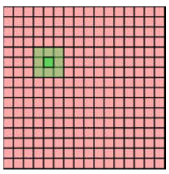
100+ cycles



참조: http://www.es.ele.tue.nl/~heco/courses/EmbeddedComputerArchitecture/

Example: Average Filters

Average over a 3x3 window for a 16x16 array



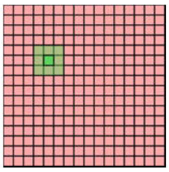
```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
    i = threadIdx.y;
    j = threadIdx.x;
    tmp = (A[i-1][j-1] + A[i-1][j] +
              … + A[i+1][i+1] ) / 9;
          A[i][j] = tmp;
}
```

Each thread loads 9 elements from global memory.
It takes hundreds of cycles.

참조: http://www.es.ele.tue.nl/~heco/courses/EmbeddedComputerArchitecture/

Utilizing the Shared Memory

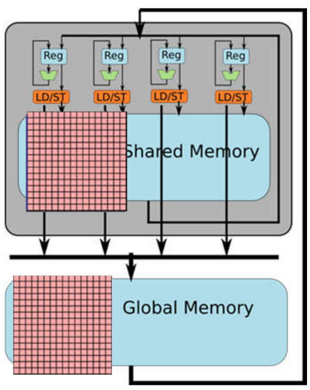
Average over a 3x3 window for a 16x16 array



```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
   __shared__ smem[16][16];
   i = threadIdx.y;
   j = threadIdx.x;
   smem[i][j] = A[i][j]; // load to smem
   A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
              … + smem[i+1][i+1] ) / 9;
}
```

참조: http://www.es.ele.tue.nl/~heco/courses/EmbeddedComputerArchitecture/

Utilizing the Shared Memory




```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
   __shared__ smem[16][16];
   i = threadIdx.y;
   j = threadIdx.x;
   smem[i][j] = A[i][j]; // load to smem
   A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
              … + smem[i+1][i+1] ) / 9;
}
```

allocate shared mem

Each thread loads one element from global memory.

참조: http://www.es.ele.tue.nl/~heco/courses/EmbeddedComputerArchitecture/

However, the Program Is Incorrect

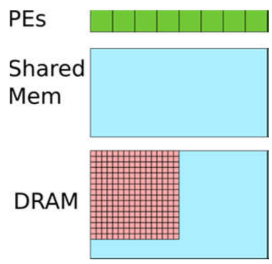
Hazards!

```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j]; // load to smem
    A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
                … + smem[i+1][i+1] ) / 9;
}
```

참조: http://www.es.ele.tue.nl/~heco/courses/EmbeddedComputerArchitecture/

Let's See What's Wrong

Assume 256 threads are scheduled on 8 PEs.

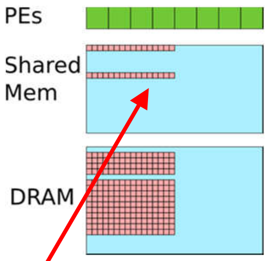



```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;        Before load instruction
    smem[i][j] = A[i][j]; // load to smem
    A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
                … + smem[i+1][i+1] ) / 9;
}
```

참조: http://www.es.ele.tue.nl/~heco/courses/EmbeddedComputerArchitecture/

Let's See What's Wrong

Assume 256 threads are scheduled on 8 PEs.



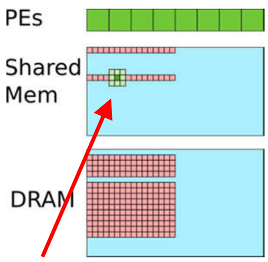
Some threads finish the load earlier than others.

```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j]; // load to smem
    A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
                … + smem[i+1][i+1] ) / 9;
}
```

참조: http://www.es.ele.tue.nl/~heco/courses/EmbeddedComputerArchitecture/

Let's See What's Wrong

Assume 256 threads are scheduled on 8 PEs.



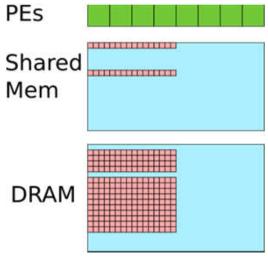
Some elements in the window are not yet loaded by other threads. Error!

```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j]; // load to smem
    A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
                … + smem[i+1][i+1] ) / 9;
}
```

참조: http://www.es.ele.tue.nl/~heco/courses/EmbeddedComputerArchitecture/

How To Solve It?

Assume 256 threads are scheduled on 8 PEs.

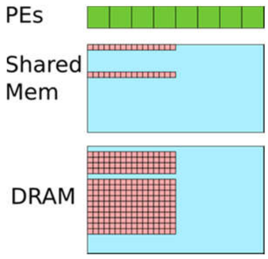



```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j]; // load to smem
    A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
                  … + smem[i+1][i+1] ) / 9;
}
```

참조: http://www.es.ele.tue.nl/~heco/courses/EmbeddedComputerArchitecture/

Use a "SYNC" barrier

Assume 256 threads are scheduled on 8 PEs.

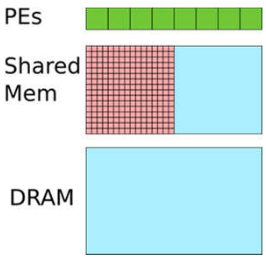



```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j]; // load to smem
    __SYNC();
    A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
                  … + smem[i+1][i+1] ) / 9;
}
```

참조: http://www.es.ele.tue.nl/~heco/courses/EmbeddedComputerArchitecture/

Use a "SYNC" barrier

Assume 256 threads are scheduled on 8 PEs.



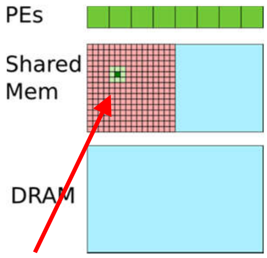

```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j]; // load to smem
    __SYNC();
    A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
                … + smem[i+1][i+1] ) / 9;
}
```

Wait until all threads hit barrier

참조: http://www.es.ele.tue.nl/~heco/courses/EmbeddedComputerArchitecture/

Use a "SYNC" barrier

Assume 256 threads are scheduled on 8 PEs.



All elements in the window are loaded when each thread starts averaging.

```
kernelF<<<(1,1),(16,16)>>>(A);
__device__    kernelF(A){
    __shared__ smem[16][16];
    i = threadIdx.y;
    j = threadIdx.x;
    smem[i][j] = A[i][j]; // load to smem
    __SYNC();
    A[i][j] = ( smem[i-1][j-1] + smem[i-1][j] +
                … + smem[i+1][i+1] ) / 9;
}
```

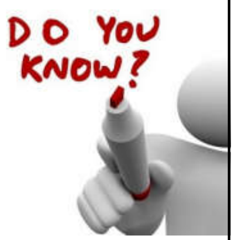
참조: http://www.es.ele.tue.nl/~heco/courses/EmbeddedComputerArchitecture/

Simple Profile

- http://docs.nvidia.com/cuda/profiler-users-guide/index.html#summary-mode
- https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-nvprof-your-handy-universal-gpu-profiler/

차례

- **Introduction**
  - Multicore/Manycore and GPU
  - GPU on Medical Applications
- **Parallel Programming on GPUs: Basics**
  - Conceptual Introduction
- **GPU Architecture Review**
- **Parallel Programming on GPUs: Practice**
  - Real programming
- **Conclusion**

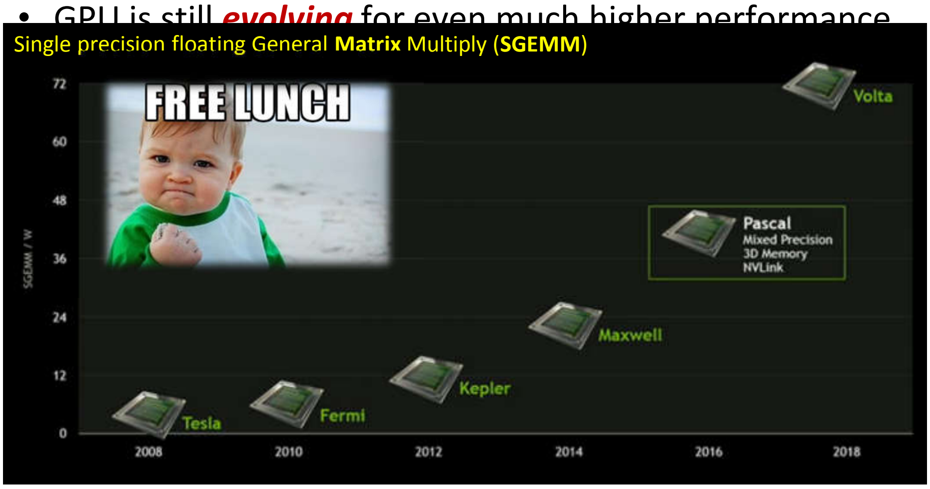


Conclusion

- GPU is still ***evolving*** for even much higher performance
  - **Stacked DRAM**
  - **NVLINK**
- *Free Lunch* ?
  - GPU continue to improve performance and power with more advanced GPU hardware/software features
- Mobile / portable medical system & Car system
  - **Tegra K1**/X1/X2, Jetson Nano …

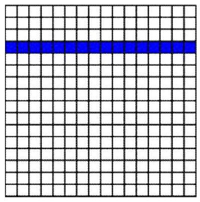
Conclusion

- GPU is still ***evolving*** for even much higher performance

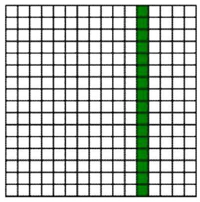


Final Questions: Matrix Multiplication

A



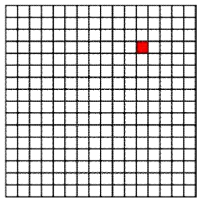
B



```
void main(){

  define A, B, C
  for i = 0 to M do
    for j = 0 to N do
      /* compute element C(i,j) */
      for k = 0 to K do
            C(i,j) <= C(i,j) + A(i,k) * B(k,j)
      end
    end
  end
}
```

C

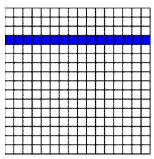
(i,j)



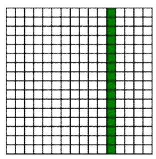
Final Questions: Matrix Multiplication

```
main:
  define A_cpu, B_cpu, C_cpu in the CPU memory;
  define A_gpu, B_gpu, C_gpu in the GPU memory;
  memcopy A_cpu to A_gpu;    memcopy B_cpu to B_gpu;
  dim3 dimBlock(16, 16);    dim3 dimGrid(N/dimBlock.x, M/dimBlock.y);
...

__global__ void matrixMul(A_gpu,B_gpu,C_gpu,K){
   temp <= 0



   C_gpu(i,j) <= accu
}
```

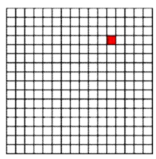
A



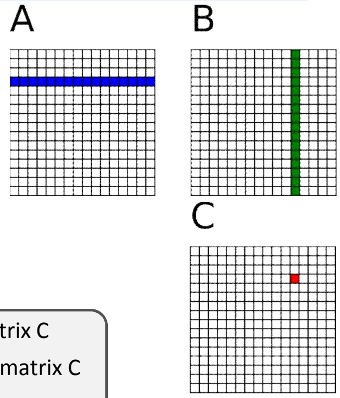
B



C



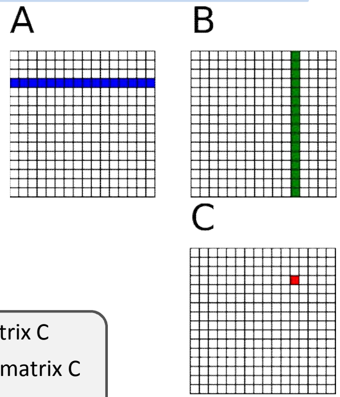
Final Questions: Matrix Multiplication



main:

```
define A_cpu, B_cpu, C_cpu in the CPU memory;
define A_gpu, B_gpu, C_gpu in the GPU memory;
memcopy A_cpu to A_gpu;    memcopy B_cpu to B_gpu;
dim3 dimBlock(16, 16);    dim3 dimGrid(N/dimBlock.x, M/dimBlock.y);
...

__global__ void matrixMul(A_gpu,B_gpu,C_gpu,K){
  temp <= 0
  i <= blockIdx.y * blockDim.y + threadIdx.y    // Row i of matrix C
  j <= blockIdx.x * blockDim.x + threadIdx.x    // Column j of matrix C
  for k = 0 to K-1 do
     accu <= accu + A_gpu(i,k) * B_gpu(k,j)
  end

  C_gpu(i,j) <= accu
}
```

Final Questions: Matrix Multiplication



main:

```
define A_cpu, B_cpu, C_cpu in the CPU memory;
define A_gpu, B_gpu, C_gpu in the GPU memory;
memcopy A_cpu to A_gpu;    memcopy B_cpu to B_gpu;
dim3 dimBlock(16, 16);    dim3 dimGrid(N/dimBlock.x, M/dimBlock.y);
...

__global__ void matrixMul(A_gpu,B_gpu,C_gpu,K){
  temp <= 0
  i <= blockIdx.y * blockDim.y + threadIdx.y    // Row i of matrix C
  j <= blockIdx.x * blockDim.x + threadIdx.x    // Column j of matrix C
  for k = 0 to K-1 do
     accu <= accu + A_gpu(i,k) * B_gpu(k,j)
  end

  C_gpu(i,j) <= accu
}
```

Thanks !