

# CS 677: Parallel Programming for Many-core Processors

## Lecture 13

Instructor: Philippos Mordohai

Webpage: [mordohai.github.io](http://mordohai.github.io)

E-mail: [Philippos.Mordohai@stevens.edu](mailto:Philippos.Mordohai@stevens.edu)

# Outline

- Deep learning
- Hardware Developments
- Developments in CUDA

# Deep Learning

# Machine Learning

- A way of building software from input-output pairs
  - Use labeled data - data that come with the input values and their desired output values - to learn what the logic should be
  - Capture each labeled data item by adjusting the program logic
- Training Phase
  - The system learns the logic for the application from labeled data.
- Deployment (inference) Phase
  - The system applies the learned program logic on new data

View deep neural network as function approximators

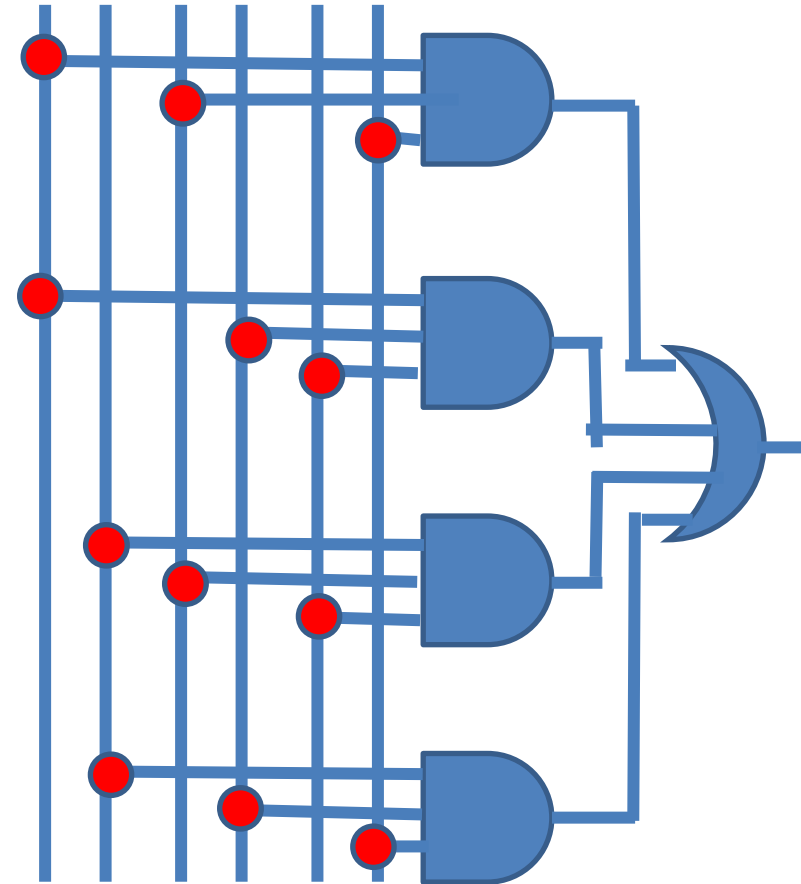
# Recent Explosion of Deep Learning Applications

- GPU computing hardware and programming interfaces such as CUDA has enabled very fast research cycle of deep neural net training
- Computer Vision, Speech Recognition, Document Translation, Self Driving Cars, ...
- Using big labeled data to train and specialize DNN based classifiers

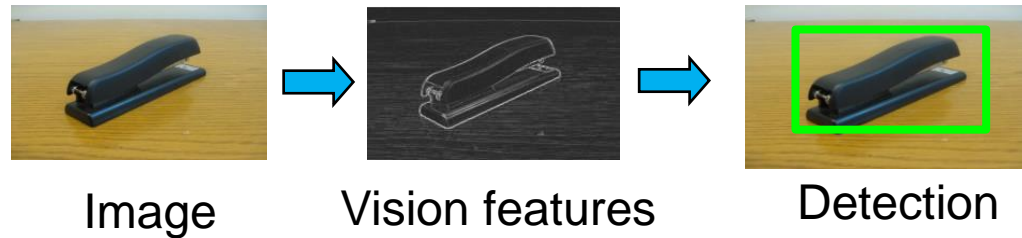
# Background: Combinations Logic Specification - Truth Table

a' a b' b c' c

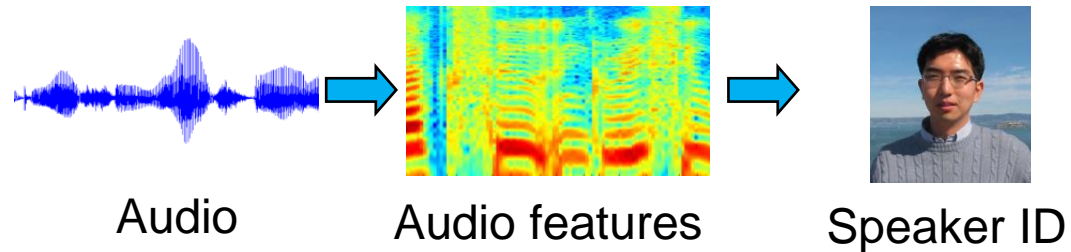
Input			output
a	b	c	
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1



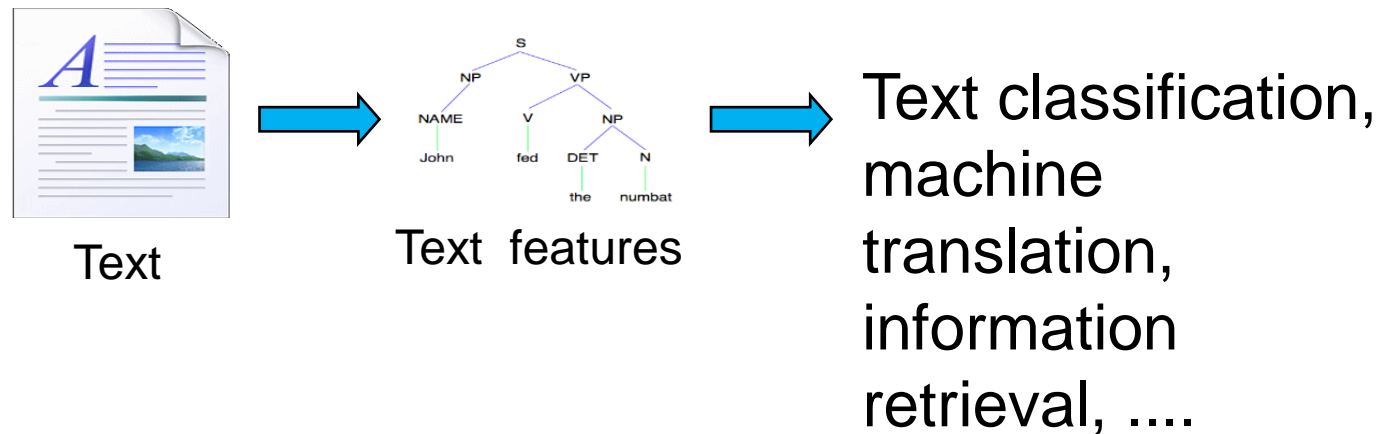
Images/video  
0



Audio



Text



# What if we did not know the truth table?

- Look at enough observation data to construct the rule

000  $\rightarrow$  0

011  $\rightarrow$  0

100  $\rightarrow$  1

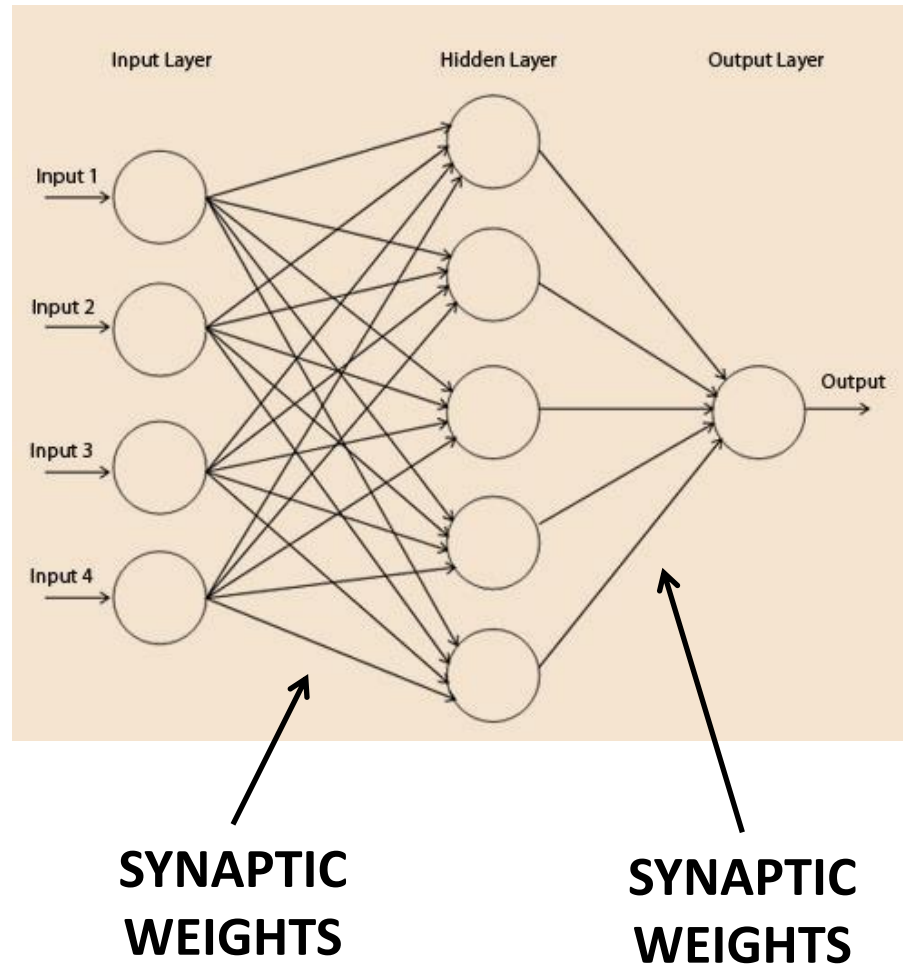
110  $\rightarrow$  0

- If we have enough observational data to cover all input patterns, we can construct the truth table and derive the logic

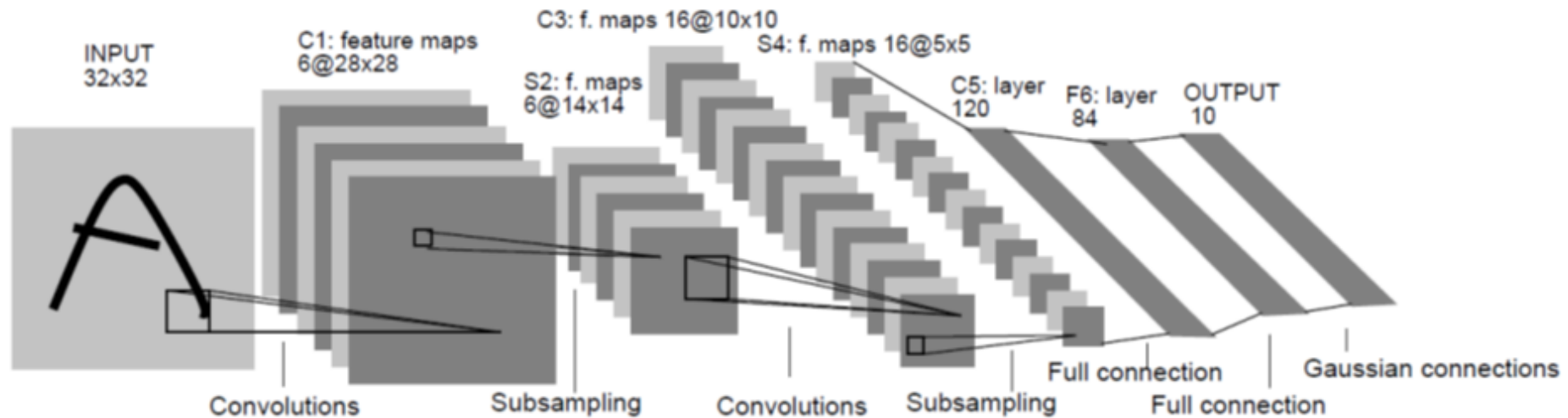


# Multilayer Perceptron Synaptic Weights

Universal for function approximation

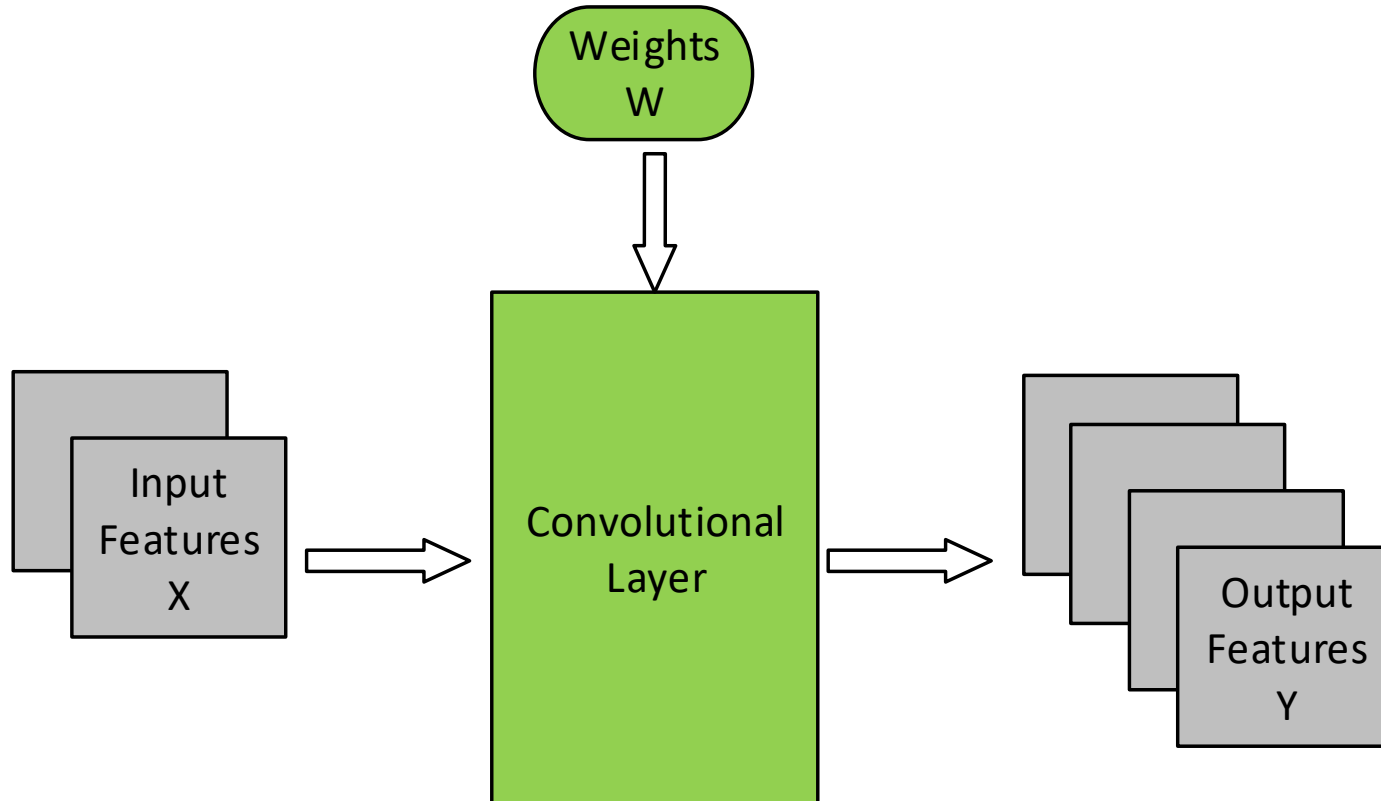


# LeNet-5, a convolutional neural network for hand-written digit recognition



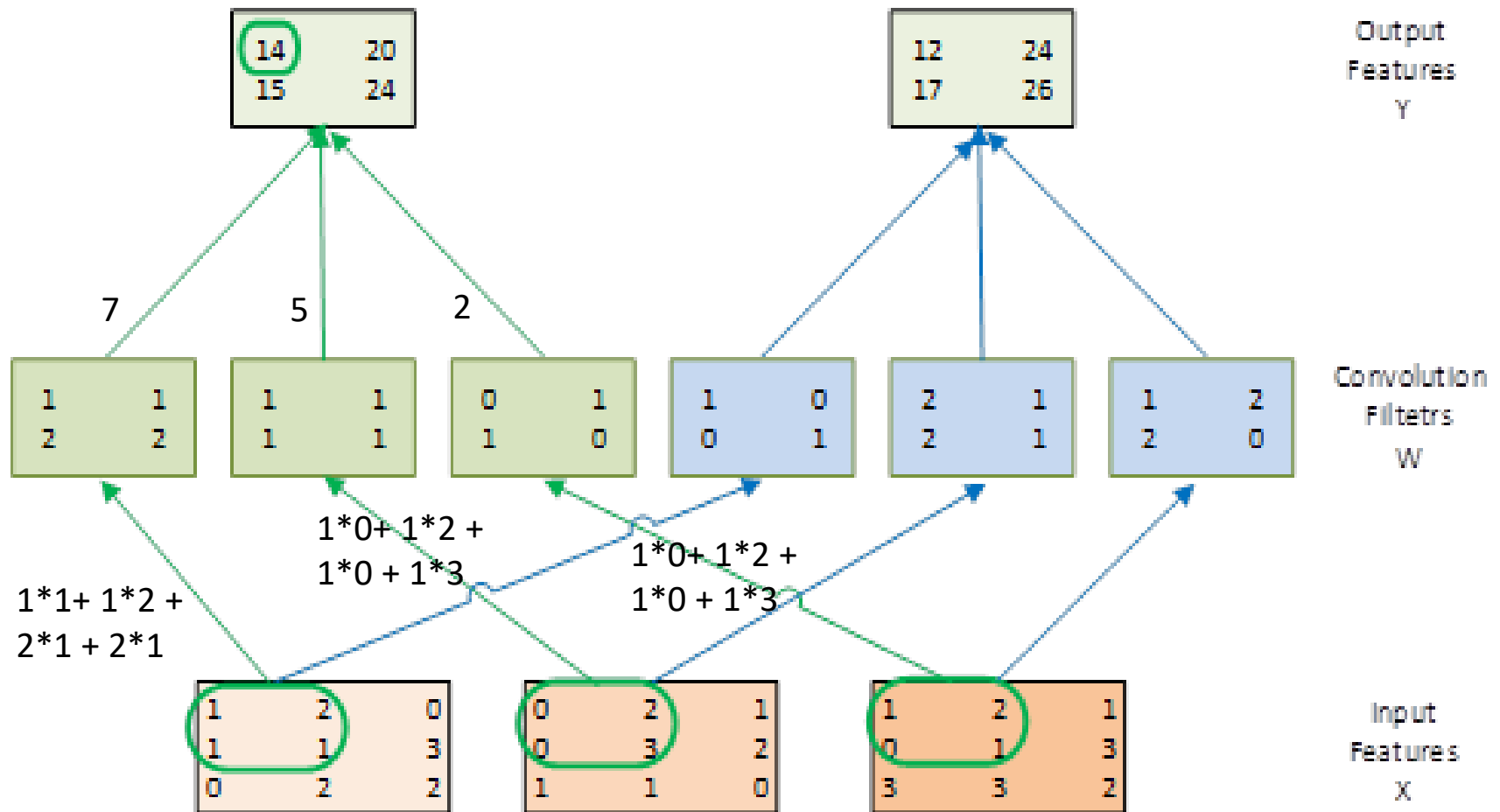
This is a  $1024 \times 8$ -bit input, which will have a truth table of  $2^{8196}$  entries

# Forward Propagation Path of a Convolution Layer



- All input feature maps contribute to all output feature maps. One convolution mask is provided for each input-output combination.

# Example of the Forward Path of a Convolution Layer



# Sequential Code for the Forward Path of a Convolution Layer

```
void convLayer_forward(int M, int C, int H, int W, int K, float* X, float* W, float* Y)
{
    int m, c, h, w, p, q;
    int H_out = H - K + 1;
    int W_out = W - K + 1;

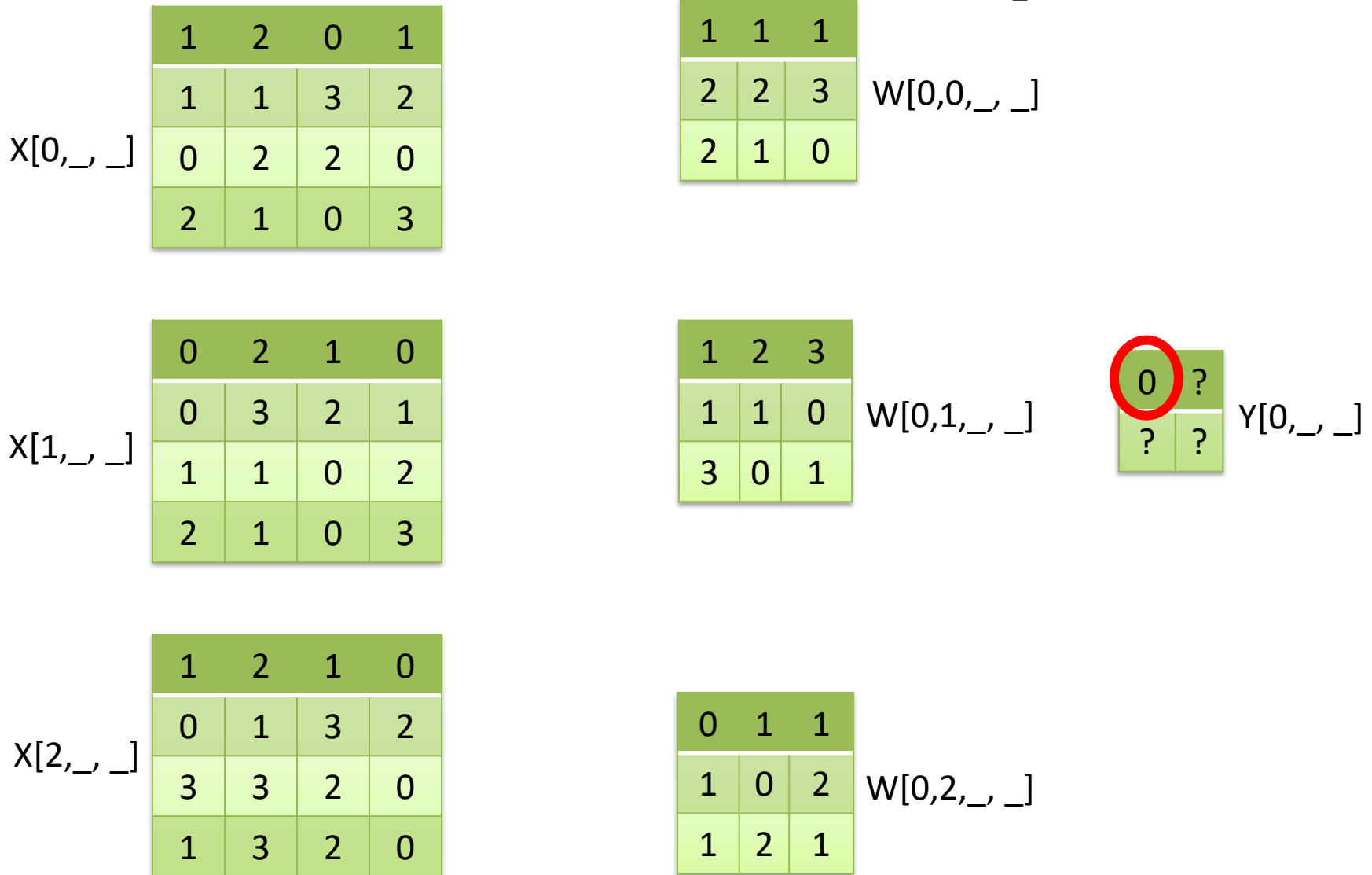
    for(int m = 0; m < M; m++)                // for each output feature map
        for(int h = 0; h < H_out; h++)        // for each output element
            for(int w = 0; w < W_out; w++) {
                Y[m, h, w] = 0;
                for(int c = 0; c < C; c++)      // sum over all input feature maps
                    for(int p = 0; p < K; p++)  // KxK filter
                        for(int q = 0; q < K; q++)
                            Y[m, h, w] += X[c, h + p, w + q] * W[m, c, p, q];
            }
}
```

# Sequential code for the Forward Path of a Sub-sampling Layer

```
void poolingLayer_forward(int M, int H, int W, int K, float* Y, float* S)
{
    for(int m = 0; m < M; m++)                // for each output feature maps
        for(int h = 0; h < H/K; h++)           // for each output element
            for(int w = 0; w < W/K; w++) {
                S[m, x, y] = 0.;
                for(int p = 0; p < K; p++) {     // loop over KxK input samples
                    for(int q = 0; q < K; q++)
                        S[m, h, w] += Y[m, K*h + p, K*w + q] / (K*K);
                }
                // add bias and apply non-linear activation
                S[m, h, w] = sigmoid(S[m, h, w] + b[m])
            }
}
```

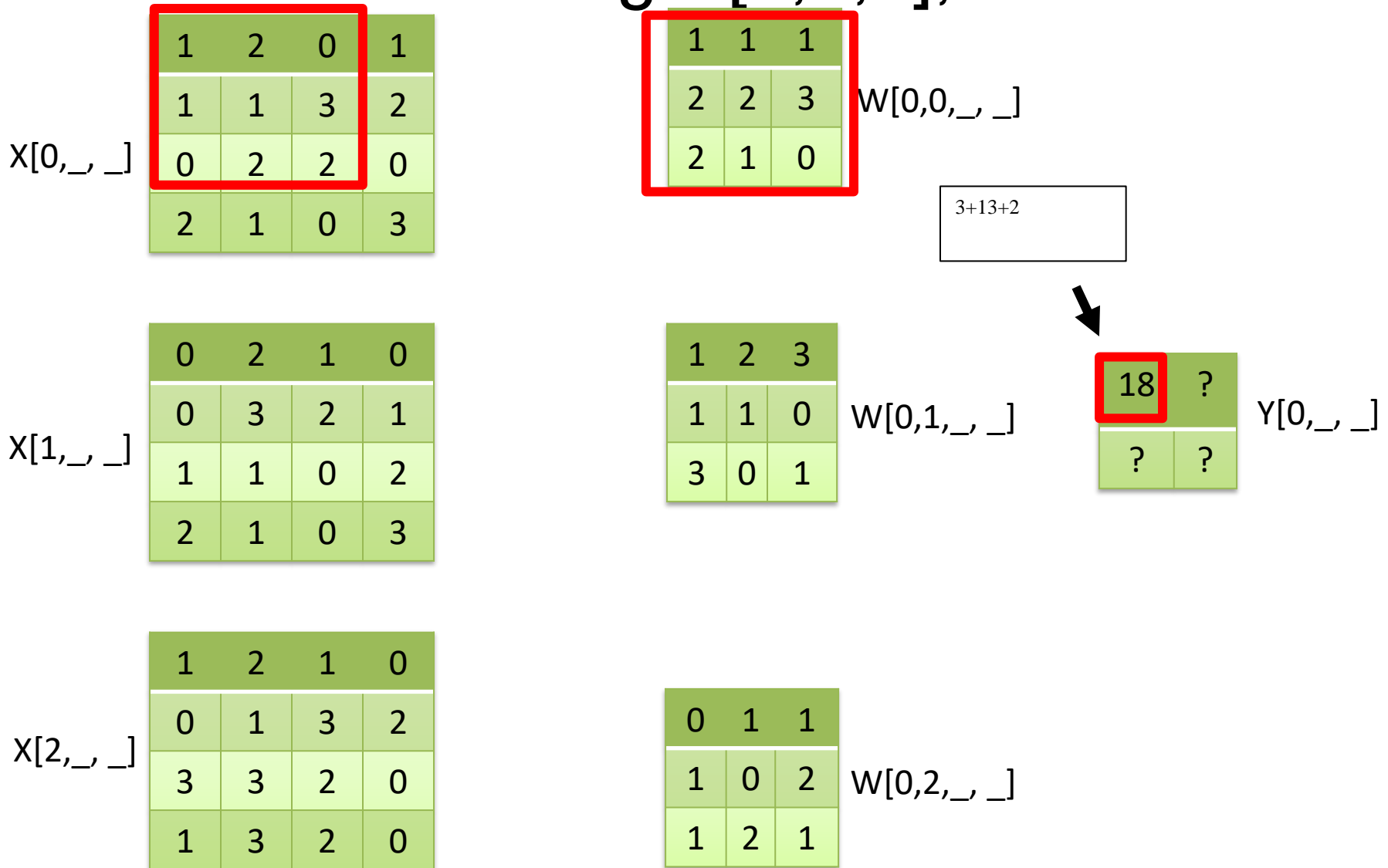
# A Small Convolution Layer Example

## Generating $Y[0,0,1]$



# A Small Convolution Layer Example

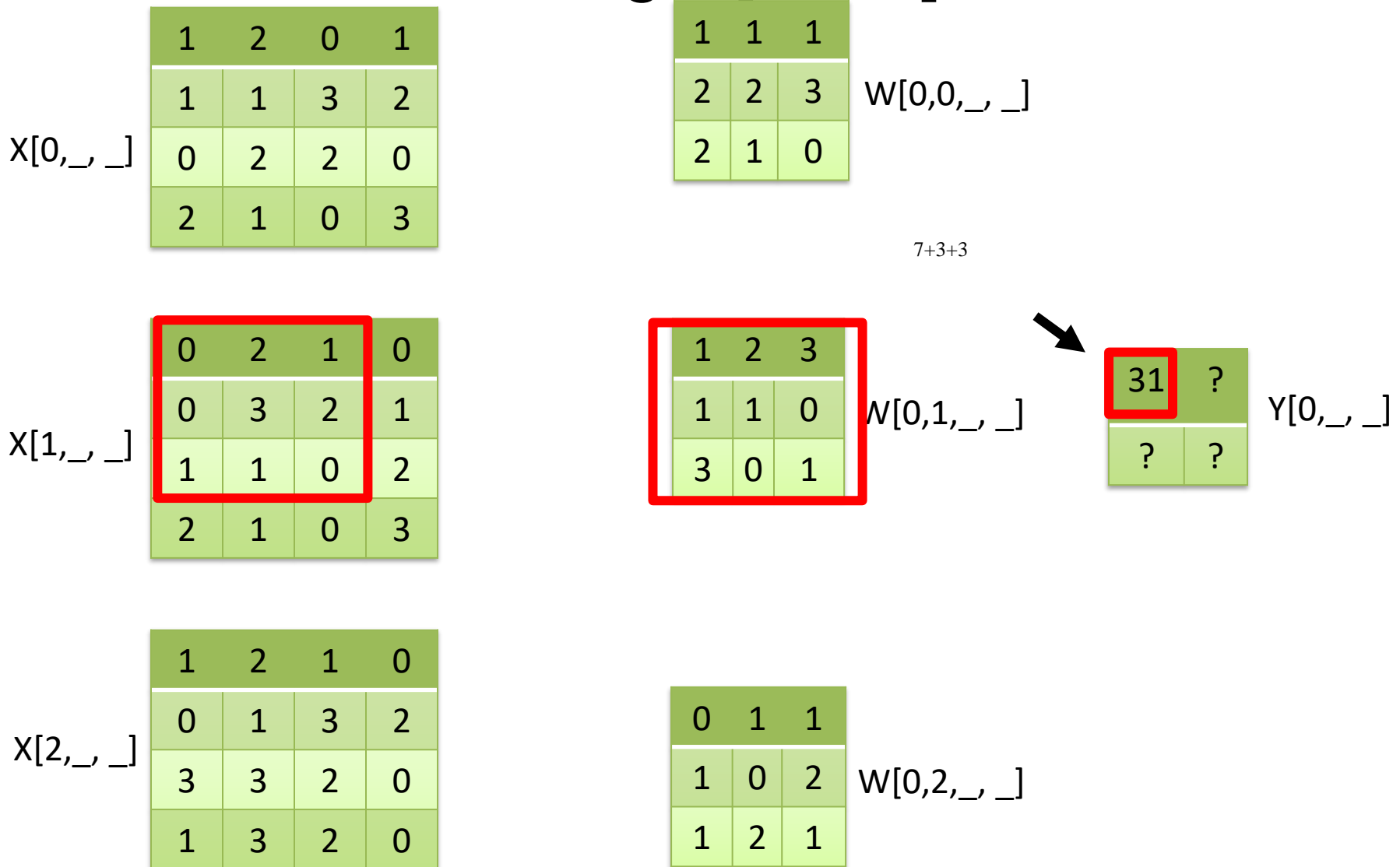
## Generating $Y[0,0,0]$ , $c=0$





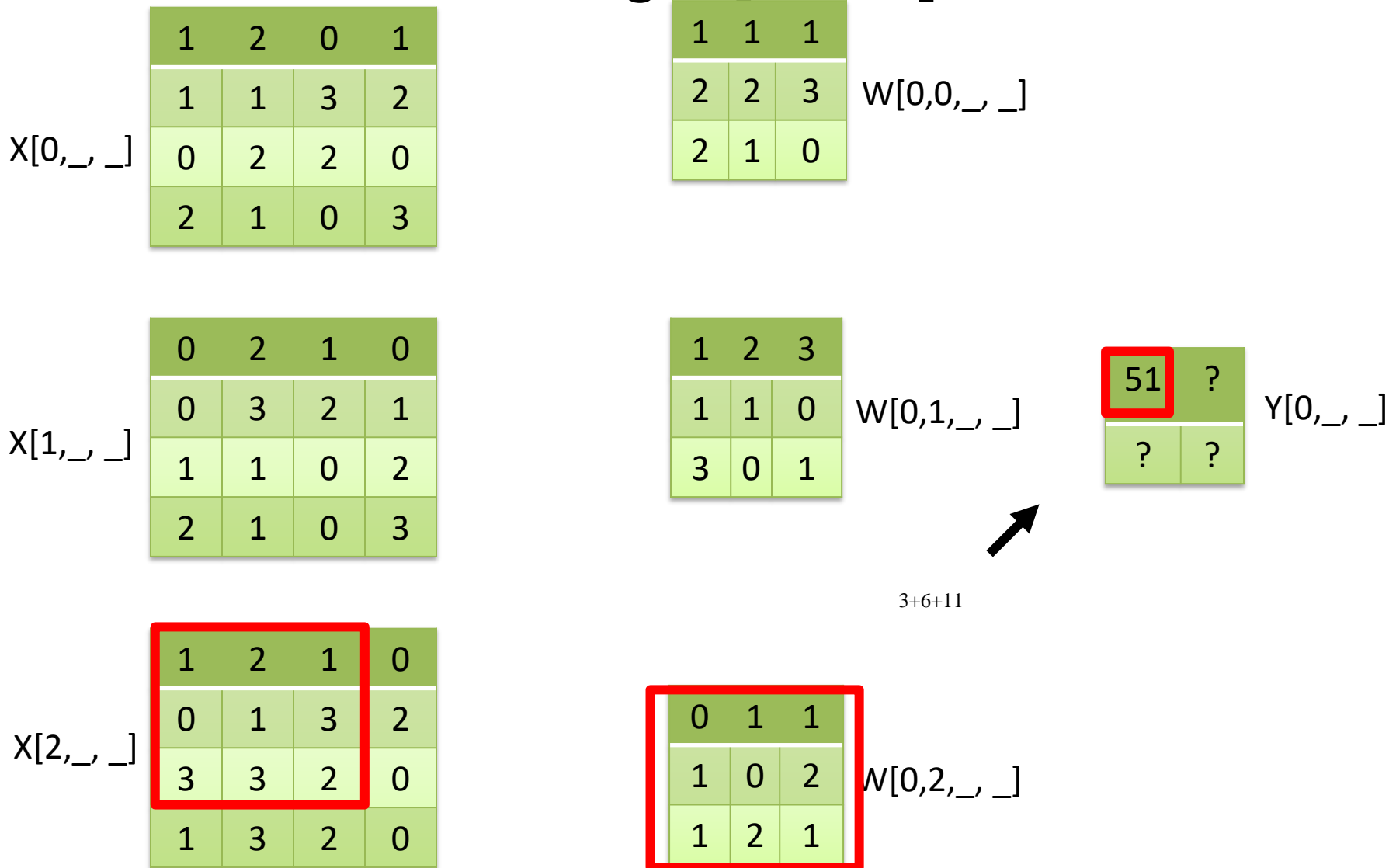
# A Small Convolution Layer Example

## Generating $Y[0,0,0]$ , $c=1$



# A Small Convolution Layer Example

## Generating $Y[0,0,0]$ , $c=2$



# Parallelism in a Convolution Layer

- All output feature maps can be calculated in parallel
  - A small number in general, not sufficient to fully utilize a GPU
- All output feature map pixels can be calculated in parallel
  - All rows can be done in parallel
  - All pixels in each row can be done in parallel
  - Large number but diminishes as we go into deeper layers
- All input feature maps can be processed in parallel, but will need atomic operation or tree reduction

# Design of a Basic Kernel

- Each block computes a tile of output pixels
  - `TILE_WIDTH` pixels in each dimension
- The first (x) dimension in the grid maps to the M output feature maps
- The second (y) dimension in the grid maps to the tiles in the output feature maps

# Host Code for the Basic Kernel

- Defining the grid configuration
  - $W_{out}$  and  $H_{out}$  are the output feature map width and height

```
# define TILE_WIDTH 16          // We will use 4 for small examples.
W_grid = W_out/TILE_WIDTH;      // number of horizontal tiles per output map
H_grid = H_out/TILE_WIDTH;      // number of vertical tiles per output map
Y = H_grid * W_grid;
dim3 blockDim(TILE_WIDTH, TILE_WIDTH, 1);
dim3 gridDim(M, Y, 1);
ConvLayerForward_Kernel<<< gridDim, blockDim>>>(...);
```

# A Small Example

- Assume that we will produce 4 output feature maps
  - Each output feature map is 8x8 image
  - We have 4 blocks in the x dimension
- If we use tiles of 4 pixels on each side (TILE\_SIZE = 4)
  - We have 4 blocks in the x dimension
    - Top two blocks in each column calculate the top row of tiles in the corresponding output feature map
    - Bottom two blocks in each column calculate the bottom row of tiles in the corresponding output feature map

# A Basic Conv. Layer Forward Kernel

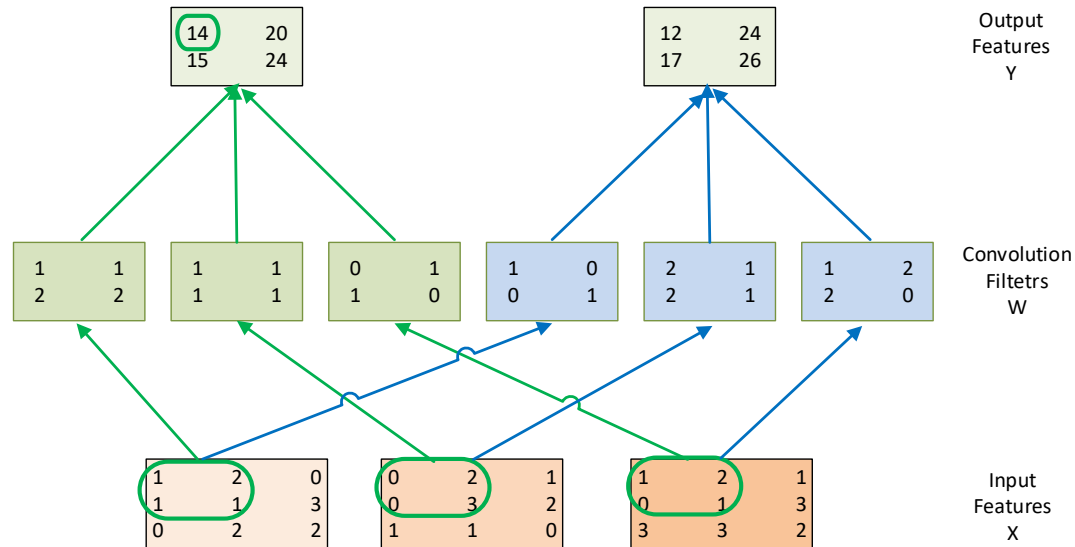
```
__global__ void ConvLayerForward_Basic_Kernel(int C, int W_grid, int K,
        float* X, float* W, float* Y)
{
    int m = blockIdx.x;
    int h = blockIdx.y / W_grid + threadIdx.y;
    int w = blockIdx.y % W_grid + threadIdx.x;
    float acc = 0.;
    for (int c = 0; c < C; c++) {                // sum over all input channels
        for (int p = 0; p < K; p++)                // loop over KxK filter
            for (int q = 0; q < K; q++)
                acc += X[c, h + p, w + q] * W[m, c, p, q];
    }
    Y[m, h, w] = acc;
}
```

# Some Observations

- The amount of parallelism is quite high as long as the total number of pixels across all output feature maps is large
  - This matches the CNN architecture well
- Each input tile is loaded multiple times, once for each block that calculates the output tile that requires it
  - Not very efficient in global memory bandwidth



# Implementing a convolution layer with matrix multiplication



$$\begin{array}{|c|c|c|c|} \hline 1 & 1 & 2 & 2 \\ \hline 1 & 0 & 0 & 1 \\ \hline \end{array} \quad
 \begin{array}{|c|c|c|c|} \hline 1 & 1 & 1 & 1 \\ \hline 2 & 1 & 2 & 1 \\ \hline \end{array} \quad
 \begin{array}{|c|c|c|c|} \hline 0 & 1 & 1 & 0 \\ \hline 1 & 2 & 2 & 0 \\ \hline \end{array}
 \quad * \quad
 \begin{array}{|c|c|c|c|} \hline 1 & 2 & 1 & 1 \\ \hline 2 & 0 & 1 & 3 \\ \hline 1 & 1 & 0 & 2 \\ \hline 1 & 3 & 2 & 2 \\ \hline 0 & 2 & 0 & 3 \\ \hline 2 & 1 & 3 & 2 \\ \hline 0 & 3 & 1 & 1 \\ \hline 3 & 2 & 1 & 0 \\ \hline 1 & 2 & 1 & 1 \\ \hline 2 & 1 & 0 & 3 \\ \hline 0 & 1 & 3 & 3 \\ \hline 1 & 3 & 3 & 2 \\ \hline \end{array}
 \quad = \quad
 \begin{array}{|c|c|c|c|} \hline 14 & 20 & 15 & 24 \\ \hline 12 & 24 & 17 & 26 \\ \hline \end{array}$$

Convolution Filters  $W'$

Input Features  $X_{\text{unrolled}}$

Output Features  $Y$

# Simple Matrix Multiplication

Each product matrix element is an output feature map pixel.

This inner product generates element 0 of output feature map 0.

Convolution Filters

0	1	1	2	2	1	1	1	1	0	1	1	0
1	1	0	0	1	2	1	2	1	1	2	2	0

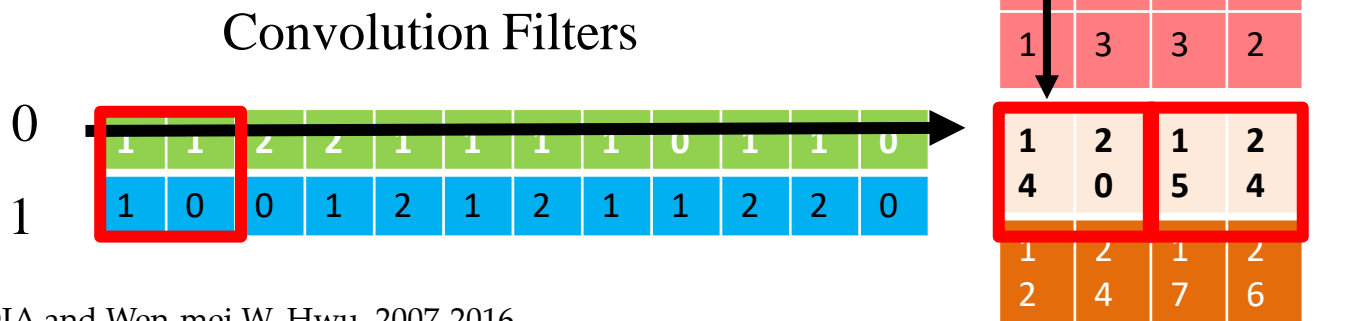
1	2	1	1	0	Input feature maps
2	0	1	3		
1	1	0	2		
1	3	2	2		
0	2	0	3	1	
2	1	3	2		
0	3	1	1		
3	2	1	0		
1	2	1	1	2	
2	1	0	3		
0	1	3	3		
1	3	3	2		
1	2	1	2		
4	0	5	4		
1	2	1	2		
2	4	7	6		

# Tiled Matrix Multiplication

## 2x2 example

Each block calculates one output tile – 2 elements from each output map

Each input element is reused 2 times in the shared memory

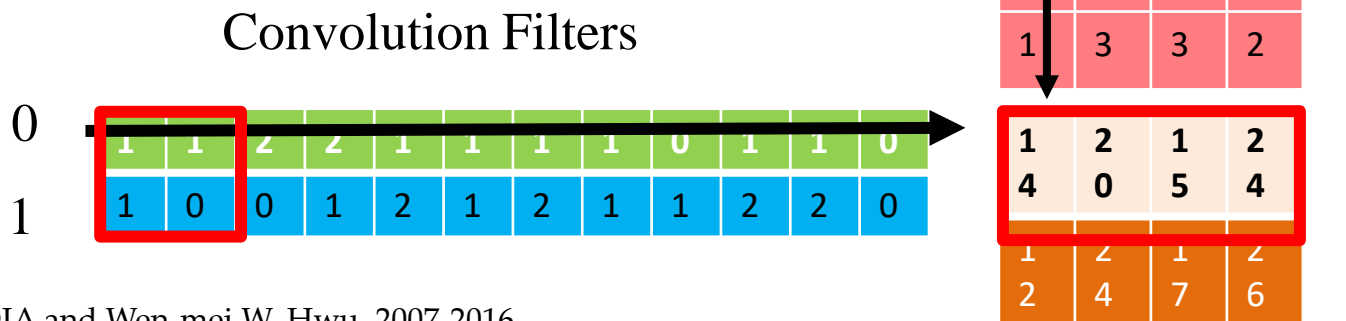


# Tiled Matrix Multiplication

## 2x4 example

Each block calculates one output tile – 4 elements from each output map

Each input element is reused 2 times in the shared memory



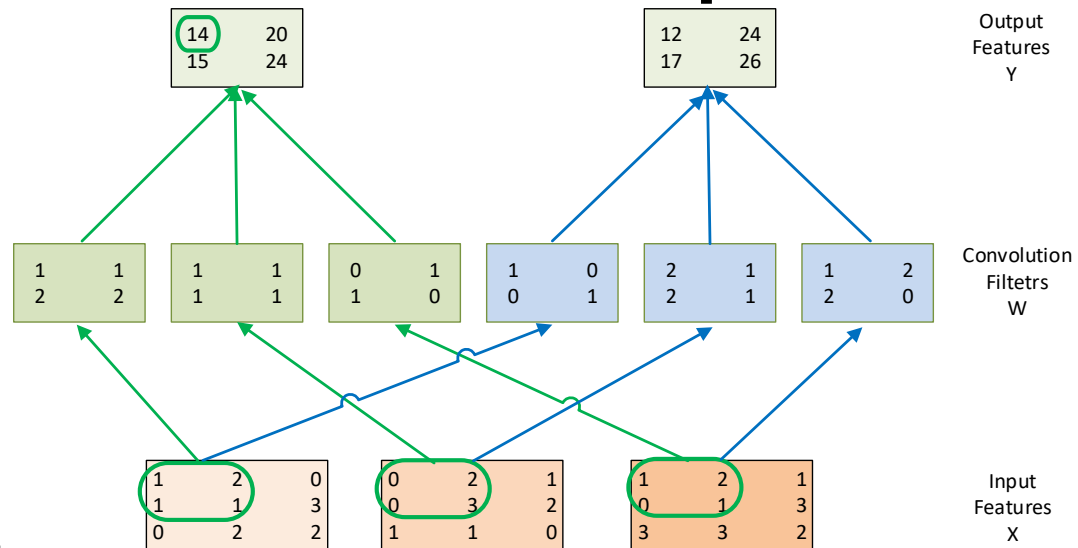
# Analysis of Efficiency

## Total Input Replication

- Each output map requires its replicated input feature map elements
  - Not replicated for different output feature maps
  - There are  $H_{out} * W_{out}$  output feature map elements
  - Each requires  $K*K$  replicated input feature map elements
  - So, the total number of input element after replication is  $H_{out}*W_{out}*K*K$  times for each input feature map
  - The total number of elements in each original input feature map is  $(H_{out}+K-1) * (W_{out}+K-1)$

# Analysis of Small Example

- $H_{out} = 2$
- $W_{out} = 2$
- $K = 2$
- There are 3 input maps (channels)
- The total number of input elements in the replicated (“unrolled”) input matrix is  $3*2*2*2*2$
- The replicating factor is  $(3*2*2*2*2)/(3*3*3) = 1.78$



1	1	2	2
1	0	0	1

1	1	1	1
2	1	2	1

0	1	1	0
1	2	2	0

\*

1	2	1	1
2	0	1	3
1	1	0	2
1	3	2	2
0	2	0	3
2	1	3	2
0	3	1	1
3	2	1	0
1	2	1	1
2	1	0	3
0	1	3	3
1	3	3	2

=

14	20	15	24
12	24	17	26

Convolution Filtrrs W'      Input Features X\_unrolled      Output Features Y

# Memory Access Efficiency of Original Convolution Algorithm

- Assume that we use tiled 2D convolution
- For input elements
  - Each output tile has  $\text{TILE\_WIDTH}^2$  elements
  - Each input tile has  $(\text{TILE\_WIDTH}+K-1)^2$
  - The total number of input feature map element accesses is  $\text{TILE\_WIDTH}^2 * K^2$
  - The reduction factor of the tiled algorithm is  $K^2 * \text{TILE\_WIDTH}^2 / (\text{TILE\_WIDTH}+K-1)^2$
- The convolution filter weight elements are reused within each output tile

# Efficiency of Tiled Matrix Multiplication

- Assuming we use  $\text{TILE\_WIDTH}^2$  input and output tiles
  - Each replicated input feature map element is reused  $\text{TILE\_WIDTH}$  times
  - Each convolution filter weight element is reused  $\text{TILE\_WIDTH}$  times
  - Matrix multiplication is better if  $\text{TILE\_WIDTH}$  is larger than  $K^2 \cdot \text{TILE\_WIDTH}^2 / (\text{TILE\_WIDTH} + K - 1)^2$



# Problem with the Later Stages

- The size ( $H_{out}$ ,  $W_{out}$ ) of each output feature map decreases as we go to the later stages of the CNN
  - The `TILE_WIDTH` may be limited to very small sizes relative to  $K$
  - The benefit of 2D tiling will diminish as we go down the pipeline
  - This is an intrinsic problem for 2D tiled convolution

# Mini-Batching

- One can use mini-batching to further increase the amount of work done in each kernel launch
  - Collect several sets of input feature maps of an input sequence
  - Use a larger unrolled input feature matrix that has all the inputs from the mini-batch

# Other Optimizations

- Use streams to overlap the reading of the next set of input feature maps with the processing of the previous input feature maps.
- Create unrolled matrix elements on the fly, only when they are loaded into shared memory
- Use more advanced algorithms such as FFT to implement convolution

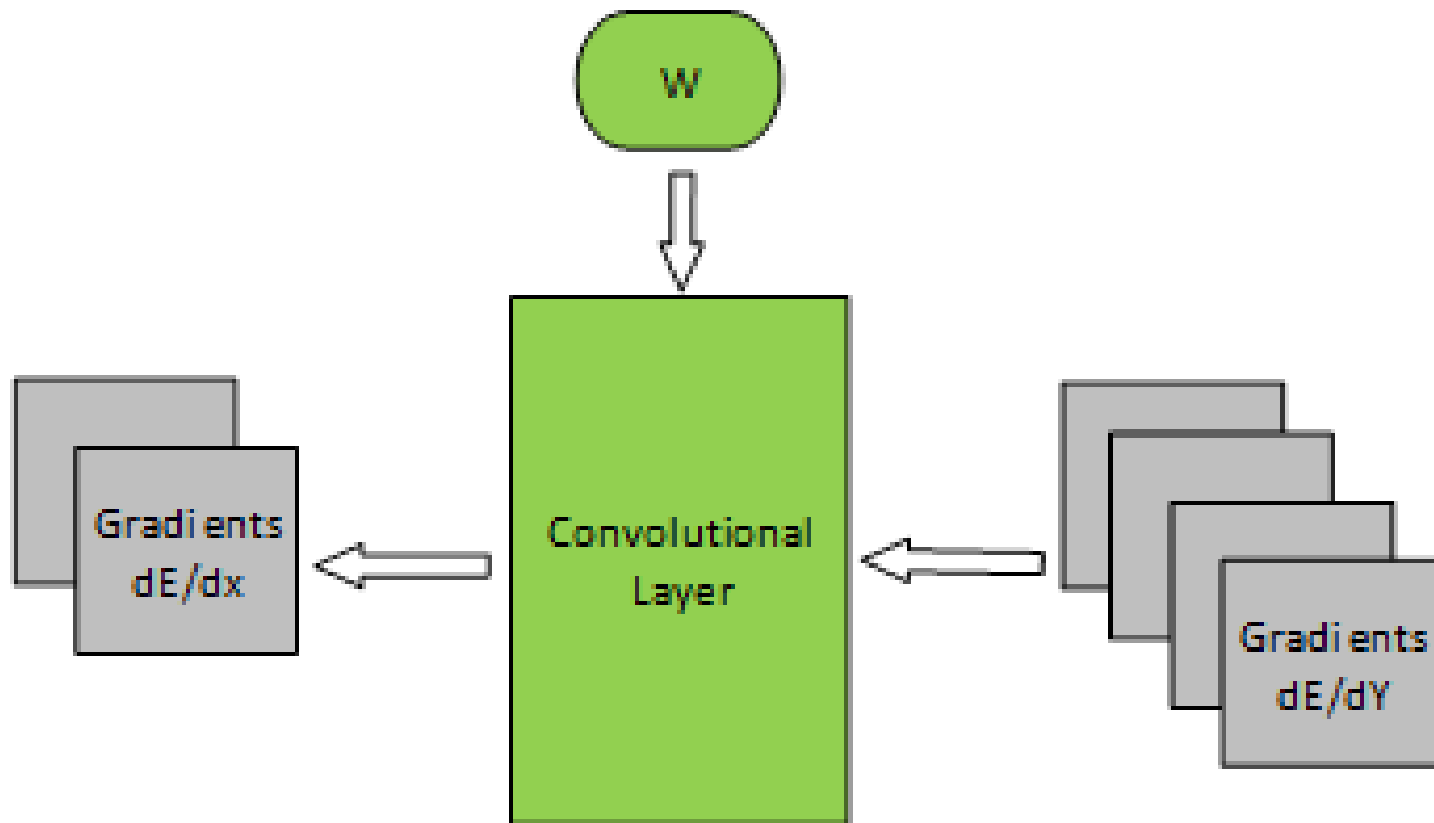
# Gradient Back-Propagation

- Training of ConvNets is based on a procedure called back-propagation.
- The training data set is labeled with the “correct answer.”
- For each training image, the final stage of the network calculates the loss function or the error as the difference between the generated output vector element values and the “correct” output vector element values.
- Given a sequence of training images, we can numerically calculate the gradient of the loss function with respect to the output vector. Intuitively, it gives the rate at which the error changes when the value of the output vector changes -  $dE/dY$

# Gradient Back Propagation (Cont.)

- The process propagates the gradient from the last layer towards the first layer through all layers of network.
- Each layer receives as  $dE/dY$  - gradient with respect to its output feature maps and computes  $dE/dX$  - gradient with respect to its input feature maps

# Convolution Layer - Back Propagation of $dE/dY$



# Adjusting Weights

- After the  $dE/dW$  values at all feature map element positions are computed, weights are updated:
- For each weight value
$$w(t+1) = w(t) - \lambda / dE/dw,$$

where  $\lambda$  is the learning rate.

# Other layer types

- Fully-connected
- Pooling/downsampling
- Upsampling
- Activation

The same operation is applied on all inputs, with same or different parameters (weights)



# The Fermi Architecture

Selected notes from  
presentation by:

Michael C. Shebanow

Principal Research Scientist,  
NV Research  
[mshebanow@nvidia.com](mailto:mshebanow@nvidia.com)

(2010)

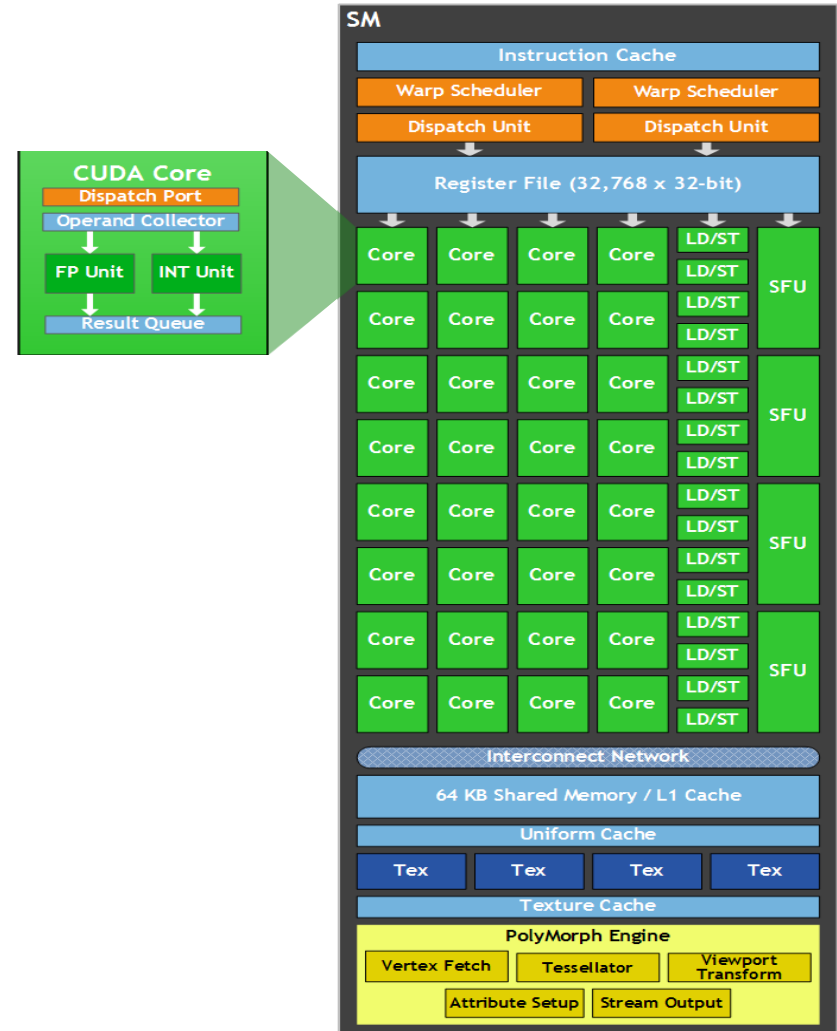
# Much Better Compute

- Programmability
  - C++ Support
  - Exceptions/Debug support
- Performance
  - Dual issue SMs
  - L1 cache
  - Larger Shared Memory
  - Much better DP math
  - Much better atomic support
- Reliability: ECC

	GT200	GF100	Benefit
L1 Texture Cache (per quad)	12 KB	12 KB	Fast texture filtering
Dedicated L1 LD/ST Cache	X	16 or 48 KB	Efficient physics and ray tracing
Total Shared Memory	16KB	16 or 48 KB	More data reuse among threads
L2 Cache	256KB (TEX read only)	768 KB (all clients read/write)	Greater texture coverage, robust compute performance
Double Precision Throughput	30 FMAs/clock	256 FMAs/clock	Much higher throughputs for Scientific codes

# Instruction Set Architecture

- Enables C++ : virtual functions, new/delete, try/catch
- Unified load/store addressing
- 64-bit addressing for large problems
- Optimized for CUDA C, OpenCL & Direct Compute
  - Direct Compute is Microsoft's general-purpose computing on GPU API
- Enables system call functionality
  - stdio.h, etc.



# Unified Load/Store Addressing

## Non-unified Address Space

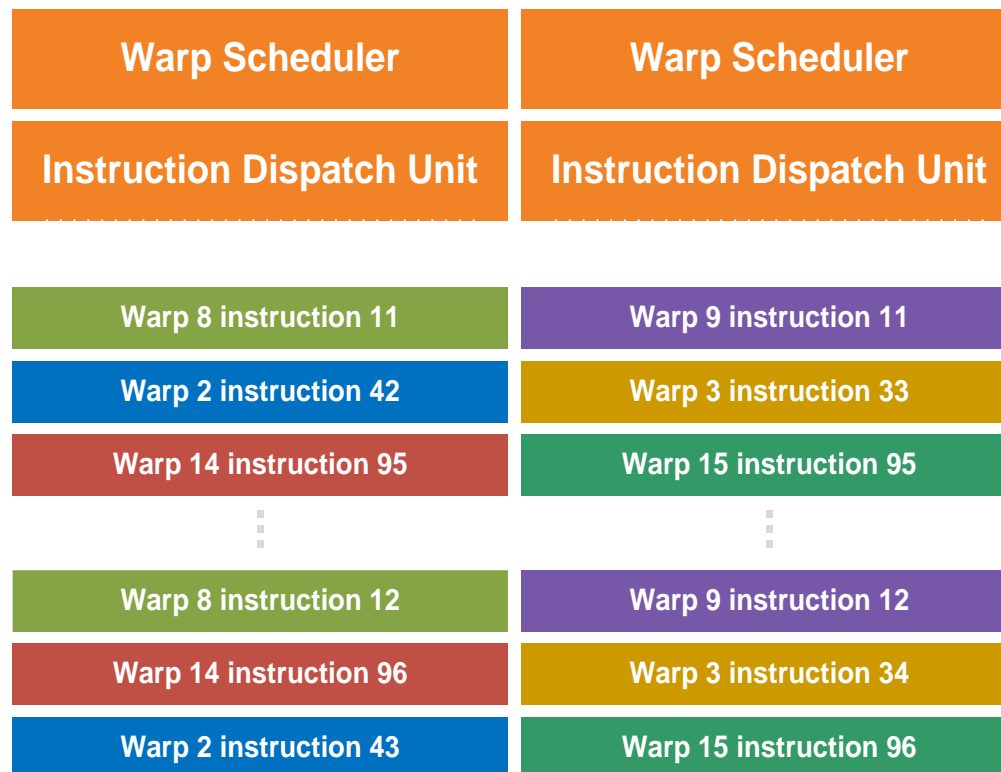


## Unified Address Space



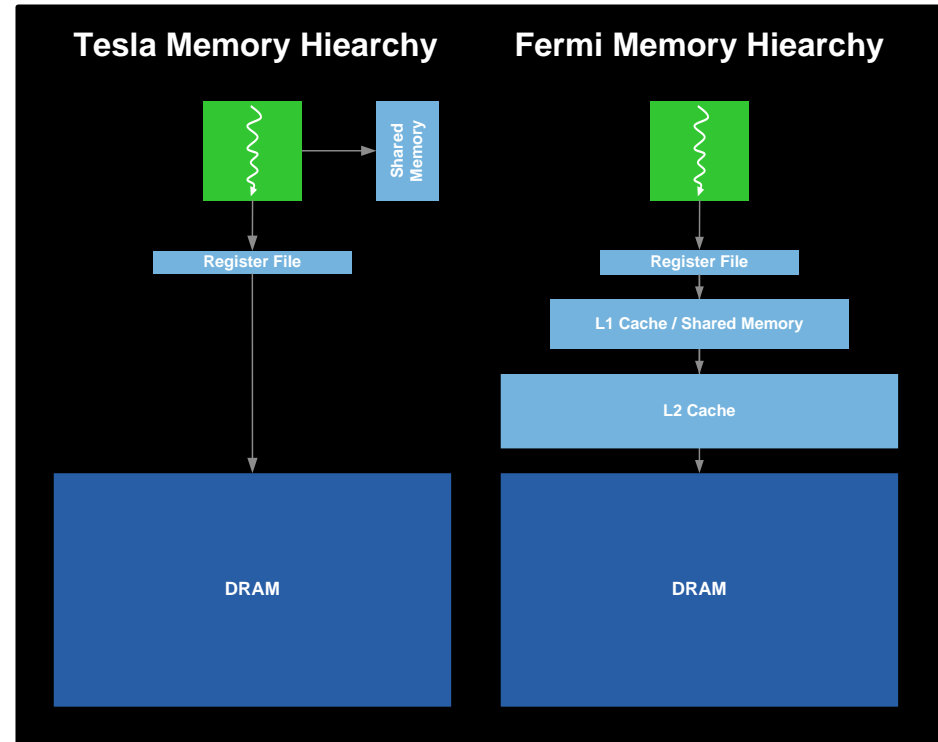
# Instruction Issue and Control Flow

- Decouple internal execution resources
  - Deliver peak IPC on branchy / int-heavy / LD-ST - heavy codes
- Dual issue pipelines select two warps to issue



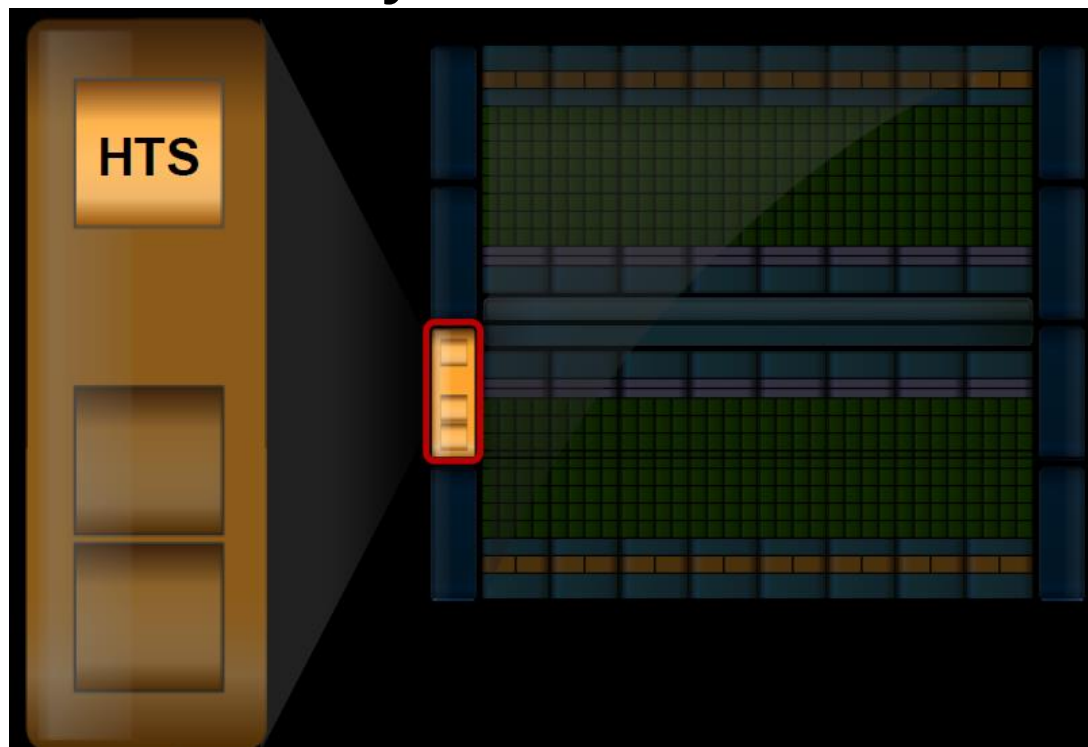
# Caches

- Configurable L1 cache per SM
  - 16KB L1\$ / 48KB Shared Memory
  - 48KB L1\$ / 16KB Shared Memory
- Shared 768KB L2 cache
- Compute motivation:
  - Caching captures locality, amplifies bandwidth
  - Caching more effective than Shared Memory for irregular or unpredictable access
    - Ray tracing, sparse matrix multiplication, physics kernels ...
  - Caching helps latency sensitive cases



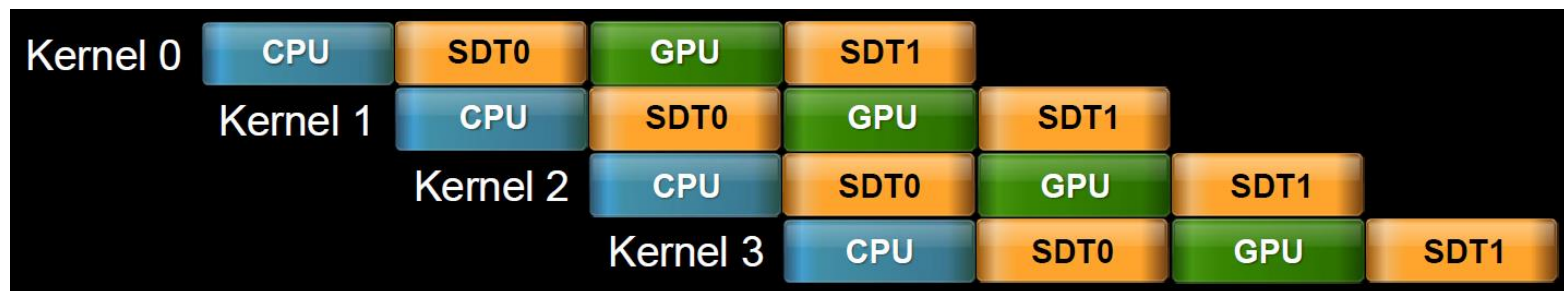
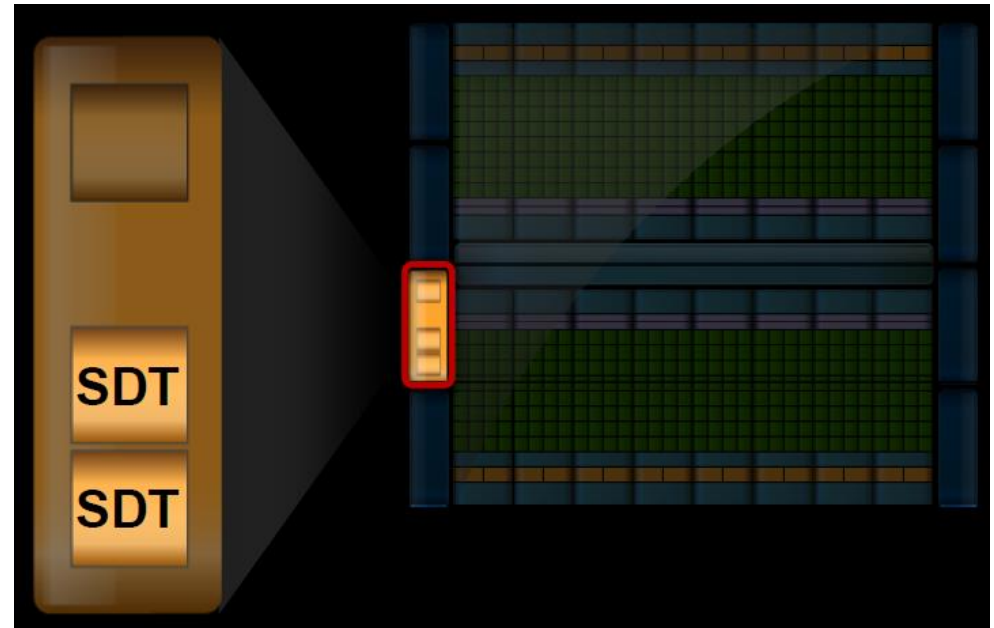
# GigaThread Hardware Thread Scheduler

- Hierarchically manages tens of thousands of simultaneously active threads
- 10x faster context switching on Fermi
- Concurrent kernel execution



# GigaThread Streaming Data Transfer Engine

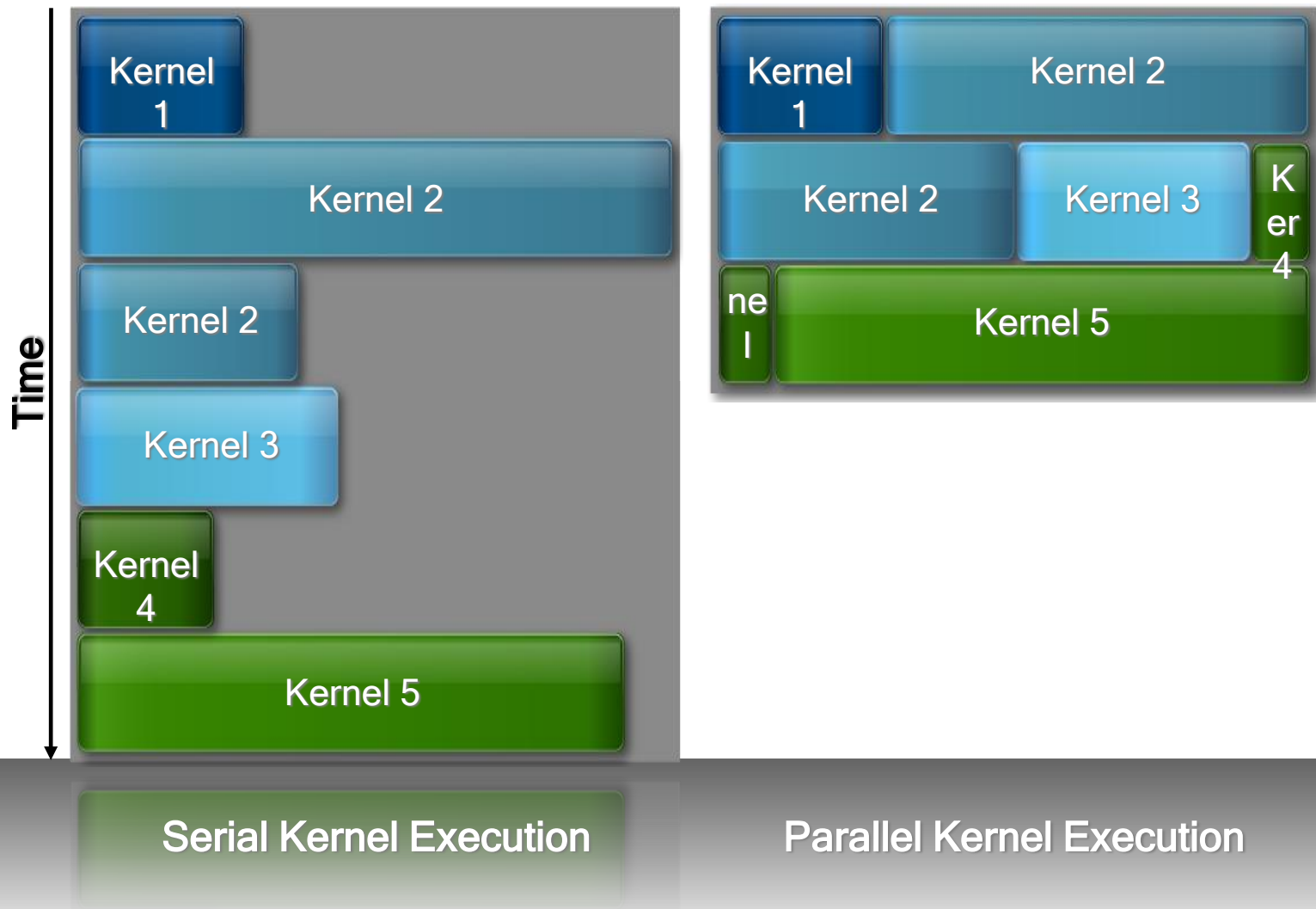
- Dual DMA engines
- Simultaneous CPU→GPU and GPU→CPU data transfer
- Fully overlapped with CPU/GPU processing





# Fermi runs independent kernels in parallel

Concurrent Kernel Execution + Faster Context Switch



# Inside Kepler

Manuel Ujaldon

Nvidia CUDA Fellow

Computer Architecture Department

University of Malaga (Spain)

Modified by P. Mordohai

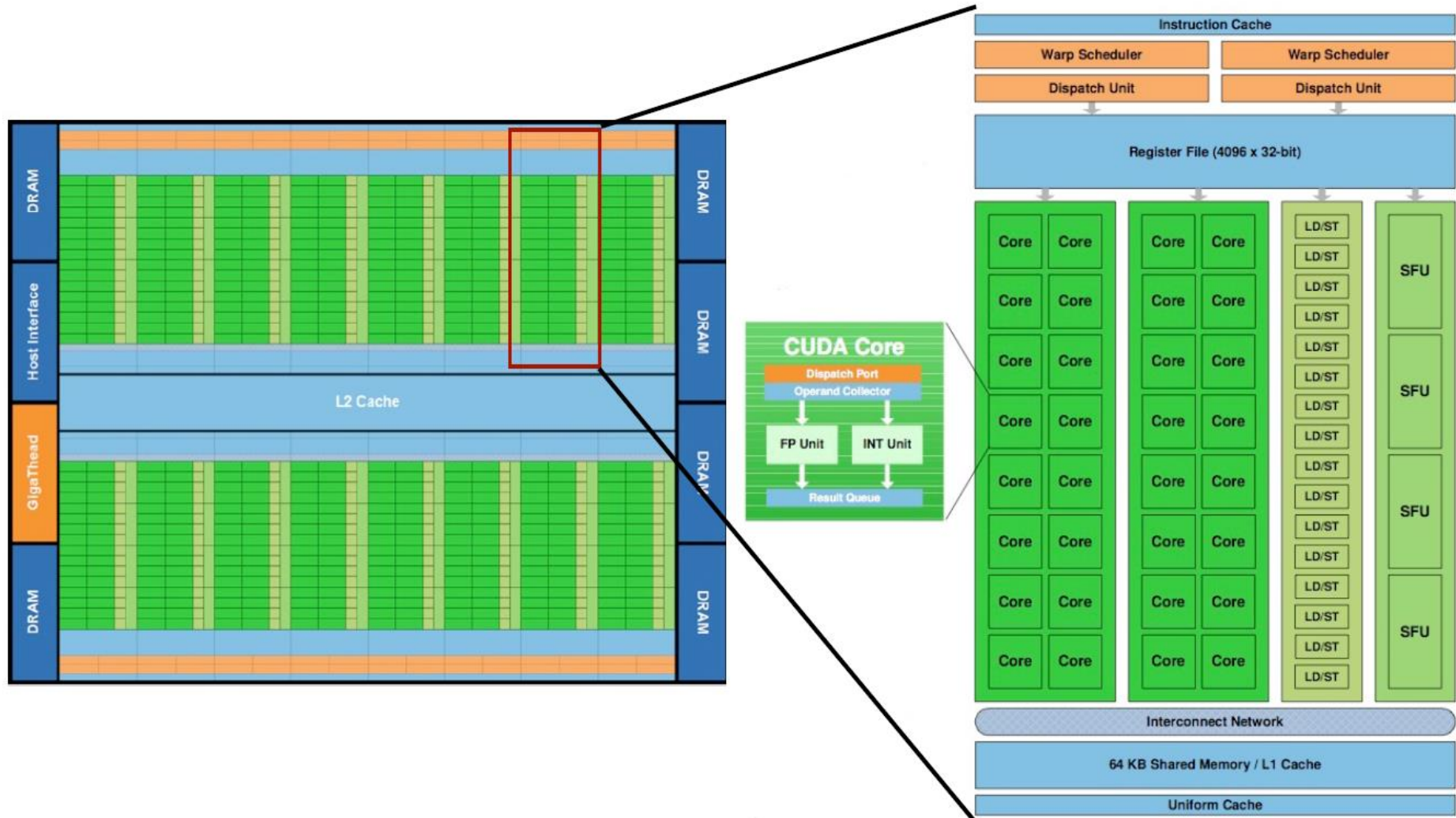
# Summary of Features

- Released in 2012
- Architecture: Between 7 and 15 multiprocessors SMX, endowed with 192 cores each.
- Arithmetic: More than 1 TeraFLOP in double precision (64 bits IEEE-754 floating-point format).
  - Specific values depend on the clock frequency for each model (usually, more on GeForce, less on Teslas).
- Major innovations in core design:
  - Dynamic parallelism
  - Thread scheduling (Hyper-Q)

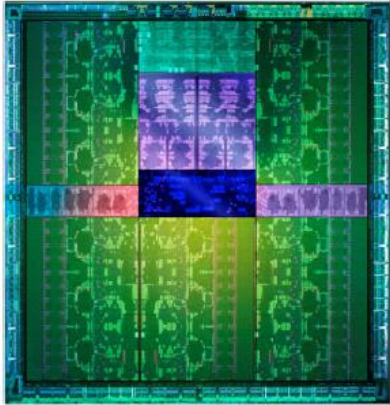
# How the Architecture Scales Up

Architecture	G80	GT200	Fermi GF100	Fermi GF104	Kepler GK104	Kepler GK110
Time frame	2006-07	2008-09	2010	2011	2012	2013
CUDA Compute Capability (CCC)	1.0	1.2	2.0	2.1	3.0	3.5
N (multiprocs.)	16	30	16	7	8	15
M (cores/multip.)	8	8	32	48	192	192
Number of cores	128	240	512	336	1536	2880

# Fermi

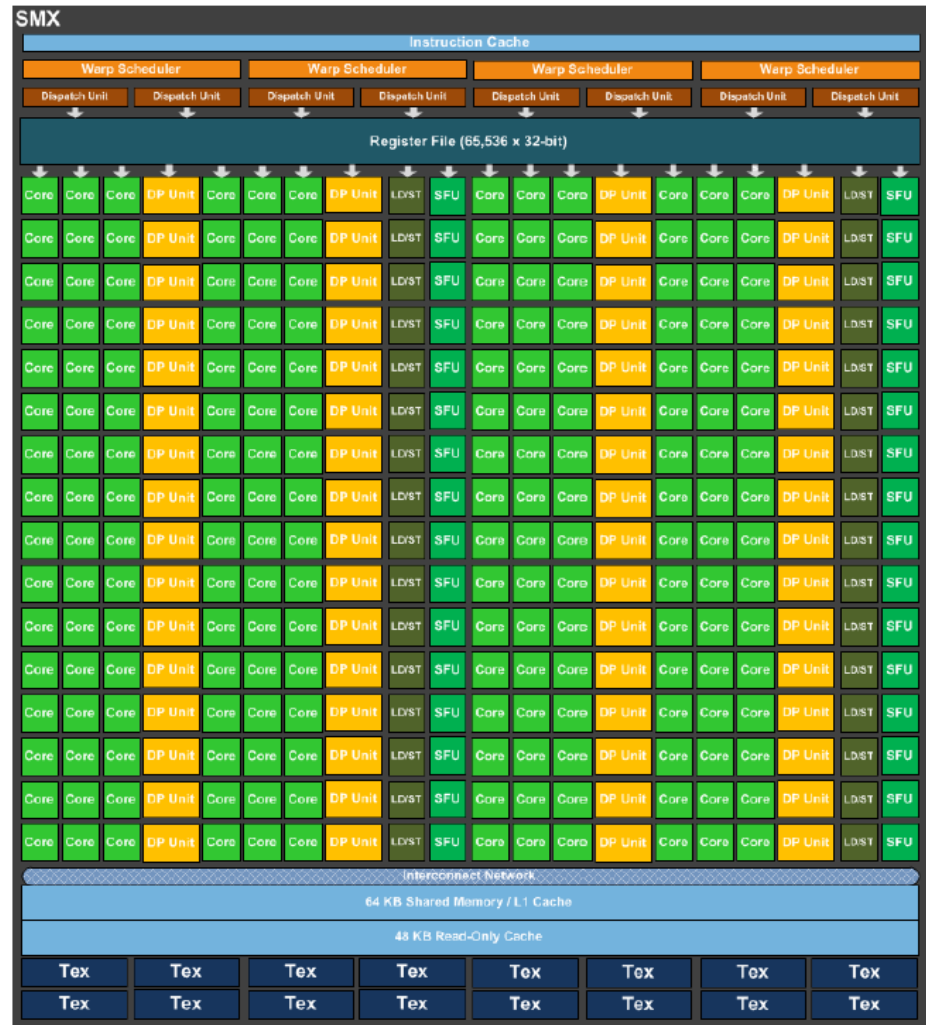
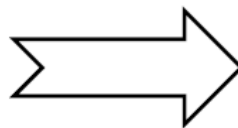
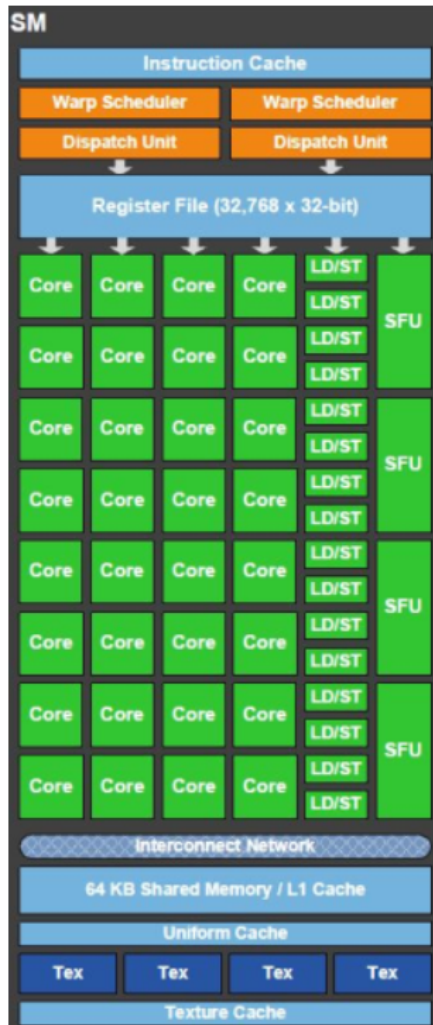


# Kepler GK110

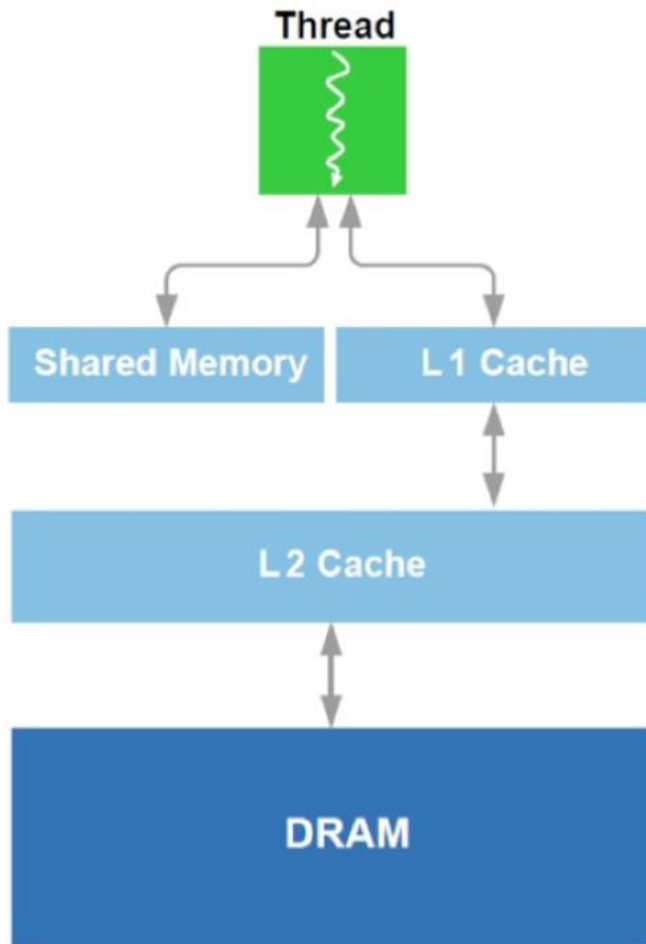




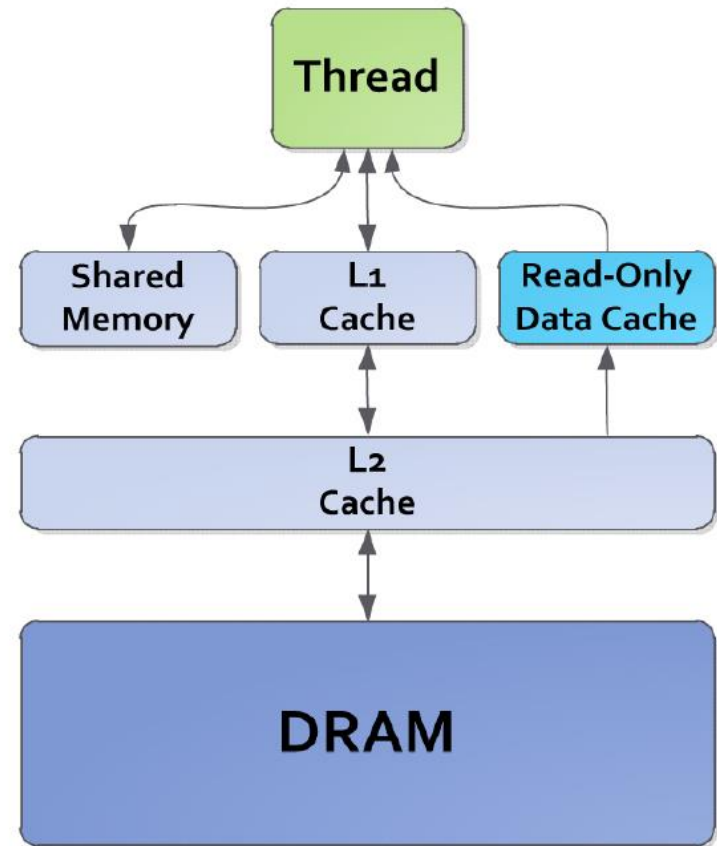
# From SM to SMX in Kepler



# Differences in Memory Hierarchy



Kepler Memory Hierarchy





# New Data Cache

- Additional 48 Kbytes to expand L1 cache size
- Avoids the texture unit
- Allows a global address to be fetched and cached, using a pipeline different from that of L1/shared
- Flexible (does not require aligned accesses)
- Eliminates texture setup
- Managed automatically by compiler ("const\_\_restrict" indicates eligibility). Next slide shows an example.

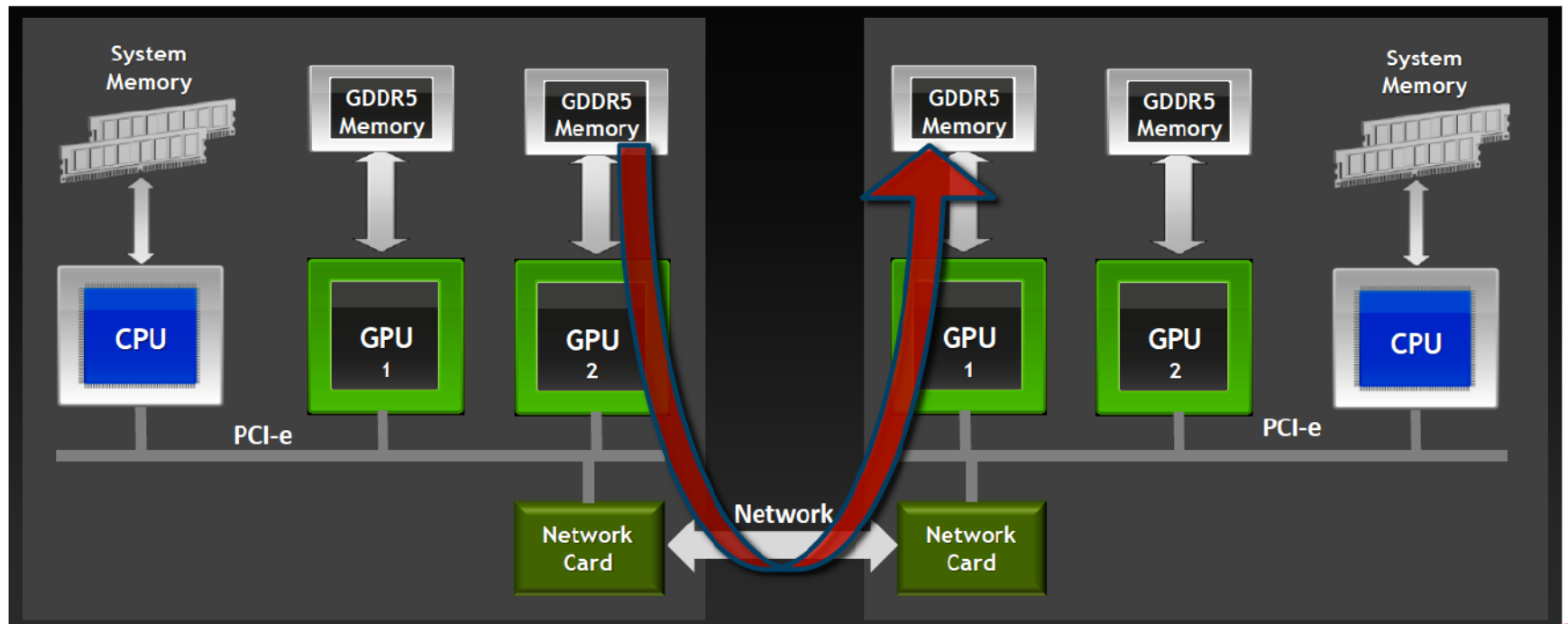
# How to use Data Cache

- Annotate eligible kernel parameters with "const \_\_restrict"
- Compiler will automatically map loads to use read-only data cache path.

```
__global__ void saxpy(float x, float y,  
    const float * __restrict input,  
    float * output)  
{  
    size_t offset = threadIdx.x +  
        (blockIdx.x * blockDim.x);  
  
    // Compiler will automatically use cache for "input"  
    output[offset] = (input[offset] * x) + y;  
}
```

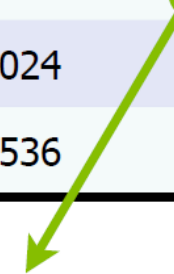
# GPUDirect now supports RDMA [Remote Direct Memory Access]

- This allows direct transfers between GPUs and network devices, for reducing the penalty on the extraordinary bandwidth of GDDR5 video memory



# Relaxing Software Constraints for Massive Parallelism

GPU generation	Fermi		Kepler	
Hardware model	GF100	GF104	GK104	GK110
CUDA Compute Capability (CCC)	2.0	2.1	3.0	3.5
Number of threads / warp (warp size)	32	32	32	32
Max. number of warps / Multiprocessor	48	48	64	64
Max. number of blocks / Multiprocessor	8	8	16	16
Max. number of threads / Block	1024	1024	1024	1024
Max. number of threads / Multiprocessor	1536	1536	2048	2048



Crucial enhancement  
for Hyper-Q (see later)

# Major Hardware Enhancements

- Large scale computations

GPU generation	Fermi		Kepler		Limitation	Impact
Hardware model	GF100	GF104	GK104	GK110		
Compute Capability (CCC)	2.0	2.1	3.0	3.5		
Max. grid size (on X dimension)	2 <sup>16</sup> -1	2 <sup>16</sup> -1	2 <sup>32</sup> -1	2 <sup>32</sup> -1	Software	Problem size

- New architectural features

GPU generation	Fermi		Kepler		Limitation	Impact
Hardware model	GF100	GF104	GK104	GK110		
Compute Capability (CCC)	2.0	2.1	3.0	3.5		
Dynamic Parallelism	No	No	No	Yes	Hardware	Problem structure
Hyper-Q	No	No	No	Yes	Hardware	Thread scheduling

# Dynamic Parallelism

- The ability to launch new grids from the GPU:
  - Dynamically: Based on run-time data
  - Simultaneously: From multiple threads at once
  - Independently: Each thread can launch a different grid



*Fermi: Only CPU  
can generate GPU work.*



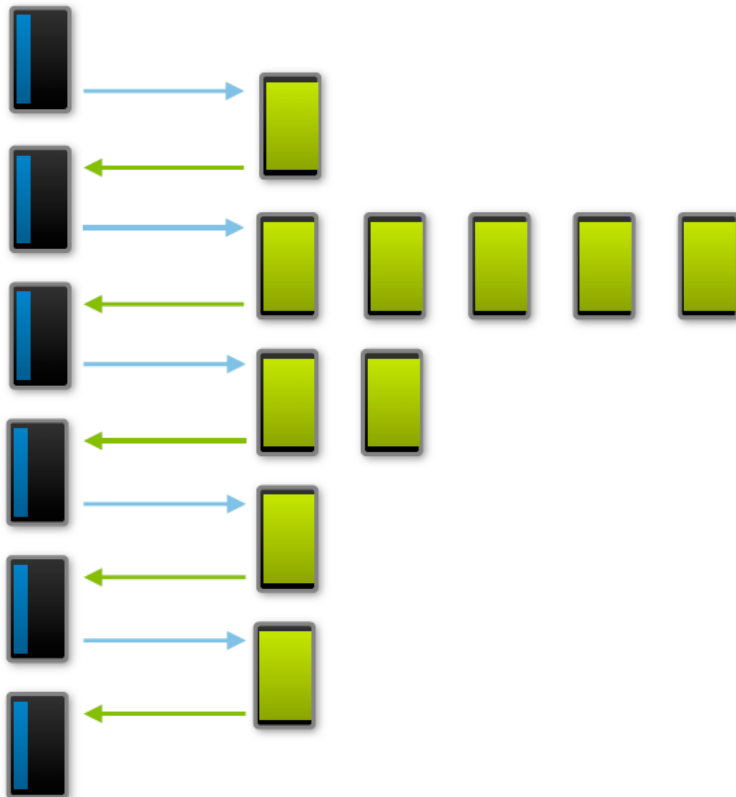
*Kepler: GPU can  
generate work for itself.*

# Dynamic Parallelism

The pre-Kepler GPU is a co-processor

CPU

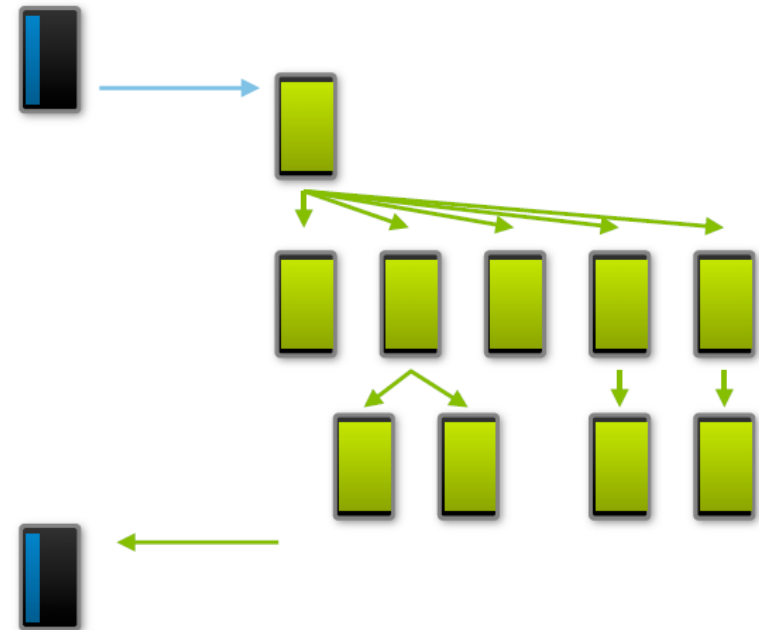
GPU



The Kepler GPU is autonomous:  
Dynamic parallelism

CPU

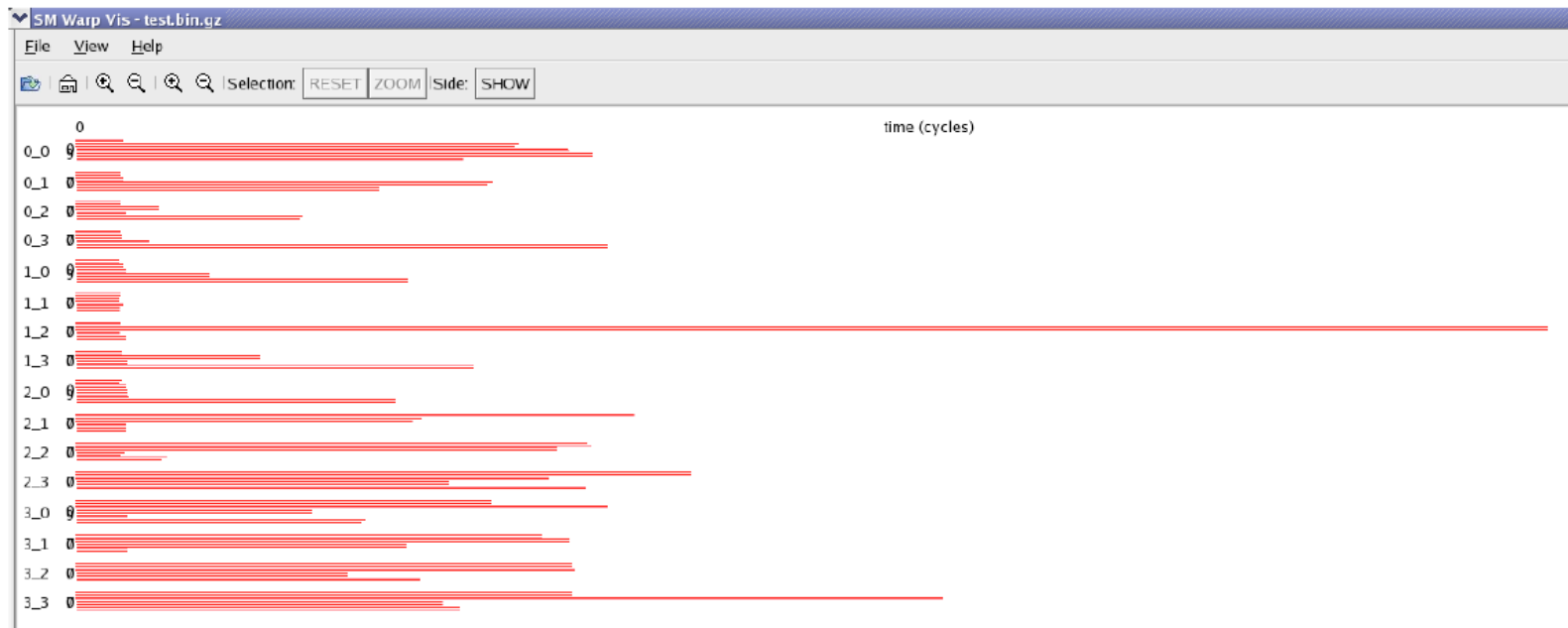
GPU



Now programs run faster and  
are expressed in a more natural way.

# Workload Balance

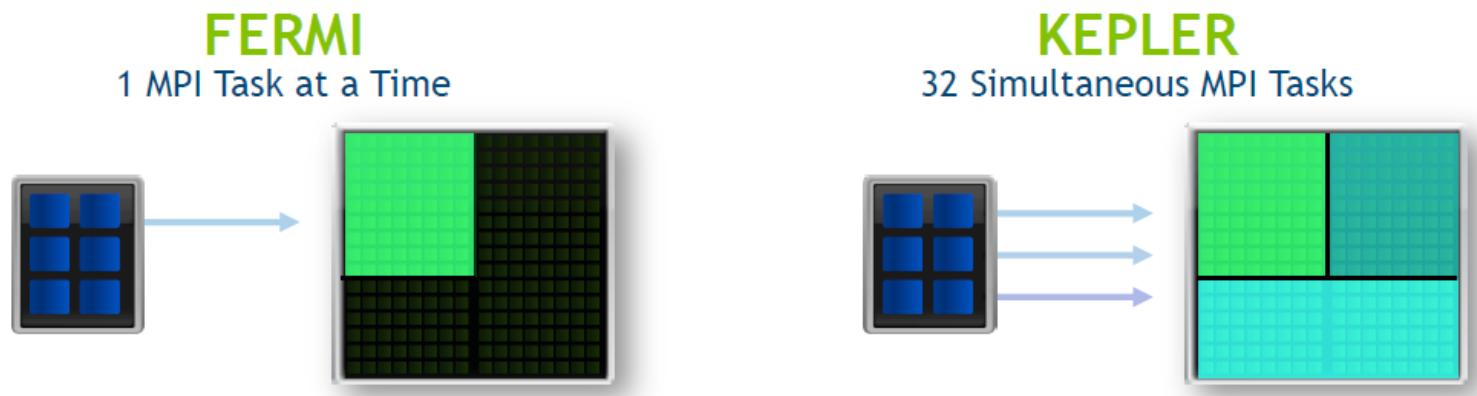
- Plenty of factors, unpredictable at run time, may transform workload balancing among multiprocessors into an impossible goal
- See below the duration of 8 warps on an SM of the G80:



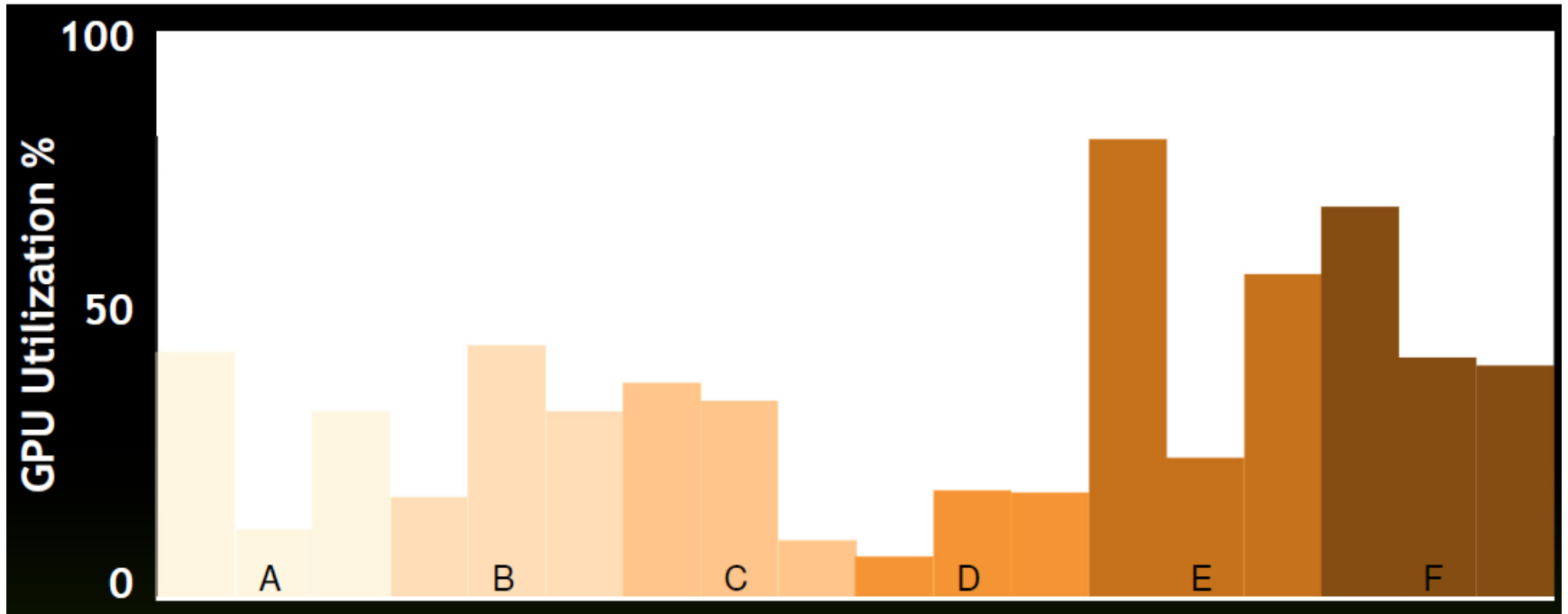


# Hyper-Q

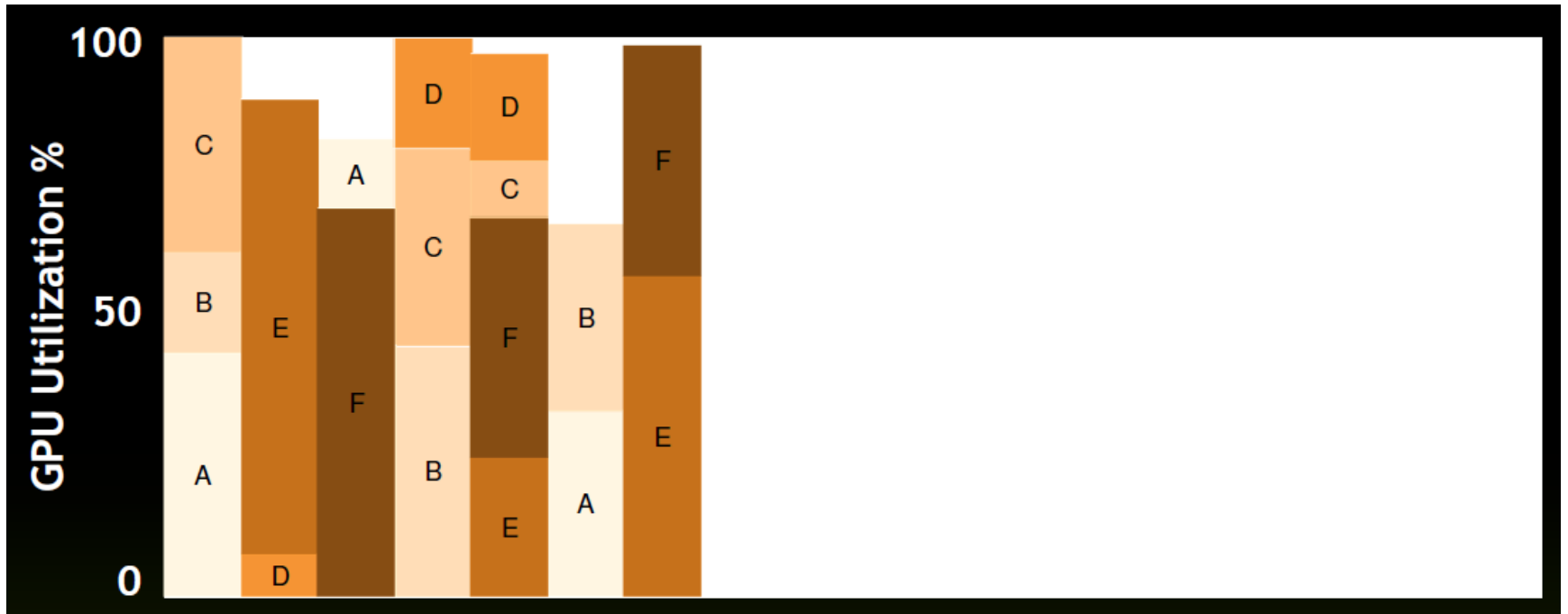
- In Fermi, several CPU processes can send thread blocks to the same GPU, but a kernel cannot start its execution until the previous one has finished
- In Kepler, we can execute simultaneously up to 32 kernels launched from different:
  - MPI processes, CPU threads (POSIX threads) or CUDA streams
- This increments the % of temporal occupancy on the GPU



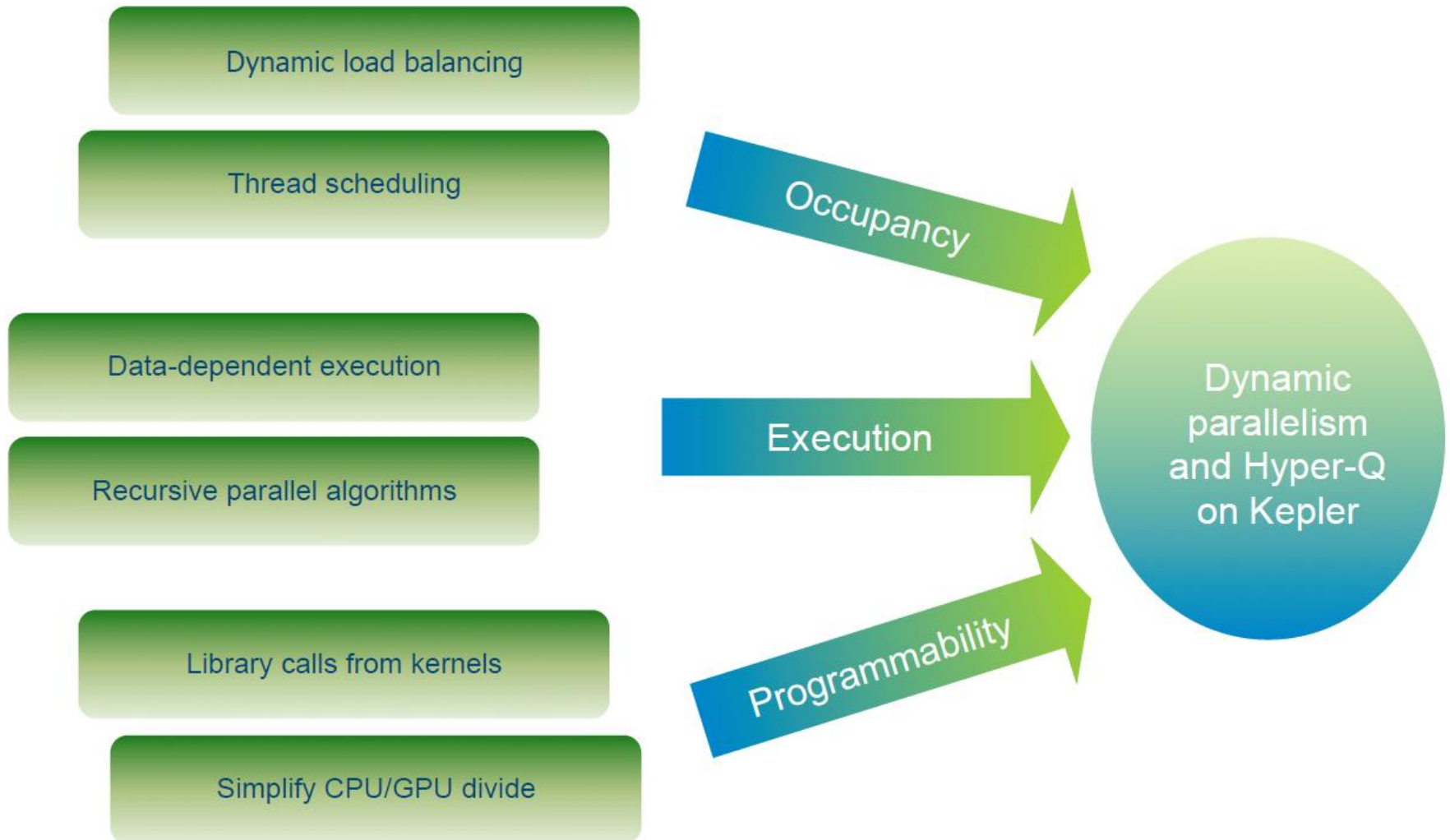
# Without Hyper-Q



# With Hyper-Q

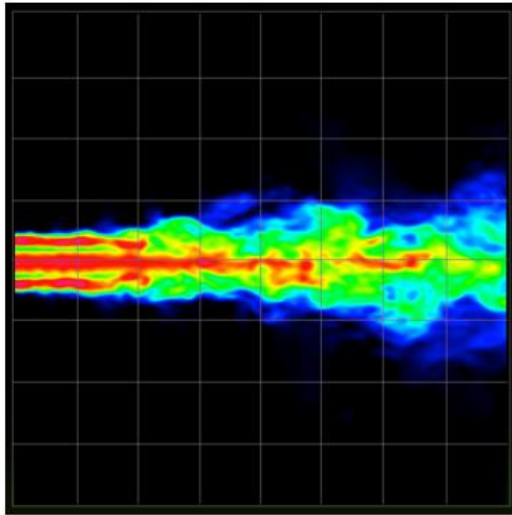


# Six Ways to Improve Code on Kepler



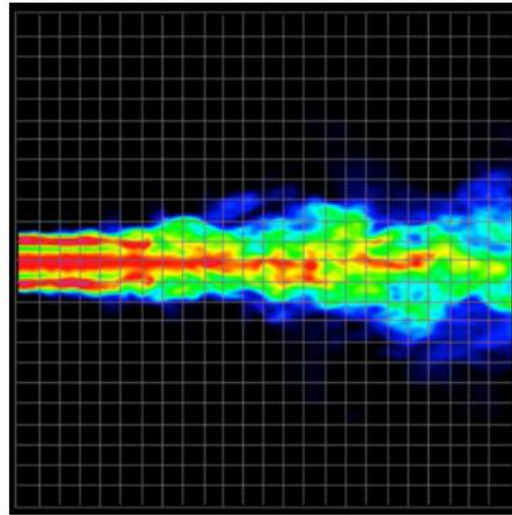
# Dynamic Work Generation

Coarse grid



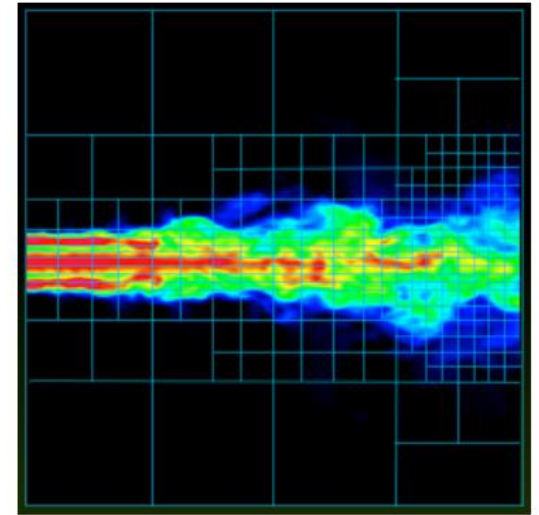
Higher performance,  
lower accuracy

Fine grid



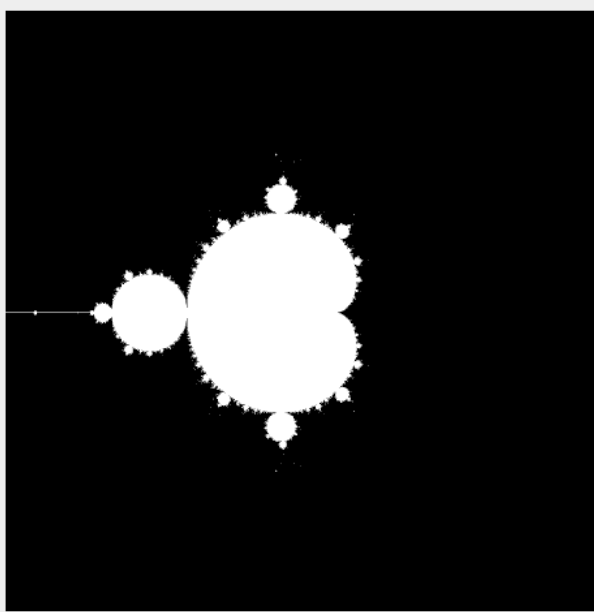
Lower performance,  
higher accuracy

Dynamic grid



Target performance  
where accuracy is required

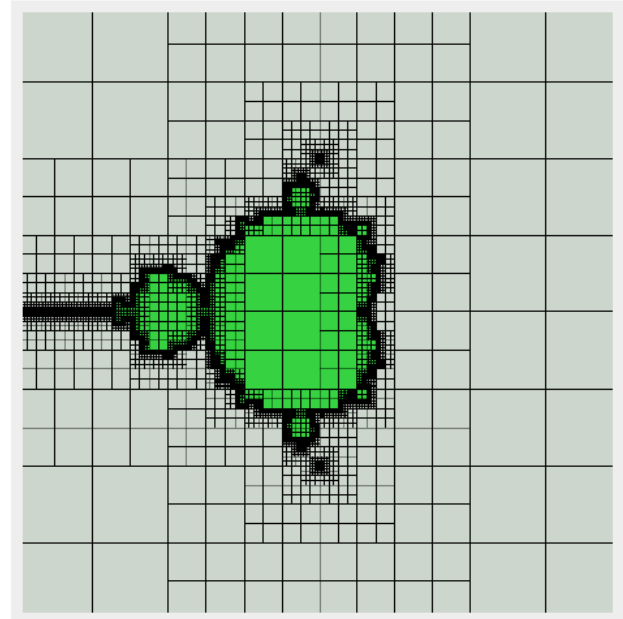
# Parallelism based on Level of Detail



**CUDA until 2012:**

- The CPU launches kernels regularly.
- All pixels are treated the same.

Computational power  
allocated to regions  
of interest



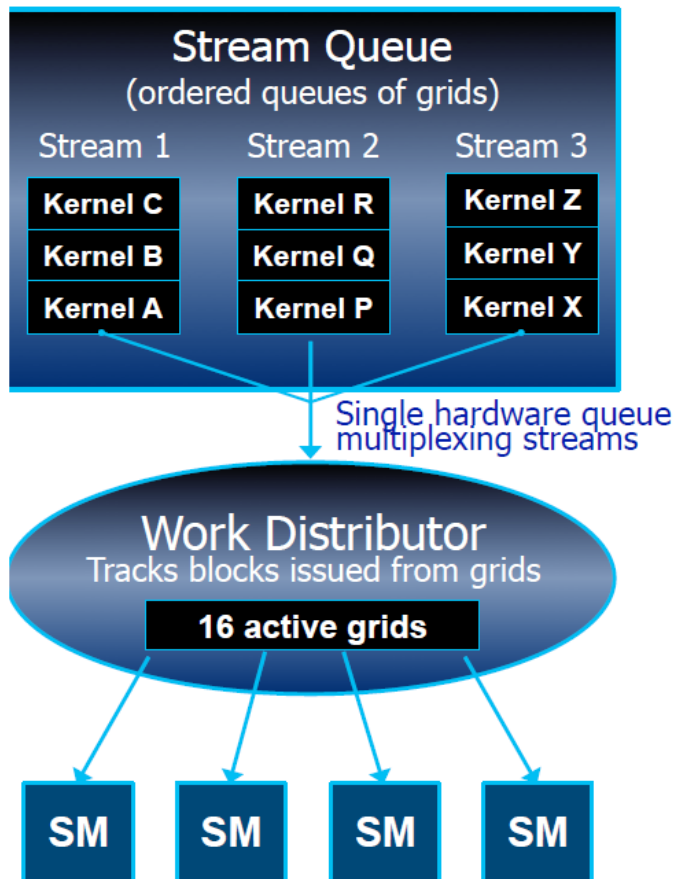
**CUDA on Kepler:**

- The GPU launches a different number of kernels/blocks for each computational region.

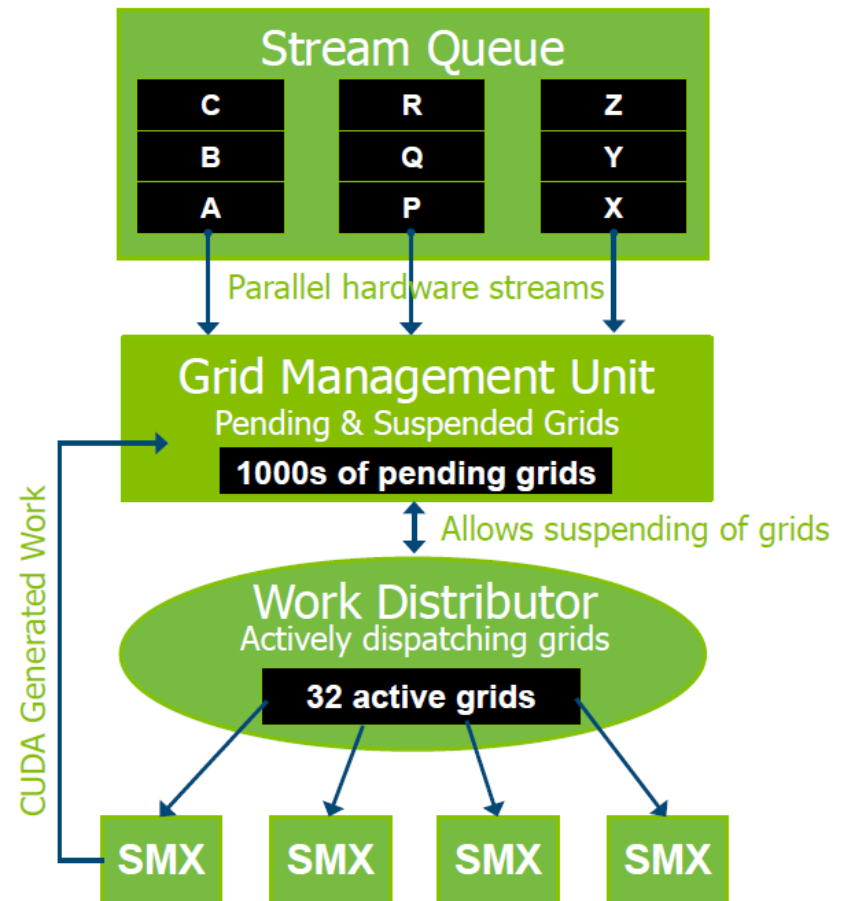


# Grid Management Unit

## Fermi



## Kepler GK110



# Software and Hardware Queues

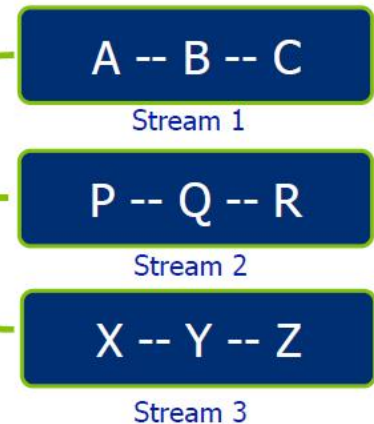
Fermi:

Up to 16 grids  
can run at once  
on GPU hardware

But CUDA streams multiplex into a single queue



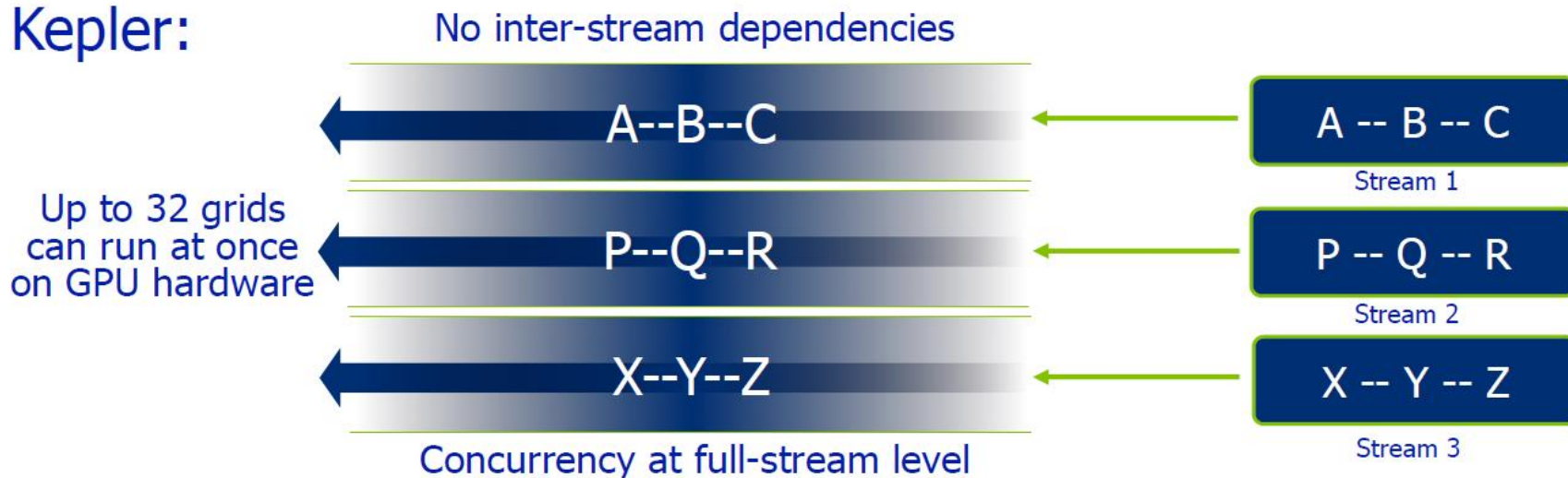
Chances for overlapping: Only at stream edges





# Software and Hardware Queues

Kepler:



# Instruction Issue and Execution

	SM-SMX fetch & issue (front-end)	SM-SMX execution (back-end)
Fermi (GF100)	<p>Can issue 2 warps, 1 instruction each. Total: <b>2 warps per cycle</b>. Active warps: 48 on each SM, chosen from up to 8 blocks. In GTX480: <math>15 * 48 = 720</math> active warps.</p>	<p>32 cores (1 warp) for "int" and "float". 16 cores for "double" (1/2 warp). 16 load/store units (1/2 warp). 4 special function units (1/8 warp). A total of up to <b>4 concurrent warps</b>.</p>
Kepler (GK110)	<p>Can issue 4 warps, 2 instructions each. Total: <b>8 warps per cycle</b>. Active warps: 64 on each SMX, chosen from up to 16 blocks. In K20: <math>13 * 64 = 832</math> active warps.</p>	<p>192 cores (6 warps) for "int" and "float". 64 cores for "double" (2 warps). 32 load/store units (1 warp). 32 special function units (1 warp). A total of up to <b>10 concurrent warps</b>.</p>

# Data-Dependent Parallelism

- The simplest possible parallel program:

- Loops are parallelizable
- Workload is known at compile-time

```
for i = 1 to N
    for j = 1 to M
        convolution(i, j);
```

- The simplest impossible program:
  - Workload is unknown at compile-time.
  - The challenge is data partitioning

```
for i = 1 to N
    for j = 1 to x[i]
        convolution(i, j);
```

# Data-Dependent Parallelism

- Kepler version:

```
__global__ void convolution(int x[])  
{  
    for j = 1 to x[blockIdx]  
        // Each block launches x[blockIdx]  
        // kernels from GPU  
        kernel <<< ... >>> (blockIdx, j)  
}  
  
// Launch N blocks of 1 thread  
// on GPU (rows start in parallel)  
convolution <<< N, 1 >>> (x);
```

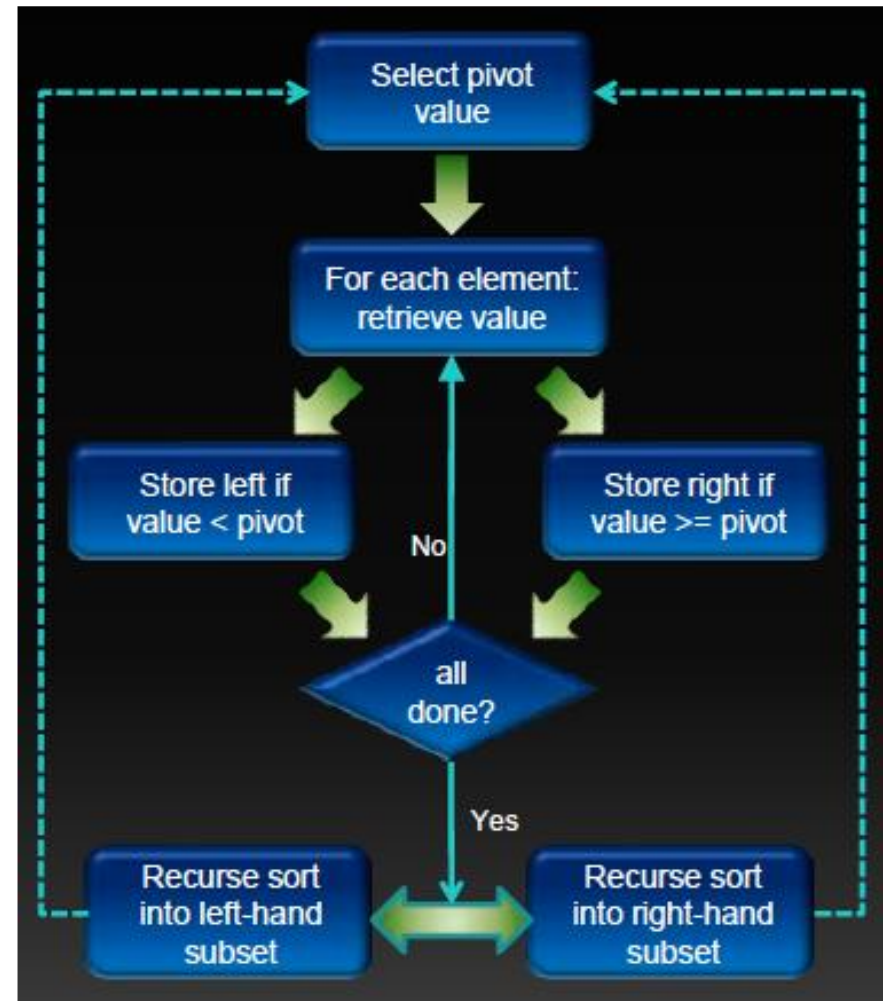
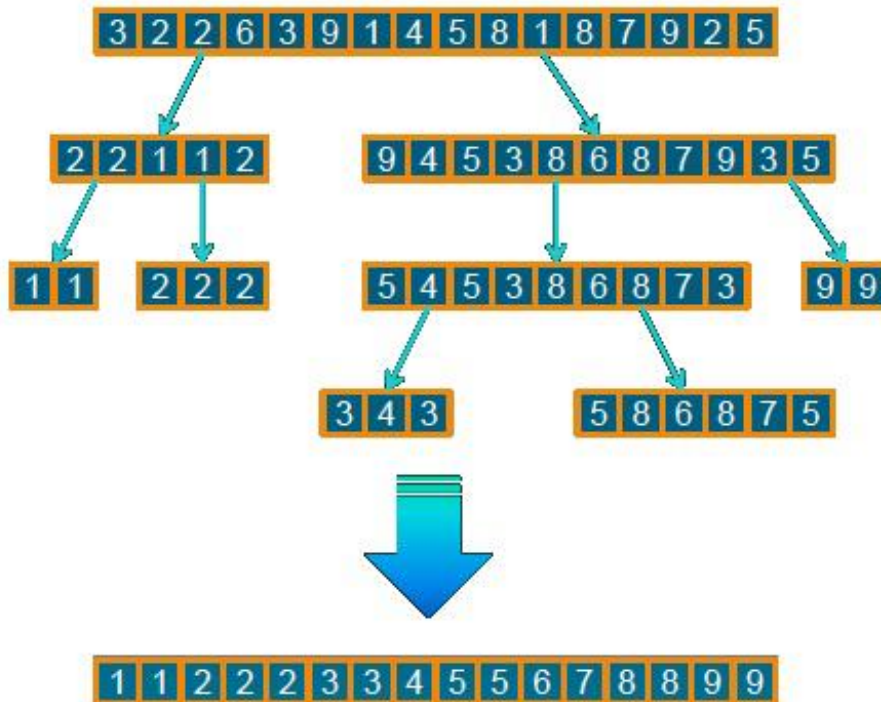
- Up to 24 nested loops supported in CUDA 5.0

# Recursive Parallel Algorithms prior to Kepler

- Early CUDA programming model did not support recursion at all
- CUDA started to support recursive functions in version 3.1, but they can easily crash if the size of the arguments is large
- A user-defined stack in global memory can be employed instead, but at the cost of a significant performance penalty
- An efficient solution is possible using dynamic parallelism

# Parallel Recursion: Quicksort

- Typical divide-and-conquer algorithm, hard to do on Fermi



# Quicksort

```
__device__ WorkStack stack;
__global__ void quicksort(int *data, int left, int right)
{
    int nleft, nright;

    // Partitions data based on pivot of first element.
    // Returns counts in nleft & nright
    partition(data+left, data+right, data[left], nleft, nright);

    // If a sub-array needs sorting, push it on the stack
    if(left < nright)
        stack.push(data, left, nright);
    if(nleft < right)
        stack.push(data, nleft, right);
}
```

# Quicksort

```
__host__ void launch_quicksort(int *data, int count)
{
    // Launch initial quicksort to populate the stack
    quicksort<<< ... >>>(data, 0, count-1);

    // Loop more quicksorts until no more work exists
    while(1)
    {
        // Wait for all sorts at this stage to finish
        cudaDeviceSynchronize();

        // Copy our stack from the device.
        WorkStack stack_copy;
        stack_copy = CopyFromDevice(stack);

        // Count of things on stack. We're done if it's zero!
        if(stack_copy.size() == 0)
            break;

        // Pop the stack and launch each new sort in its own stream
        while(stack_copy.size())
        {
            WorkStack elem = stack_copy.pop();
            cudaStream_t s;
            cudaStreamCreate(&s);
            quicksort<<< ..., s >>>(data, elem.left, elem.right);
        }
    }
}
```



# Quicksort with Dynamic Parallelism

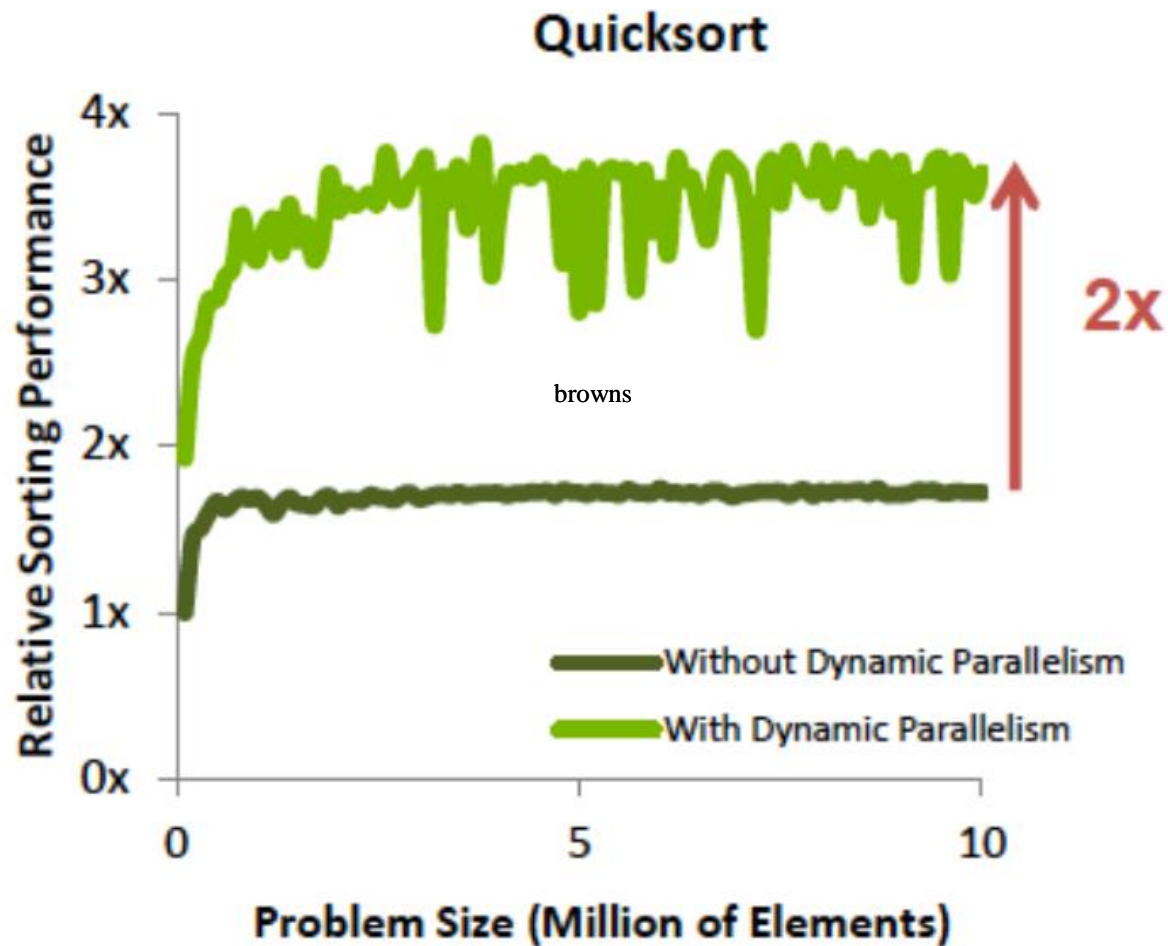
```
__global__ void quicksort(int *data, int left, int right)
{
    int nleft, nright;
    cudaStream_t s1, s2;

    // Partitions data based on pivot of first element.
    // Returns counts in nleft & nright
    partition(data+left, data+right, data[left], nleft, nright);

    // If a sub-array needs sorting, launch a new grid for it.
    // Note use of streams to get concurrency between sub-sorts
    if(left < nright) {
        cudaStreamCreateWithFlags(&s1, cudaStreamNonBlocking);
        quicksort<<< ..., s1 >>>(data, left, nright);
    }
    if(nleft < right) {
        cudaStreamCreateWithFlags(&s2, cudaStreamNonBlocking);
        quicksort<<< ..., s2 >>>(data, nleft, right);
    }
}

__host__ void launch_quicksort(int *data, int count)
{
    quicksort<<< ... >>>(data, 0, count-1);
}
```

# Quicksort Results



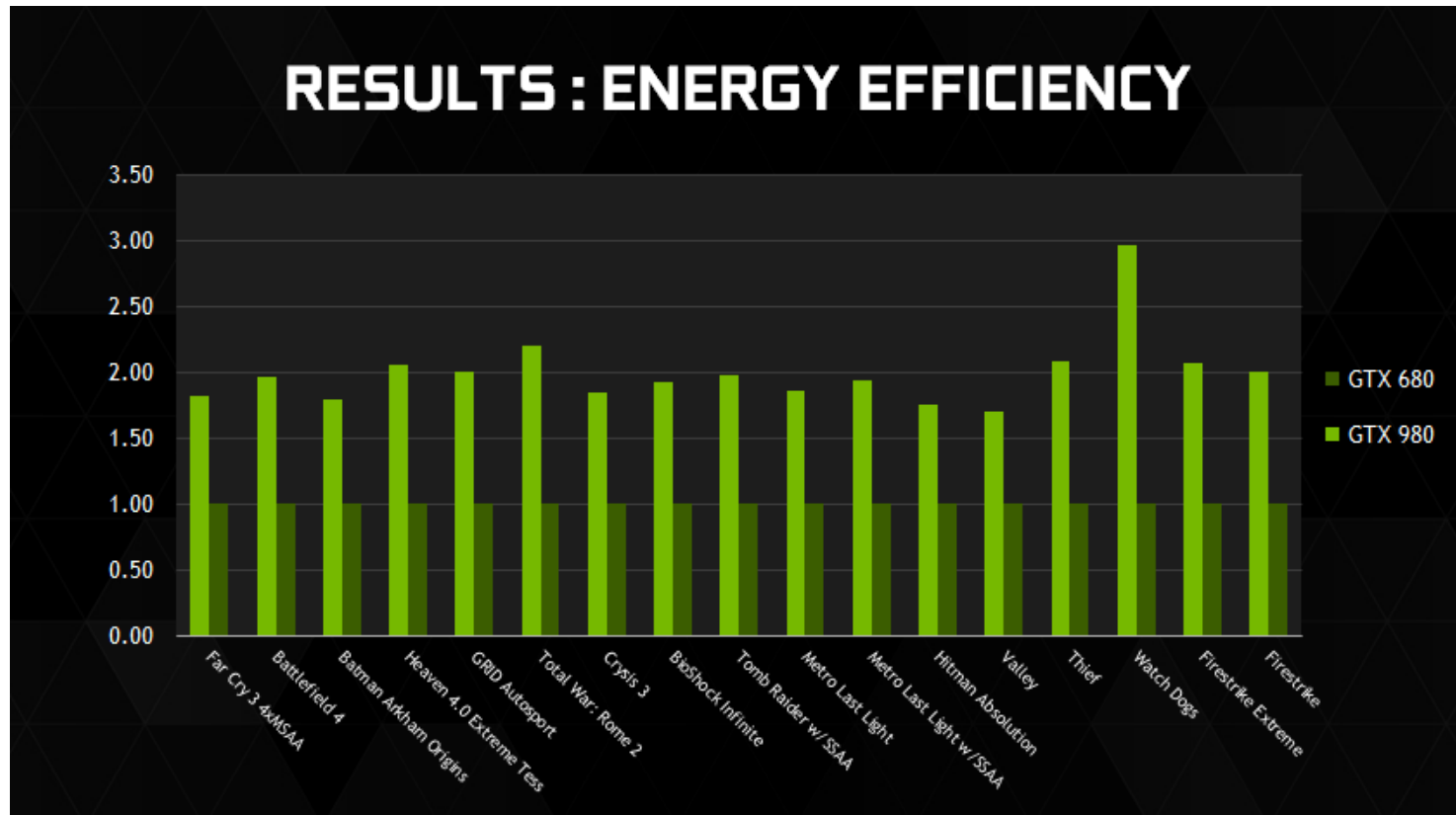
# Maxwell

## (2<sup>nd</sup> generation)

Released in 2014

Material by Mark Harris (NVIDIA)  
and others

# Energy Efficiency



Performance per Watt

GTX 680: Kepler GTX 980: Maxwell

# New Features

GPU	GeForce GTX 680 (Kepler)	GeForce GTX 980 (Maxwell)
SMs	8	16
CUDA Cores	1536	2048
Base Clock	1006 MHz	1126 MHz
GPU Boost Clock	1058 MHz	1216 MHz
GFLOPs	3090	4612 <sup>1</sup>
Texture Units	128	128
Texel fill-rate	128.8 Gigatexels/sec	144.1 Gigatexels/sec
Memory Clock	6000 MHz	7000 MHz
Memory Bandwidth	192 GB/sec	224 GB/sec
ROPs	32	64
L2 Cache Size	512KB	2048KB
TDP	195 Watts	165 Watts
Transistors	3.54 billion	5.2 billion
Die Size	294 mm <sup>2</sup>	398 mm <sup>2</sup>
Manufacturing Process	28-nm	28-nm

# New Features

- Improved instruction scheduling
  - Four warp schedulers per SMM (Maxwell SM), no shared core functional units
- Increased occupancy
  - Maximum active blocks per SMM has doubled
- Larger dedicated shared memory
  - L1 is now with texture cache
- Faster shared memory atomics
- Broader support for dynamic parallelism

# Graphics

## NEXT GENERATION GRAPHICS

Enabling New Algorithms and  
Superior Image Quality

- ▶ Voxel Global Illumination
- ▶ Multi Projection
- ▶ Conservative Raster
- ▶ Shader : Raster Ordered View
- ▶ Tiled Resources
- ▶ Advanced Sampling



# Pascal



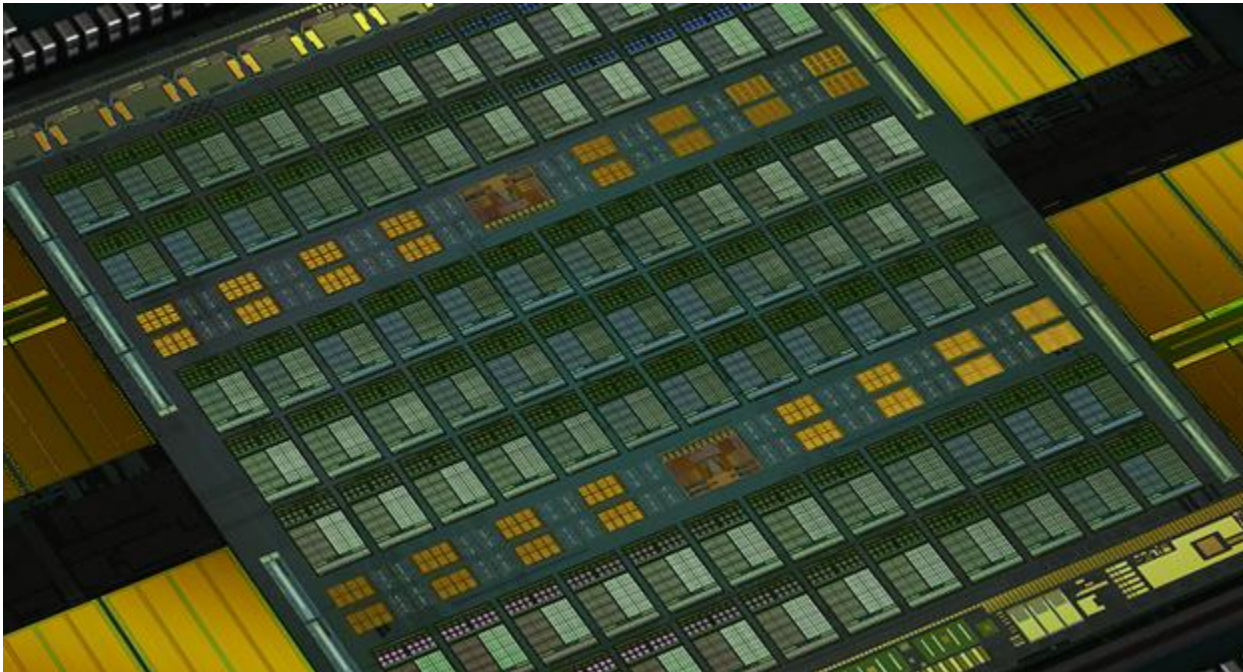
Released in 2016



# Key New Features

- Smaller manufacturing process
  - 16 nm vs. 28 nm of previous generations
- Much faster memory
- Higher clock frequency
  - 1607 MHz vs. 1216 MHz
- Dynamic load balancing including graphics pipeline
- Page Migration Engine

# Volta



Released in 2017

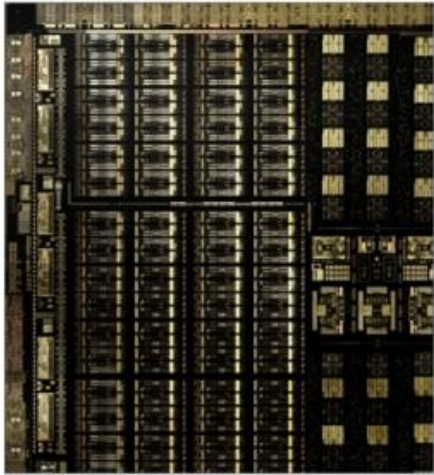
# Key New Features

- Up to 640 Tensor Cores for deep learning
  - Multiply and add floating point matrices (64 operations per clock)
  - Over 125 TFLOPS (5x more than Pascal)
- Next generation NVLink doubles bandwidth (up to 300 GB/s)
- 84 SMs
- Simultaneous execution of FP32 and INT32 operations

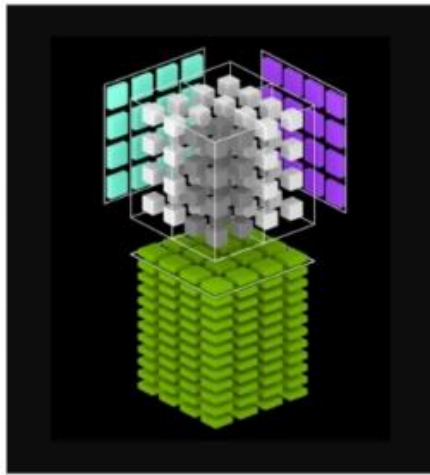
Tesla Product	Tesla K40	Tesla M40	Tesla P100	Tesla V100
GPU	GK180 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)	GV100 (Volta)
SMs	15	24	56	80
TPCs	15	24	28	40
FP32 Cores / SM	192	128	64	64
FP32 Cores / GPU	2880	3072	3584	5120
FP64 Cores / SM	64	4	32	32
FP64 Cores / GPU	960	96	1792	2560
Tensor Cores / SM	NA	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 <sup>1</sup>	5	6.8	10.6	15.7
Peak FP64 TFLOPS <sup>1</sup>	1.7	.21	5.3	7.8
Peak Tensor TFLOPS <sup>1</sup>	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 mm <sup>2</sup>	601 mm <sup>2</sup>	610 mm <sup>2</sup>	815 mm <sup>2</sup>
Manufacturing Process	28 nm	28 nm	16 nm FinFET+	12 nm FFN

<sup>1</sup> Peak TFLOPS rates are based on GPU Boost Clock

# Turing



NEW CORE ARCHITECTURE



TENSOR CORE



RT CORE



ADVANCED SHADING

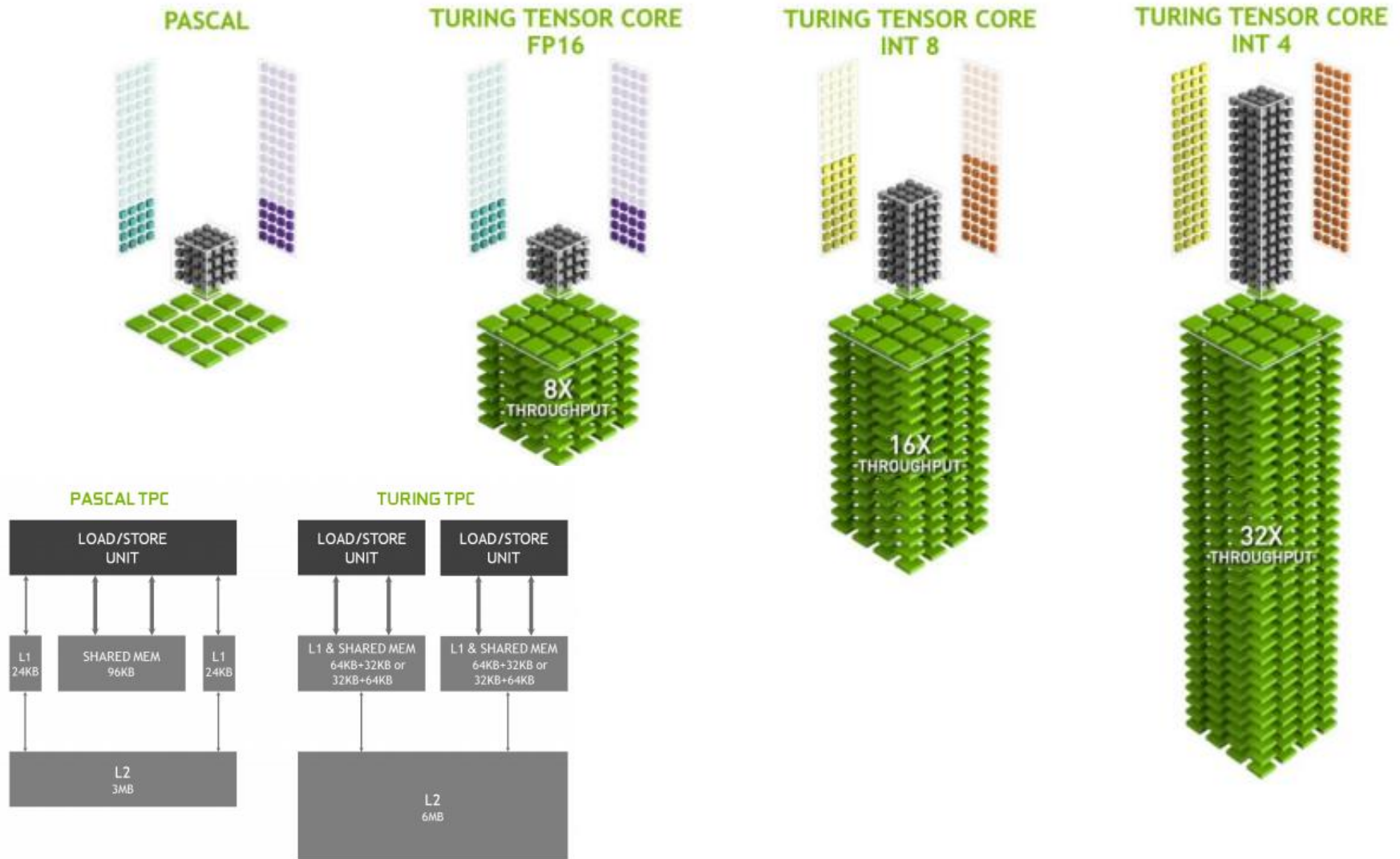
Released in 2018

# Key New Features

- CUDA, Ray-tracing and Tensor cores
  - 14.2 TFLOPS of FP32 performance, 113.8 Tensor TFLOPS and 10 Giga Rays/sec
- Up to 24 GB of RAM in Titan RTX and up to 48 GB in Quadro RTX 8000
- Independent integer and floating-point datapaths and unified shared memory, texture caching and memory load caching lead to 50% performance improvement per core



# Turing Tensor Cores



# Memory Compression

- Several lossless memory compression techniques to reduce bandwidth demands
- Improvements over Pascal



# Reflections Demo



# NVIDIA DGX-1



## NVIDIA DGX-1

WORLD'S FIRST DEEP LEARNING SUPERCOMPUTER

Engineered for deep learning | 170TF FP16 | 8x Tesla P100

NVLink hybrid cube mesh | Accelerates major AI frameworks

# "250 SERVERS IN-A-BOX"

	DUAL XEON	DGX-1
<b>FLOPS (CPU + GPU)</b>	3 TF	170 TF
<b>AGGREGATE NODE BW</b>	76 GB/ s	768 GB/ s
<b>ALEXNET TRAIN TIME</b>	150 HOURS	2 HOURS
<b>TRAIN IN 2 HOURS</b>	>250 NODES*	1 NODE

\*Caffe Training on Multi-node Distributed-memory Systems Based on Intel® Xeon® Processor E5 Family (extrapolated)  
Gennady Fedorov (Intel)'s picture Submitted by Gennady Fedorov (Intel), Vadim P. (Intel) on October 29, 2015  
<https://software.intel.com/en-us/articles/caffe-training-on-multi-node-distributed-memory-systems-based-on-intel-xeon-processor-e5>

## SYSTEM SPECIFICATIONS

GPUs	<b>8X Tesla V100</b>
Performance (Mixed Precision)	<b>1 petaFLOPS</b>
GPU Memory	<b>256 GB total system</b>
CPU	<b>Dual 20-Core Intel Xeon E5-2698 v4 2.2 GHz</b>
NVIDIA CUDA® Cores	<b>40,960</b>
NVIDIA Tensor Cores (on V100 based systems)	<b>5,120</b>
Power Requirements	<b>3,500 W</b>
System Memory	<b>512 GB 2,133 MHz DDR4 RDIMM</b>
Storage	<b>4X 1.92 TB SSD RAID 0</b>
Network	<b>Dual 10 GbE, 4 IB EDR</b>
Operating System	<b>Canonical Ubuntu, Red Hat Enterprise Linux</b>
System Weight	<b>134 lbs</b>
System Dimensions	<b>866 D x 444 W x 131 H (mm)</b>
Packing Dimensions	<b>1,180 D x 730 W x 284 H (mm)</b>
Operating Temperature Range	<b>5–35 °C</b>

# NVIDIA DGX-2

DATA CENTER

PRODUCTS ▾

SOLUTIONS ▾

APPS ▾

FOR DEVELOPERS

TECHNOLOGIES ▾

DGX-2

OVERVIEW

TOUR

CONTACT US

## NVIDIA DGX-2

The world's most powerful AI system for the most complex AI challenges.



## SYSTEM SPECIFICATIONS

GPUs	<b>16X NVIDIA® Tesla V100</b>
GPU Memory	<b>512GB total</b>
Performance	<b>2 petaFLOPS</b>
NVIDIA CUDA® Cores	<b>81920</b>
NVIDIA Tensor Cores	<b>10240</b>
NVSwitches	<b>12</b>
Maximum Power Usage	<b>10kW</b>
CPU	<b>Dual Intel Xeon Platinum 8168, 2.7 GHz, 24-cores</b>
System Memory	<b>1.5TB</b>
Network	<b>8X 100Gb/sec Infiniband/100GigE Dual 10/25/40/50/100GbE</b>
Storage	<b>OS: 2X 960GB NVME SSDs Internal Storage: 30TB (8X 3.84TB) NVME SSDs</b>
Software	<b>Ubuntu Linux OS Red Hat Enterprise Linux OS See Software stack for details</b>
System Weight	<b>360 lbs (163.29 kgs)</b>
Packaged System Weight	<b>400lbs (181.44kgs)</b>
System Dimensions	<b>Height: 17.3 in (440.0 mm) Width: 19.0 in (482.3 mm) Length: 31.3 in (795.4 mm) - No Front Bezel 32.8 in (834.0 mm) - With Front Bezel</b>
Operating Temperature Range	<b>5°C to 35°C (41°F to 95°F)</b>

# NVIDIA DGX STATION





## SYSTEM SPECIFICATIONS

GPUs	4X Tesla V100
TFLOPS (Mixed precision)	500
GPU Memory	128 GB total system
NVIDIA Tensor Cores	2,560
NVIDIA CUDA® Cores	20,480
CPU	Intel Xeon E5-2698 v4 2.2 GHz (20-Core)
System Memory	256 GB RDIMM DDR4
Storage	Data: 3X 1.92 TB SSD RAID 0 OS: 1X 1.92 TB SSD
Network	Dual 10GBASE-T (RJ45)
Display	3X DisplayPort, 4K resolution
Additional Ports	2x eSATA, 2x USB 3.1, 4x USB 3.0
Acoustics	< 35 dB
System Weight	88 lbs / 40 kg
System Dimensions	518 D x 256 W x 639 H (mm)
Maximum Power Requirements	1,500 W
Operating Temperature Range	10–30 °C
Software	Ubuntu Desktop Linux OS, Red Hat Enterprise Linux OS, DGX Recommended GPU Driver CUDA Toolkit



# AMD RX Vega

- 8 GB high bandwidth memory (HBM2)
  - 14 nm production process
- 4096 cores
- 12.7 TFLOPS
  - Compared to 11 TFLOPS of NVIDIA GTX Titan X and 15.7 TFLOPS of NVIDIA GV100 (Volta)

# AMD RADEON VII

- 16 GB high bandwidth memory (HBM2)
  - 7 nm production process
- 3840 cores
- 13.2 billion transistors
- 13.8 TFLOPS



# CUDA 4.0

# CUDA 4.0: Highlights

## *Easier Parallel Application Porting*

- Share GPUs across multiple threads
- Single thread access to all GPUs
- No-copy pinning of system memory
- New CUDA C/C++ features
- Thrust templated primitives library
- NPP image/video processing library
- Layered Textures

## *Faster Multi-GPU Programming*

- Unified Virtual Addressing
- NVIDIA GPUDirect™ v2.0
  - Peer-to-Peer Access
  - Peer-to-Peer Transfers
  - GPU-accelerated MPI

## *New & Improved Developer Tools*

- Auto Performance Analysis
- C++ Debugging
- GPU Binary Disassembler
- cuda-gdb for MacOS

# CUDA 4.0 Release

- March 2011
- Independent software release
- Unlike:
  - CUDA 1.0 released with G80/G9x in 2007 (nearly a year later than the hardware)
  - CUDA 2.0 released for GT200 in 2008
  - CUDA 3.0 released for Fermi in 2009

# CUDA 4.0 - Application Porting

- Unified Virtual Addressing
- Faster Multi-GPU Programming
  - NVIDIA GPUDirect 2.0
- Easier Parallel Programming in C++
  - Thrust

# Easier Porting of Existing Applications

Share GPUs across multiple threads

- Easier porting of multi-threaded apps
  - pthreads / OpenMP threads share a GPU
- Launch concurrent kernels from different host threads
  - Eliminates context switching overhead
- New, simple context management APIs
  - Old context migration APIs still supported

Single thread access to all GPUs

- Each host thread can now access all GPUs in the system
  - One thread per GPU limitation removed
- Easier than ever for applications to take advantage of multi-GPU
  - Single-threaded applications can now benefit from multiple GPUs
  - Easily coordinate work across multiple GPUs

# New CUDA C/C++ Language Features

- C++ new/delete
  - Dynamic memory management
- C++ virtual functions
  - Easier porting of existing applications
- Inline PTX
  - Enables assembly-level optimization



# GPU-Accelerated Image Processing

- NVIDIA Performance Primitives (NPP) library
  - 10x to 36x faster image processing
  - Initial focus on imaging and video related primitives
    - Data exchange and initialization
    - Color conversion
    - Threshold and compare operations
    - Statistics
    - Filter functions
    - Geometry transforms
    - Arithmetic and logical operations
    - JPEG



# NVIDIA GPUDirect: Towards Eliminating the CPU Bottleneck

## Version 1.0

*for applications that communicate over a network*

- Direct access to GPU memory for 3<sup>rd</sup> party devices
- Eliminates unnecessary sys mem copies & CPU overhead
- Supported by Mellanox and Qlogic
- Up to 30% improvement in communication performance

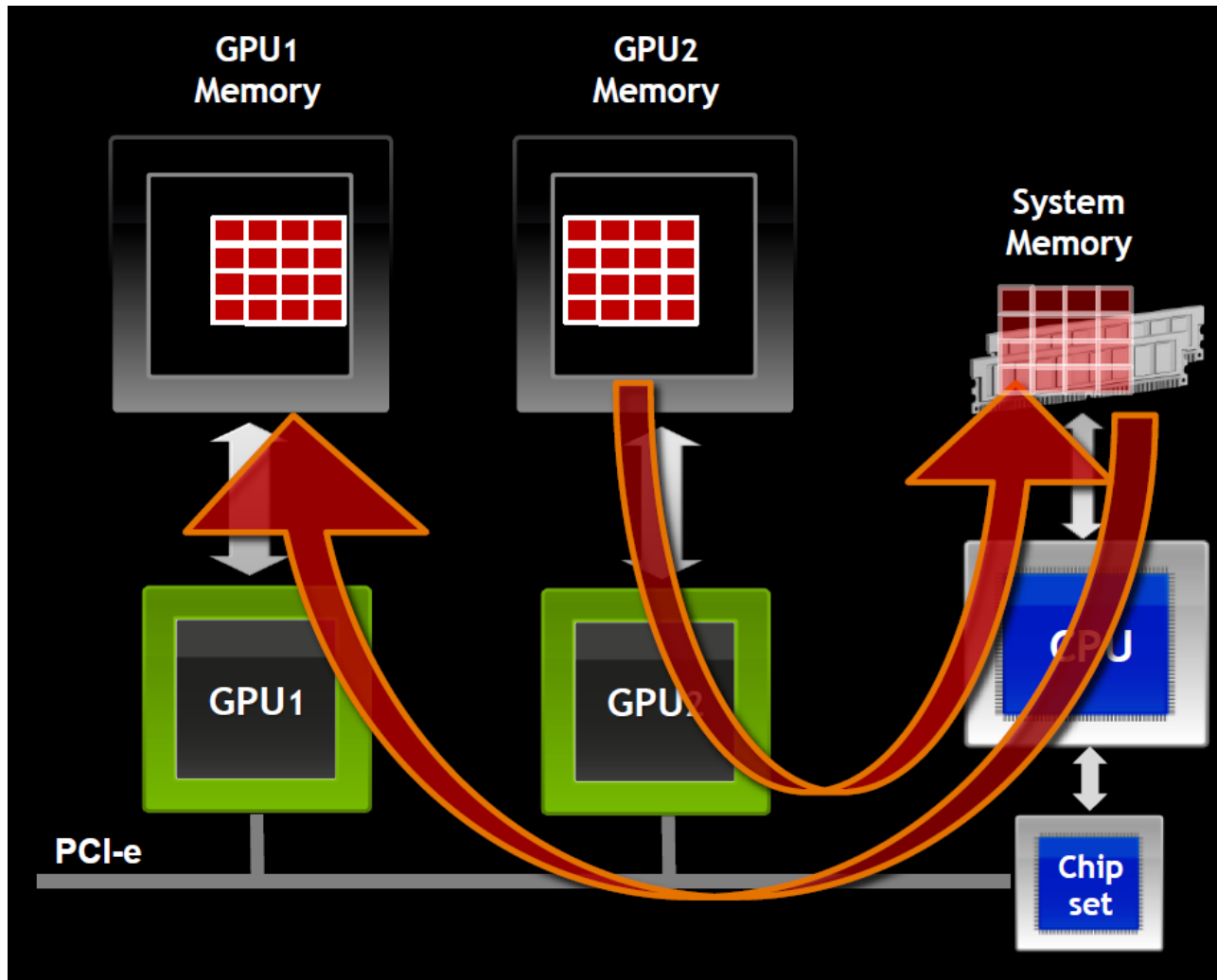
## Version 2.0

*for applications that communicate within a node*

- Peer-to-Peer memory access, transfers & synchronization
- Less code, higher programmer productivity

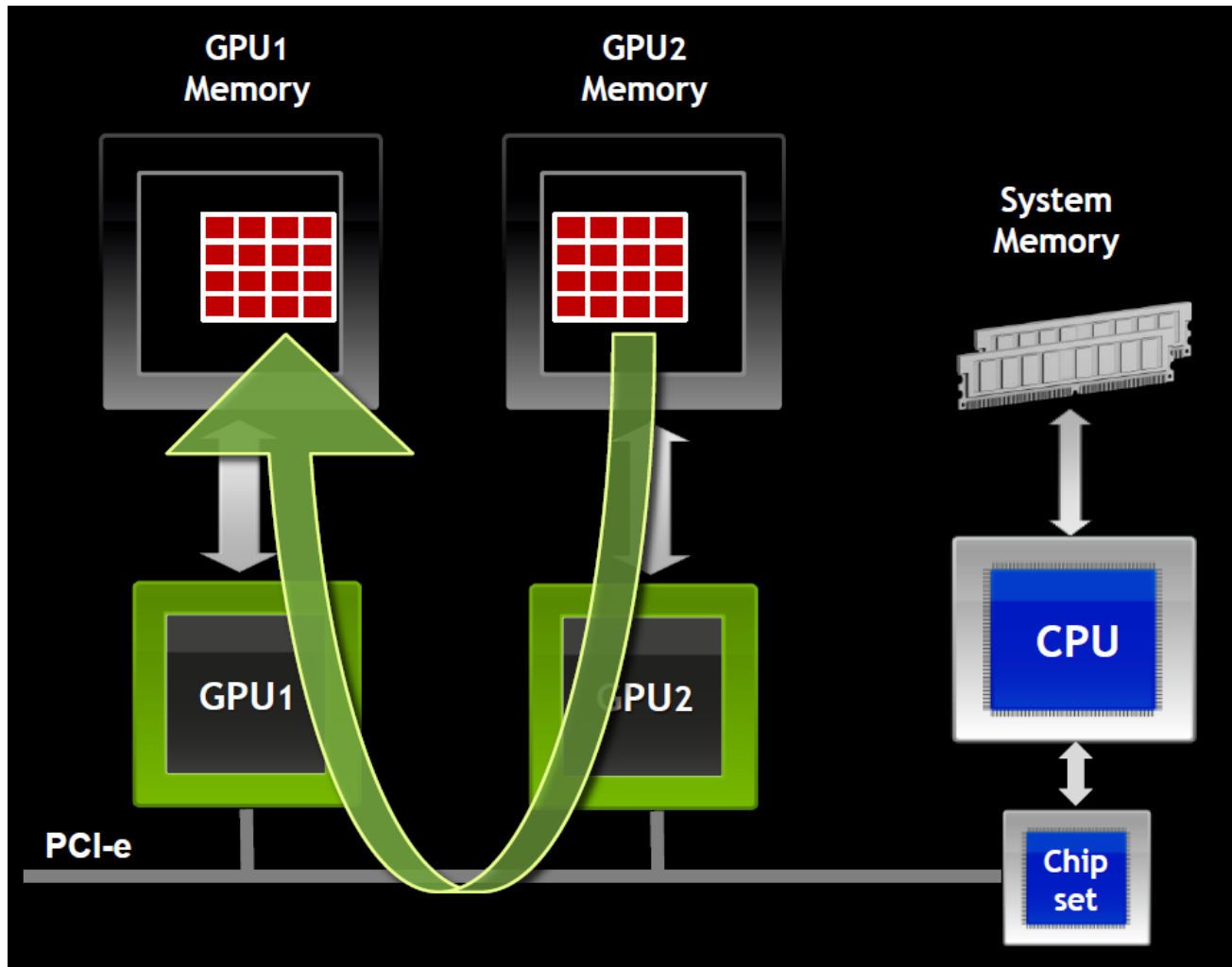
# Before GPUDirect 2.0

Two copies required



# GPUDirect 2.0: Peer-to-Peer Communication

Only one copy required

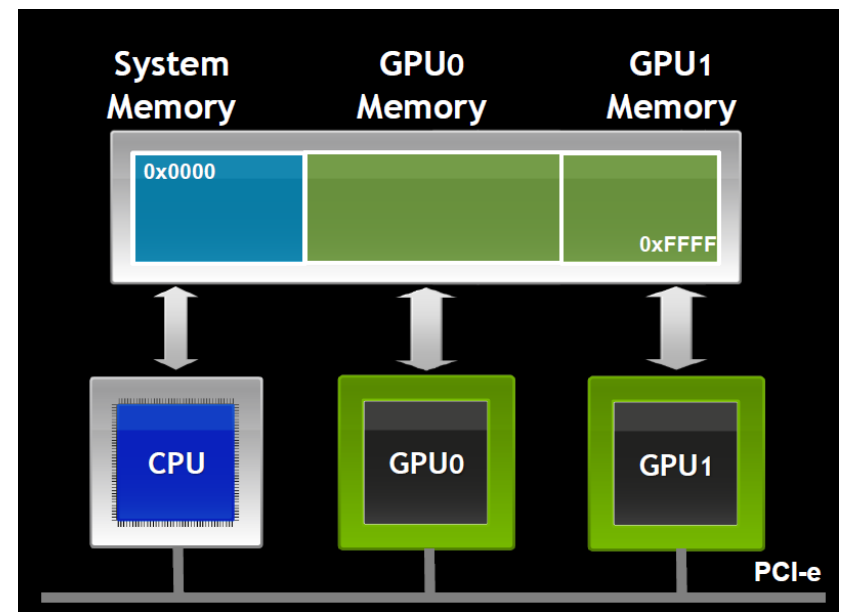
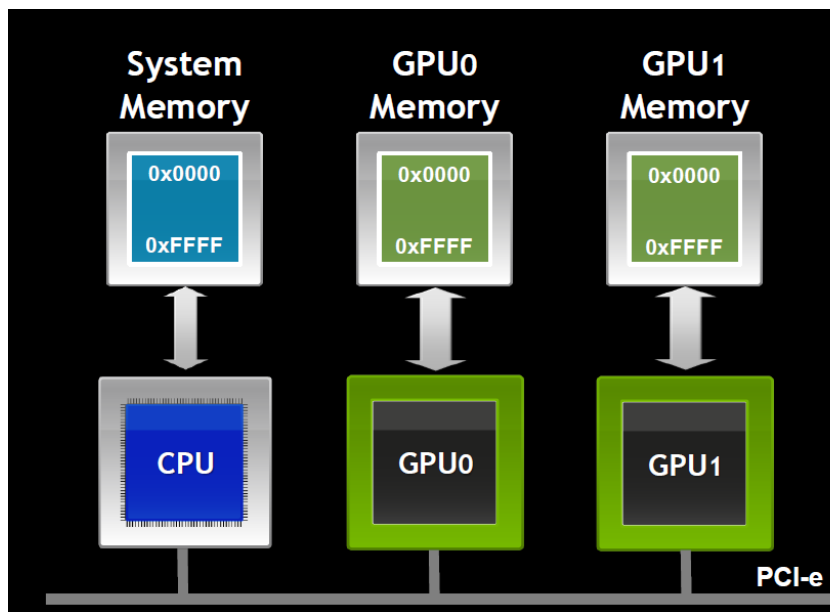


# GPUDirect 2.0: Peer-to-Peer Communication

- Direct communication between GPUs
  - Faster - no system memory copy overhead
  - More convenient multi-GPU programming
- Direct Transfers
  - Copy from GPU0 memory to GPU1 memory
  - Works transparently with UVA
- Direct Access
  - GPU0 reads or writes GPU1 memory (load/store)

# Unified Virtual Addressing

- No UVA: Multiple Memory Spaces
- UVA: Single Address Space



# Unified Virtual Addressing

- One address space for all CPU and GPU memory
  - Determine physical memory location from pointer value
  - Enables libraries to simplify their interfaces (e.g. `cudaMemcpy`)
- Supported on Tesla 20-series and other Fermi GPUs

Before UVA	With UVA
Separate options for each permutation	One function handles all cases
<code>cudaMemcpyHostToHost</code> <code>cudaMemcpyHostToDevice</code> <code>cudaMemcpyDeviceToHost</code> <code>cudaMemcpyDeviceToDevice</code>	<code>cudaMemcpyDefault</code> (data location becomes an implementation detail)

# New Developer Tools

- Auto Performance Analysis: Visual Profiler
  - Identify limiting factor
  - Analyze instruction throughput
  - Analyze memory throughput
  - Analyze kernel occupancy
- C++ Debugging
  - cuda-gdb for MacOS
- GPU Binary Disassembler

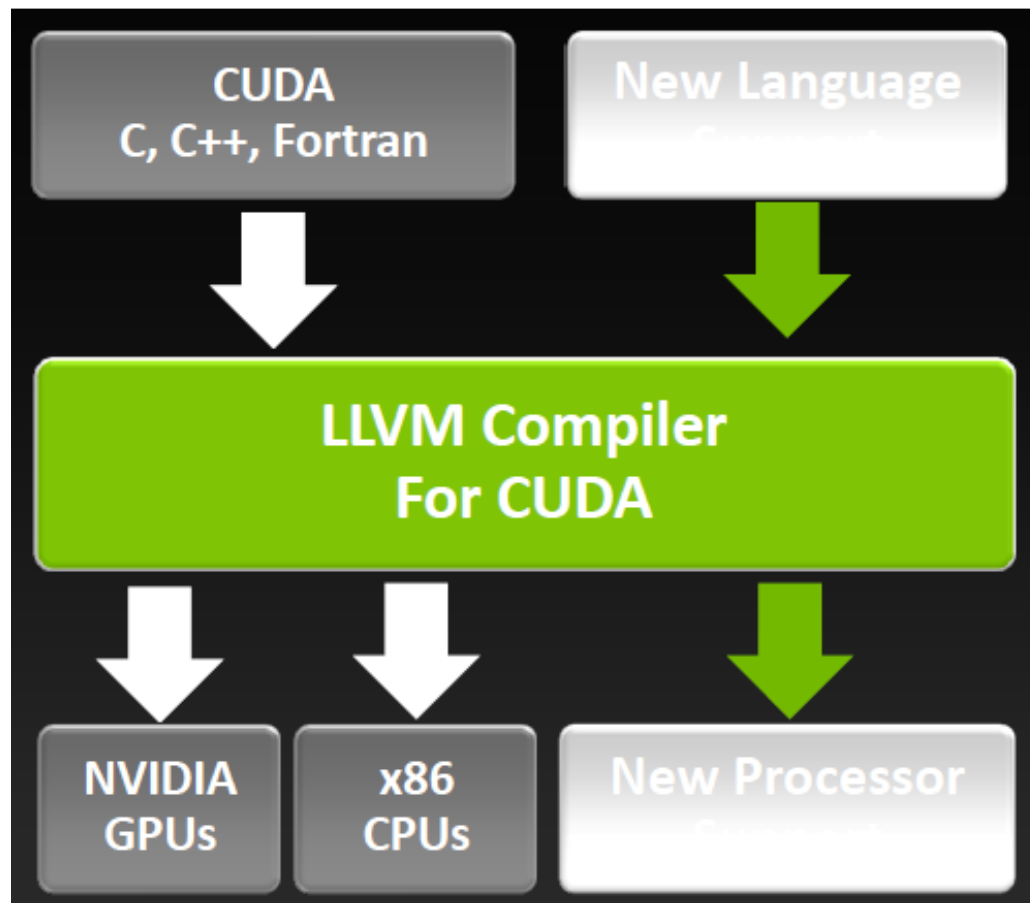


# CUDA 5.0

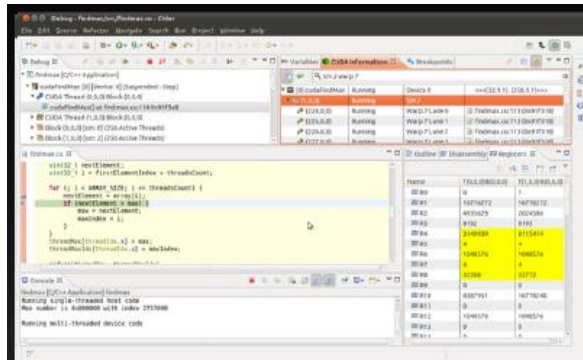
Mark Harris  
Chief Technologist, GPU  
Computing

# Open Source LLVM Compiler

- Provides ability for anyone to add CUDA to new languages and processors

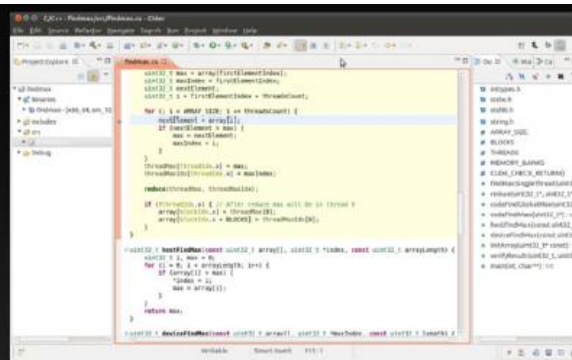


# NVIDIA Nsight, Eclipse Edition



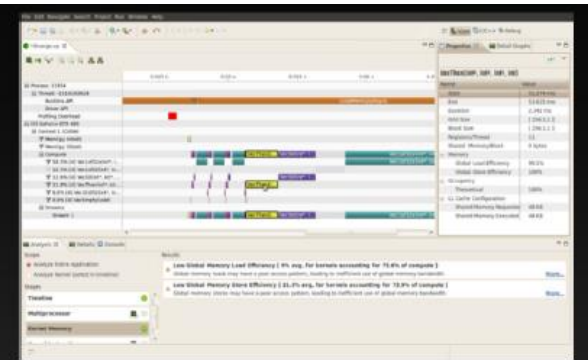
## CUDA-Aware Editor

- Automated CPU to GPU code refactoring
- Semantic highlighting of CUDA code
- Integrated code samples & docs



## Nsight Debugger

- Simultaneously debug of CPU and GPU
- Inspect variables across CUDA threads
- Use breakpoints & single-step debugging

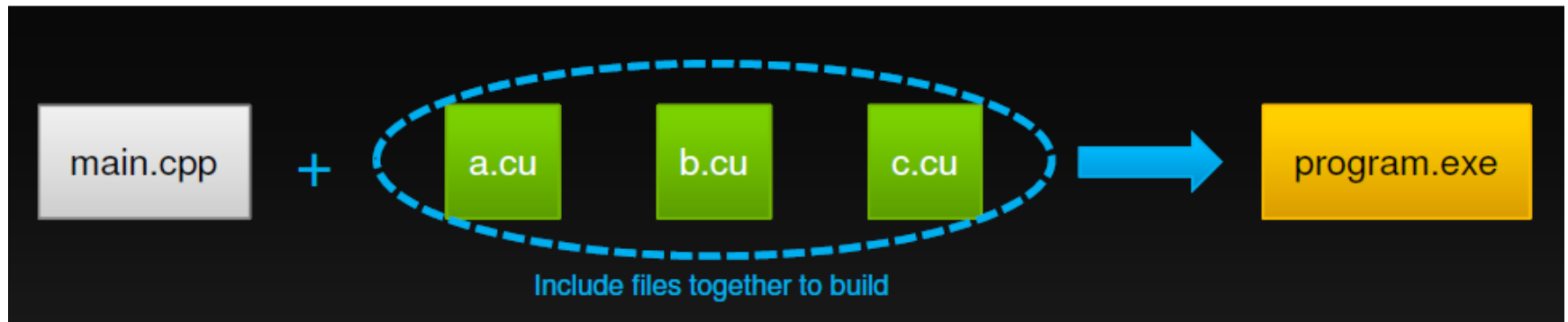


## Nsight Profiler

- Quickly identifies performance issues
- Integrated expert system
- Automated analysis
- Source line correlation

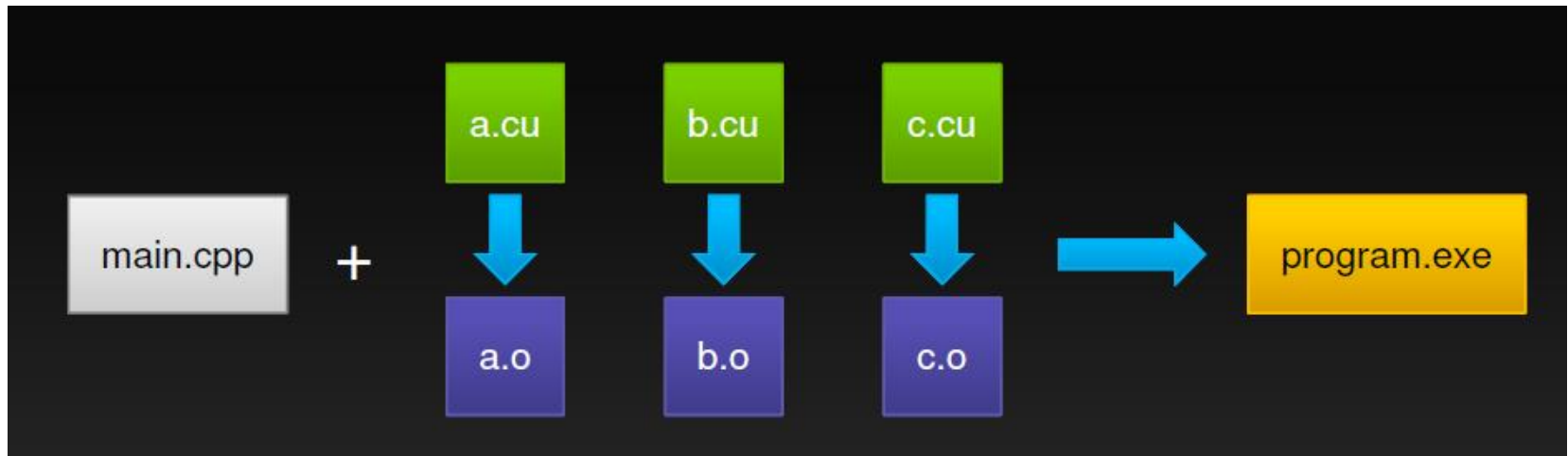
For Linux and Mac OS

# CUDA 4: Whole-Program Compilation & Linking



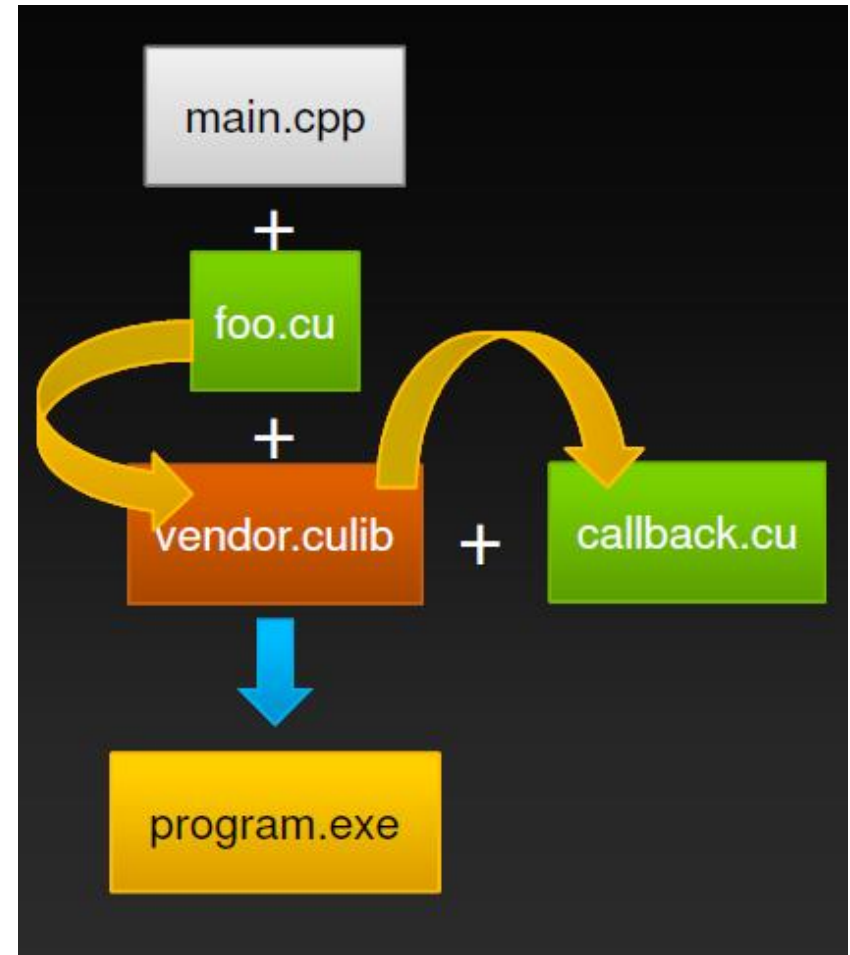
# CUDA 5: GPU Library Object Linking

- Separate compilation allows building independent object files
- CUDA 5 can link multiple object files into one program
- Can also combine object files into static libraries
  - Link and externally call *device* code



# CUDA 5: GPU Library Object Linking

- Enables 3rd party closed-source device libraries
- User-defined device callback functions



# CUDA 5.0: Run-time Syntax and Semantics

```
__device__ float buf[1024];
__global__ void dynamic(float *data)
{
    int tid = threadIdx.x;
    if (tid % 2)
        buf[tid/2] = data[tid]+data[tid+1];
    __syncthreads();

    if (tid == 0) {
        launchkernel<<<128,256>>>(buf);
        cudaDeviceSynchronize();
    }
    __syncthreads();

    if (tid == 0) {
        cudaMemCpyAsync(data, buf, 1024);
        cudaDeviceSynchronize();
    }
}
```

← This launch is per-thread

← CUDA 5.0: Sync. all launches within my block

← idle threads wait for the others here

← CUDA 5.0: Only async. launches  
are allowed on data gathering

# CUDA 6.0

Manuel Ujaldon  
Nvidia CUDA Fellow  
Computer Architecture  
Department  
University of Malaga (Spain)



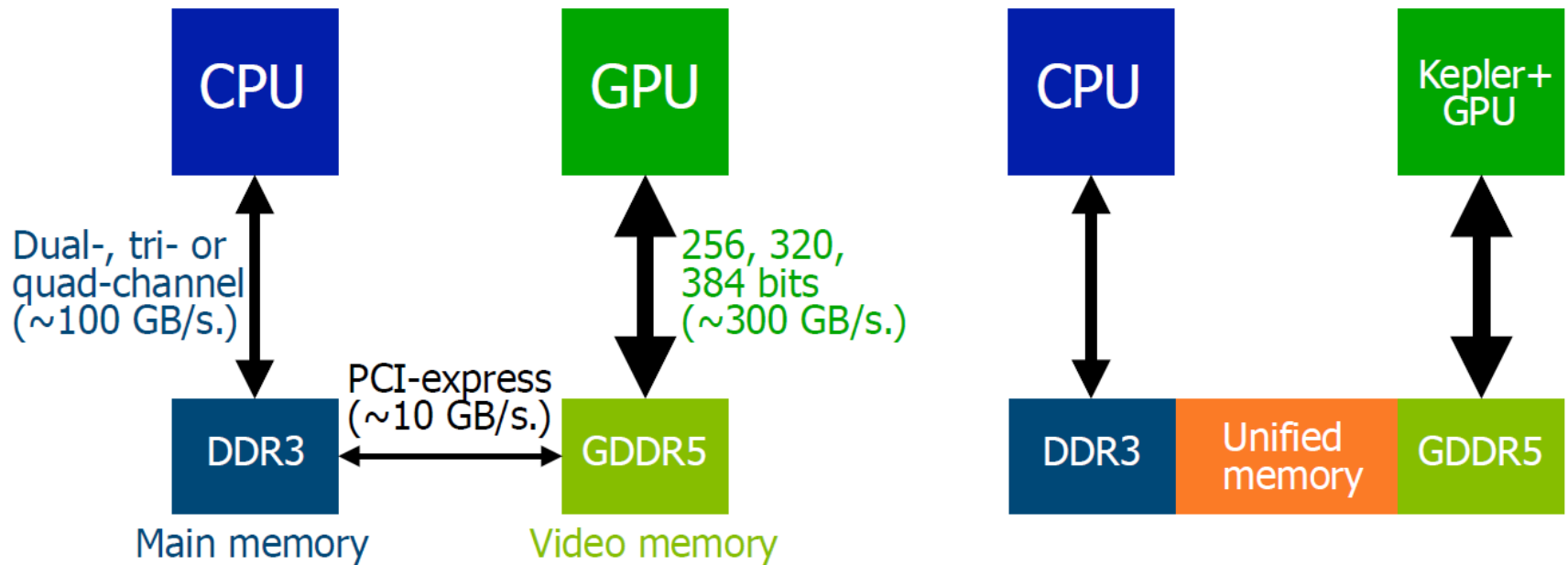
# CUDA 6 Highlights

- Unified Memory:
  - CPU and GPU can share data without much programming effort
- Extended Library Interface (XT) and Drop-in Libraries:
  - Libraries much easier to use
- GPUDirect RDMA:
  - A key achievement in multi-GPU environments
- Developer tools:
  - Visual Profiler enhanced with:
    - Side-by-side source and disassembly view showing.
    - New analysis passes (per SM activity level), generates a kernel analysis report.
- Multi-Process Server (MPS) support in nvprof and cuda-memcheck
- Nsight Eclipse Edition supports remote development (x86 and ARM)

# CUDA 6.0: Performance Improvements in Key Use Cases

- Kernel launch
- Repeated launch of the same set of kernels
- `cudaDeviceSynchronize()`
- Back-to-back grids in a stream

# Unified Memory



# Unified Memory Contributions

- Creates pool of managed memory between CPU and GPU
- Simpler programming and memory model:
  - Single pointer to data, accessible anywhere
  - Eliminate need for `cudaMemcpy()`, use `cudaMallocManaged()`
  - No need for deep copies
- Performance through data locality:
  - Migrate data to accessing processor
  - Guarantee global coherency
  - Still allows `cudaMemcpyAsync()` hand tuning

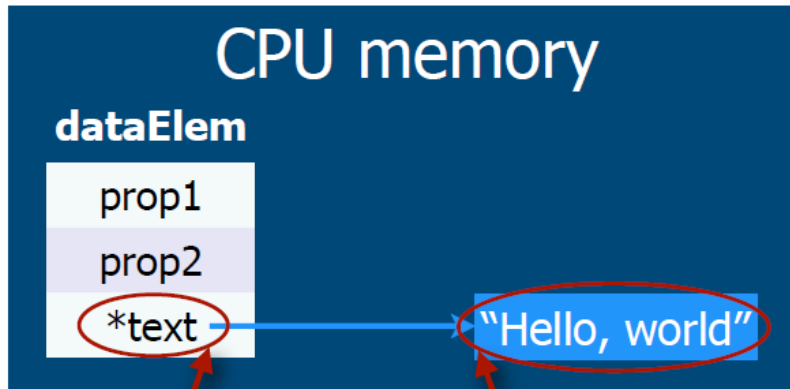
# Memory Types

	<b>Zero-Copy (pinned memory)</b>	<b>Unified Virtual Addressing</b>	<b>Unified Memory</b>
CUDA call	<code>cudaMallocHost(&amp;A, 4);</code>	<code>cudaMalloc(&amp;A, 4);</code>	<code>cudaMallocManaged(&amp;A, 4);</code>
Allocation fixed in	Main memory (DDR3)	Video memory (GDDR5)	Both
Local access for	CPU	Home GPU	CPU and home GPU
PIC-e access for	All GPUs	Other GPUs	Other GPUs
Other features	Avoid swapping to disk	No CPU access	On access CPU/GPU migration
Coherency	At all times	Between GPUs	Only at launch & sync.
Full support in	CUDA 2.2	CUDA 1.0	CUDA 6.0

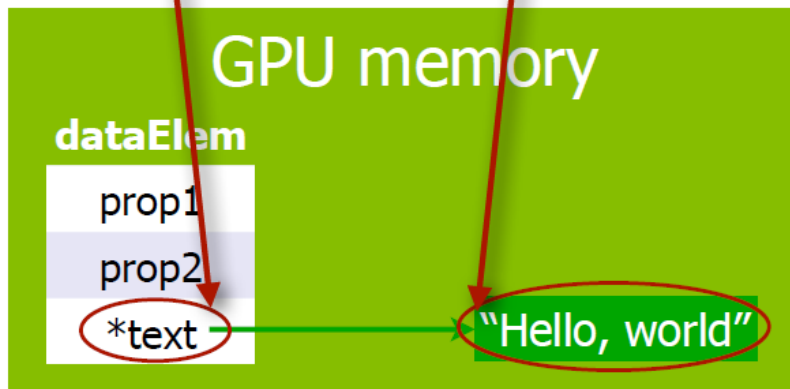
# Additions to the CUDA API

- New call: **cudaMallocManaged()**
  - Drop-in replacement for `cudaMalloc()` allocates managed memory
  - Returns pointer accessible from both Host and Device
- New call: **cudaStreamAttachMemAsync()**
  - Manages concurrency in multi-threaded CPU applications
- New keyword: **\_\_managed\_\_**
  - Declares global-scope migratable device variable
  - Symbol accessible from both GPU and CPU code

# Code without Unified Memory



Two addresses  
and two copies  
of the data



```
void launch(dataElem *elem) {  
    dataElem *g_elem;  
    char *g_text;  
  
    int textlen = strlen(elem->text);  
  
    // Allocate storage for struct and text  
    cudaMalloc(&g_elem, sizeof(dataElem));  
    cudaMalloc(&g_text, textlen);  
  
    // Copy up each piece separately, including  
    // new "text" pointer value  
    cudaMemcpy(g_elem, elem, sizeof(dataElem));  
    cudaMemcpy(g_text, elem->text, textlen);  
    cudaMemcpy(&(g_elem->text), &g_text,  
               sizeof(g_text));  
  
    // Finally we can launch our kernel, but  
    // CPU and GPU use different copies of "elem"  
    kernel<<< ... >>>(g_elem);  
}
```

# Code with Unified Memory

CPU memory

Unified memory

**dataElem**

prop1

prop2

\*text

→ "Hello, world"

GPU memory

```
void launch(dataElem *elem) {  
    kernel<<< ... >>>(elem);  
}
```

- What remains the same:
  - Data movement
  - GPU accesses a local copy of text
- What has changed:
  - Programmer sees a single pointer
  - CPU and GPU both reference the same object
  - There is coherence



# CUDA 7.0

By Mark Harris  
NVIDIA

# New Features: C++11

- C++11 features on device including:
  - auto,
  - lambda,
  - variadic templates,
  - rvalue references,
  - range-based for loops

# Example

```
#include <initializer_list>
#include <iostream>
#include <cstring>

// Generic parallel find routine. Threads search through the
// array in parallel. A thread returns the index of the
// first value it finds that satisfies predicate `p`, or -1.
template <typename T, typename Predicate>
__device__ int find(T *data, int n, Predicate p)
{
    for (int i = blockIdx.x * blockDim.x + threadIdx.x;
         i < n;
         i += blockDim.x * gridDim.x)
    {
        if (p(data[i])) return i;
    }
    return -1;
}
```

```

// Use find with a lambda function that searches for x, y, z
// or w. Note the use of range-based for loop and
// initializer_list inside the functor, and auto means we
// don't have to know the type of the lambda or the array
__global__
void xyzw_frequency(unsigned int *count, char *data, int n)
{
    auto match_xyzw = [](char c) {
        const char letters[] = { 'x', 'y', 'z', 'w' };
        for (const auto x : letters)
            if (c == x) return true;
        return false;
    };

    int i = find(data, n, match_xyzw);

    if (i >= 0) atomicAdd(count, 1);
}

```

```

int main(void)
{
    char text[] = "zebra xylophone wax";
    char *d_text;

    cudaMalloc(&d_text, sizeof(text));
    cudaMemcpy(d_text, text, sizeof(text), cudaMemcpyHostToDevice);

    unsigned int *d_count;
    cudaMalloc(&d_count, sizeof(unsigned int));
    cudaMemset(d_count, 0, sizeof(unsigned int));

    xyzw_frequency<<<1, 64>>>(d_count, d_text, strlen(text));

    unsigned int count;
    cudaMemcpy(&count, d_count, sizeof(unsigned int), cudaMemcpyDeviceToHost);

    std::cout << count << " instances of 'x', 'y', 'z', 'w'"
              << "in " << text << std::endl;

    cudaFree(d_count);
    cudaFree(d_text);

    return 0;
}

```

# Other Features

- Thrust version 1.8
  - Thrust algorithms can now be invoked from the device
- cuSOLVER, cuFFT
  - cuSolver library is a high-level package based on the cuBLAS and cuSPARSE libraries
- Runtime compilation
  - No need to generate multiple optimized kernels at compile time

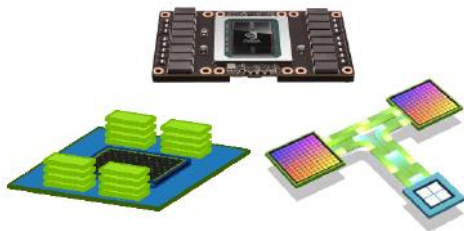
# CUDA 8.0

By Milind Kukanur  
NVIDIA

# What's New

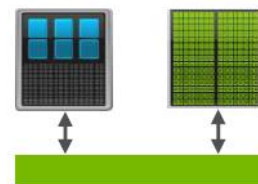
## PASCAL SUPPORT

New Architecture  
NVLINK  
HBM2 Stacked Memory  
Page Migration Engine



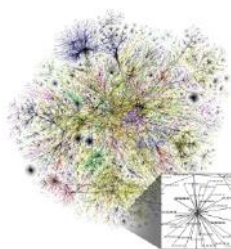
## UNIFIED MEMORY

Larger Datasets  
Demand Paging  
New Tuning APIs  
Data Coherence & Atomics



## LIBRARIES

New nvGRAPH library  
cuBLAS improvements  
for Deep Learning



## DEVELOPER TOOLS

Critical Path Analysis  
2x Faster Compile Time  
OpenACC Profiling  
Debug CUDA Apps on Display GPU





# Unified Memory

- Oversubscribe GPU memory, up to system memory size

```
void foo() {  
    // Allocate 64 GB  
    char *data;  
    size_t size = 64*1024*1024*1024;  
    cudaMallocManaged(&data, size);  
}
```

# Unified Memory

```
__global__ void mykernel(char *data) {  
    data[1] = 'g';  
}
```

```
void foo() {  
    char *data;  
    cudaMallocManaged(&data, 2);  
  
    mykernel<<<...>>>(data);  
    // no synchronize here  
    data[0] = 'c';  
  
    cudaFree(data);  
}
```

# CUDA 9.0

By Mark Harris  
NVIDIA

# New Features

- Support for Volta
- Cooperative groups
- Tensor Core API
- New Visual Profiler
- Support for C++ 14

# Cooperative Groups

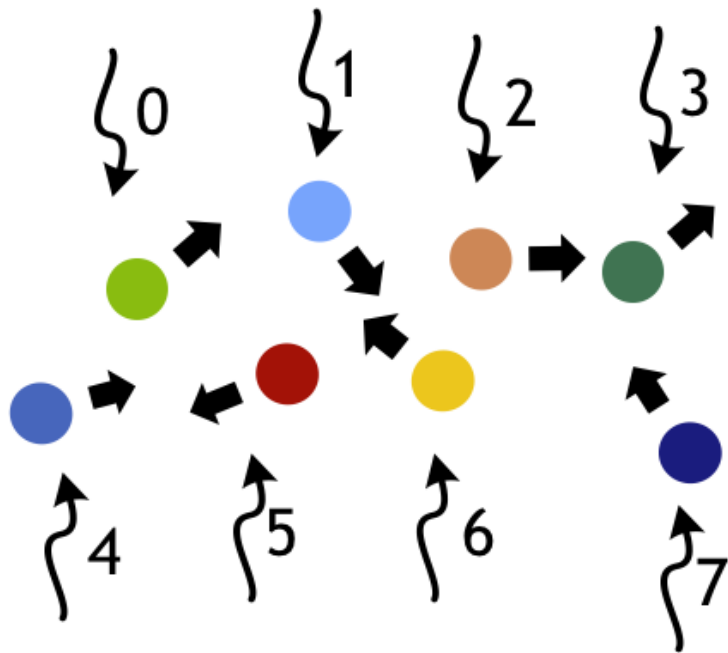
Ability to define groups of threads explicitly at sub-block and multiblock granularities

```
__global__ void cooperative_kernel(...)
{
    // obtain default "current thread block" group
    thread_group my_block = this_thread_block();

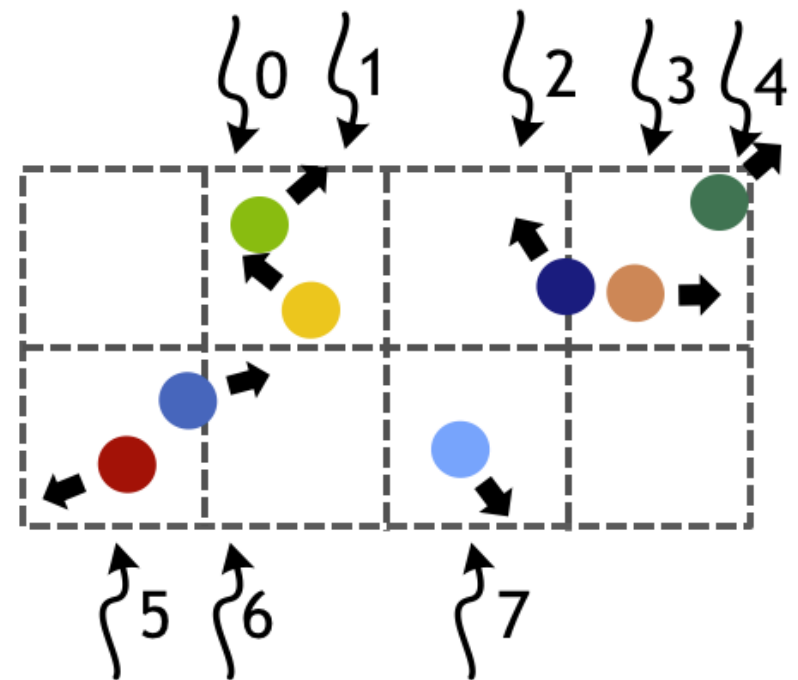
    // subdivide into 32-thread, tiled subgroups
    // Tiled subgroups evenly partition a parent group into
    // adjacent sets of threads - in this case each one warp in size
    thread_group my_tile = tiled_partition(my_block, 32);

    // This operation will be performed by only the
    // first 32-thread tile of each block
    if (my_block.thread_rank() < 32) {
        ...
        my_tile.sync();
    }
}
```

# Cooperative Groups - Particle Simulation



Phase 1: Integration



Phase 2: Collision Detection

Figure 2: Two phases of a particle simulation, with numbered arrows representing the mapping of parallel threads to particles. Note that after integration and construction of the regular grid data structure, the ordering of particles in memory and mapping to threads changes, necessitating a synchronization between phases.

# Old Implementation

```
// threads update particles in parallel  
integrate<<<blocks, threads, 0, s>>>(particles);
```

```
// Note: implicit sync between kernel launches
```

```
// Collide each particle with others in neighborhood  
collide<<<blocks, threads, 0, s>>>(particles);
```

# New Implementation

```
__global__ void particleSim(Particle *p, int N) {  
  
    grid_group g = this_grid();  
    // phase 1  
    for (i = g.thread_rank(); i < N; i += g.size())  
        integrate(p[i]);  
  
    g.sync() // Sync whole grid  
  
    // phase 2  
    for (i = g.thread_rank(); i < N; i += g.size())  
        collide(p[i], p, N);  
}
```



# CUDA 10.0

By Pramod Ramarao  
NVIDIA

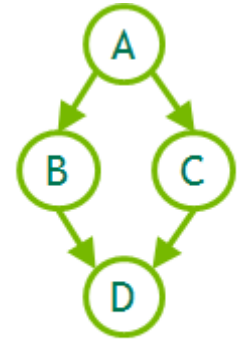
# New Features

- Support for Turing
- CUDA graphs
- New asynchronous task-graph programming model
- New profiler and debugger

# New Turing Warp Matrix Functions

	Input Precision	Output	Supported Sizes	Max Ops/Clock/SM
Native Types	half	half or float	16 x 16 x 16	1024
	char	integer (int32)	32 x 8 x 16	2048
	unsigned char		8 x 32 x 16	
Experimental	precision::u4 (4-bit unsigned)	integer (int32)	8 x 8 x 32	4096
	precision::s4 (4-bit signed)			
	precision::b1 (1-bit)		8 x 8 x 128	16384

# CUDA graphs



Workflow Graph

```
// Define graph of work + dependencies

cudaGraphCreate(&graph);
cudaGraphAddNode(graph, kernel_a, {}, ...);
cudaGraphAddNode(graph, kernel_b, { kernel_a }, ...);
cudaGraphAddNode(graph, kernel_c, { kernel_a }, ...);
cudaGraphAddNode(graph, kernel_d, { kernel_b, kernel_c }, ...);

// Instantiate graph and apply optimizations

cudaGraphInstantiate(&instance, graph);

// Launch executable graph 100 times

for(int i=0; i<100; i++)
    cudaGraphLaunch(instance, stream);
```