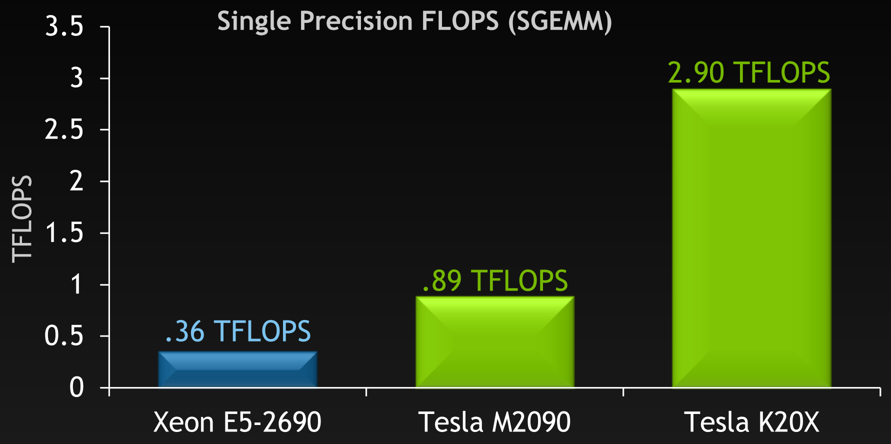
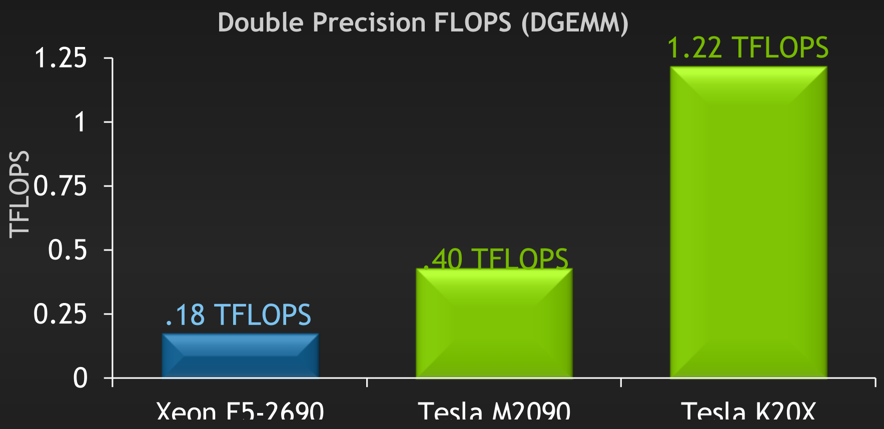
Kepler Overview

Mark Ebersole

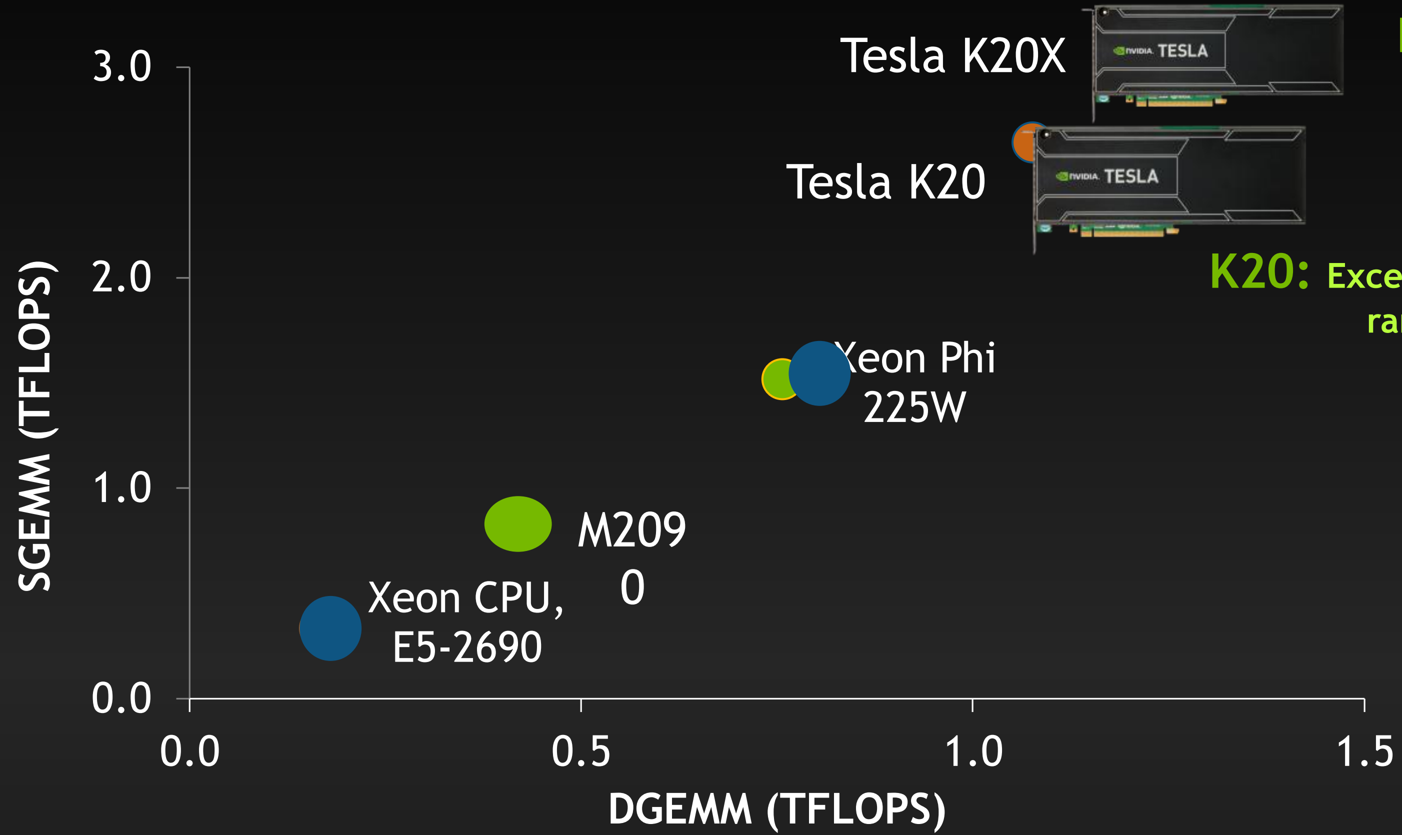
3x Performance in a Single Generation





	Tesla K20X	Tesla K20
# CUDA Cores	2688	2496
Peak Double Precision Peak DGEMM	1.32 TF 1.22 TF	1.17 TF 1.10 TF
Peak Single Precision Peak SGEMM	3.95 TF 2.90 TF	3.52 TF 2.61 TF
Memory Bandwidth	250 GB/s	208 GB/s
Memory size	6 GB	5 GB
Total Board Power	235W	225W

K20X / K20 GPUs Meant for DP Performance



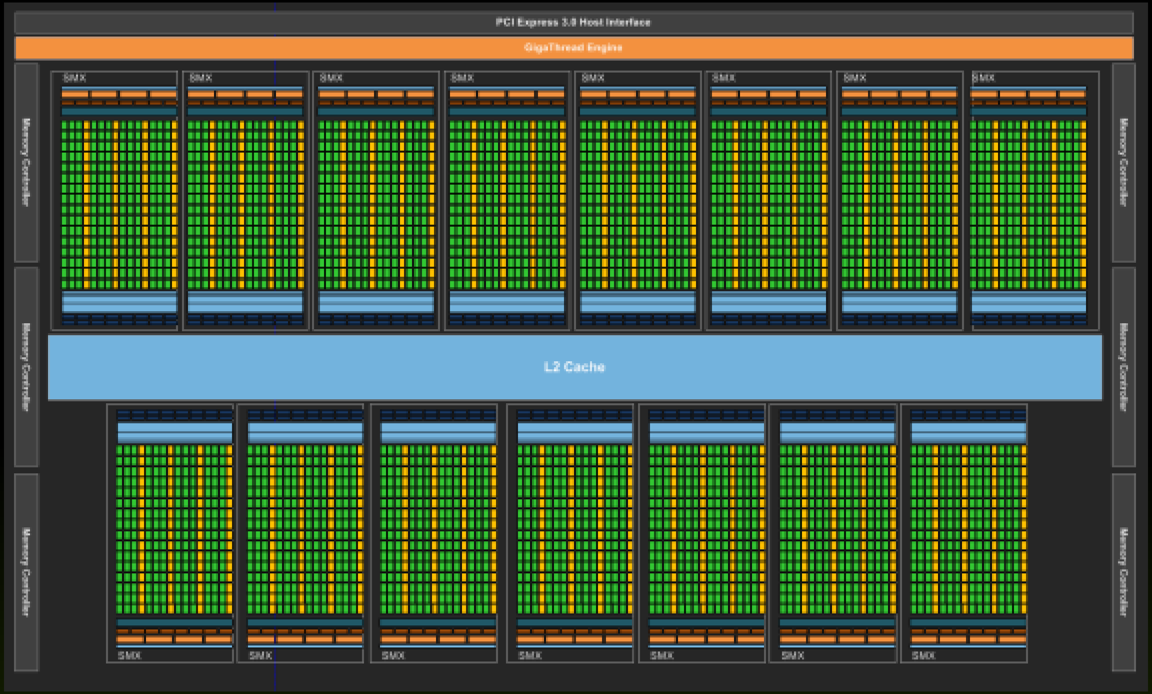
K20X : DP Performance leader for the most demanding HPC applications

K20: Excellent DP for widest range of applications

Kepler GK110 Block Diagram

Architecture

- 7.1B Transistors
- Up to 15 SMX units
- > 1 TFLOP FP64
- 1.5 MB L2 Cache
- 384-bit GDDR5



SMX Balance of Resources

Resource	Kepler GK110 vs Fermi
Floating point throughput	**2-3x**
Max Blocks per SMX	**2x**
Max Threads per SMX	**1.3x**
Register File Bandwidth	**2x**
Register File Capacity	**2x**
Shared Memory Bandwidth	**2x**
Shared Memory Capacity	**1x**

New ISA Encoding: 255 Registers per Thread

- Fermi limit: 63 registers per thread
  - A common Fermi performance limiter
  - Leads to excessive spilling
- Kepler : Up to 255 registers per thread
  - Especially helpful for FP64 apps
  - Spills are eliminated with extra registers

New High-Performance SMX Instructions

SHFL (shuffle) -- Intra-warp data exchange

ATOM -- Broader functionality, Faster

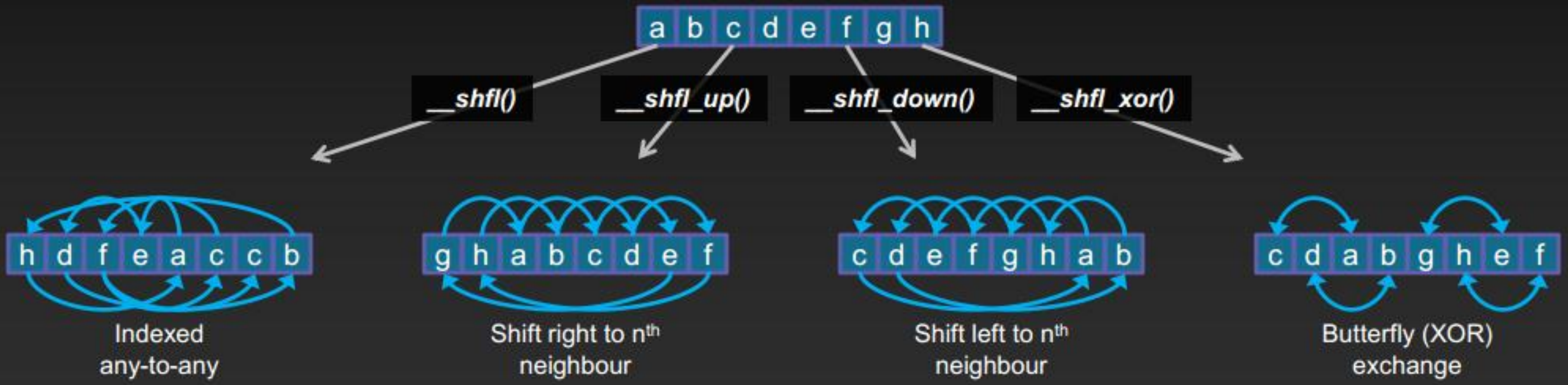
Compiler-generated, high performance instructions:

- bit shift
- bit rotate
- fp32 division
- read-only cache

New Instruction: SHFL

Data exchange between threads within a warp

- Avoids use of shared memory
- One 32-bit value per exchange
- 4 variants:



ATOM instruction enhancements

- **Added int64 functions to match existing int32**

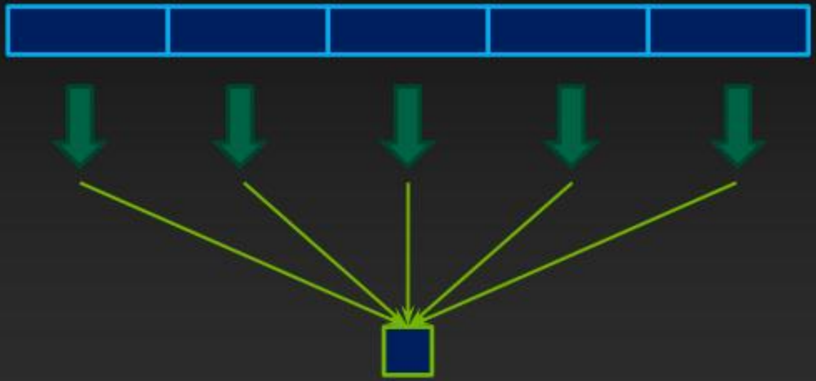
Atom Op	int32	int64
add	x	x
cas	x	x
exch	x	x
min/max	x	X
and/or/xor	x	X

- **2 – 10x performance gains**
  - **Shorter processing pipeline**
  - **More atomic processors**
  - **Slowest 10x faster**
  - **Fastest 2x faster**

High Speed Atomics Enable New Uses

Atomics are now fast enough to use within inner loops

- Example: Data reduction (sum of all values)

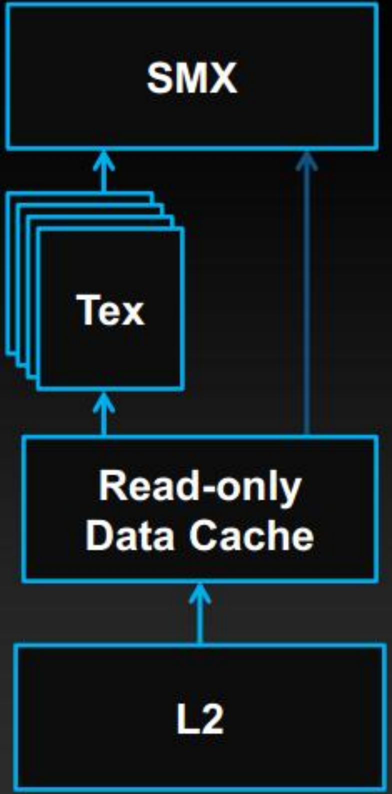


With Atomics

1. Divide input data array into N sections
2. Launch N blocks, each reduces one section
3. Write output directly via atomic. No need for second kernel launch.

Texture Cache Unlocked

- Added a new path for compute
  - Avoids the texture unit
  - Allows a global address to be fetched and cached
  - Eliminates texture setup
- Why use it?
  - Separate pipeline from shared/L1
  - Highest miss bandwidth
  - Flexible, e.g. unaligned accesses
- Managed automatically by compiler
  - “const __restrict” indicates eligibility



const __restrict Example

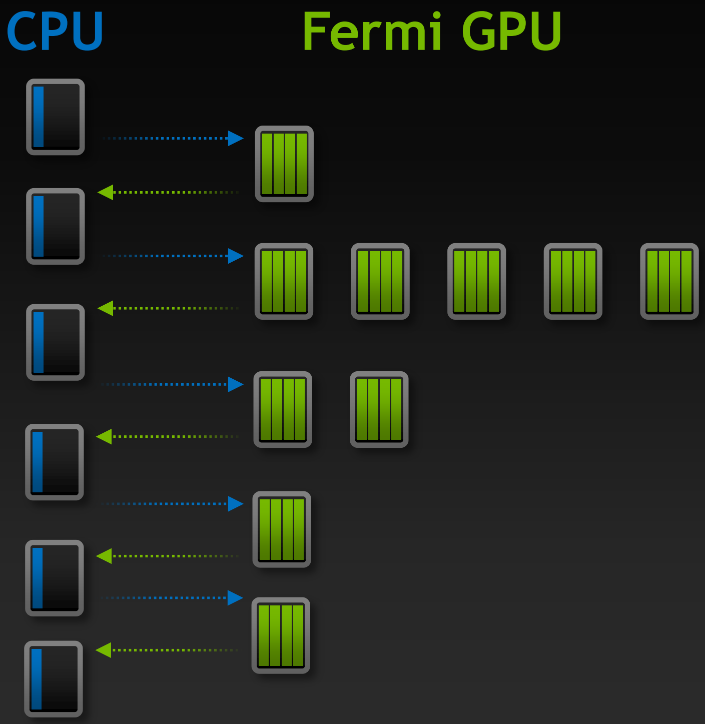
- Annotate eligible kernel parameters with `const __restrict`
- Compiler will automatically map loads to use read-only data cache path

```
__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 texture
    // for "input"
    output[offset] = (input[offset] * x) + y;
}
```

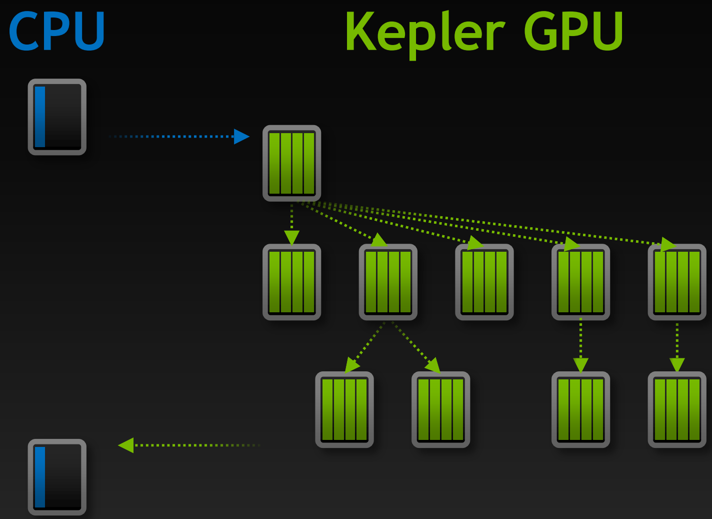
Kepler GK110 Memory System Highlights

- Efficient memory controller for GDDR5
  - Peak memory clocks achievable
- More L2
  - Double bandwidth
  - Double size
- More efficient DRAM ECC Implementation
  - DRAM ECC lookup overhead reduced by 66% (average, from a set of application traces)

Dynamic Parallelism



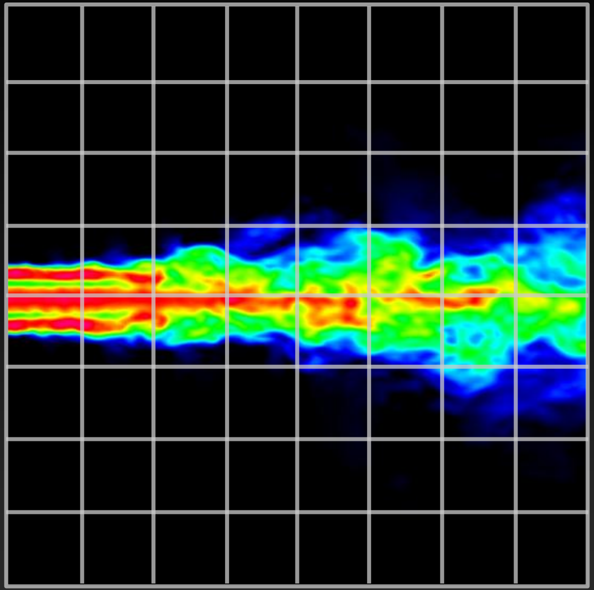
GPU as Co-Processor



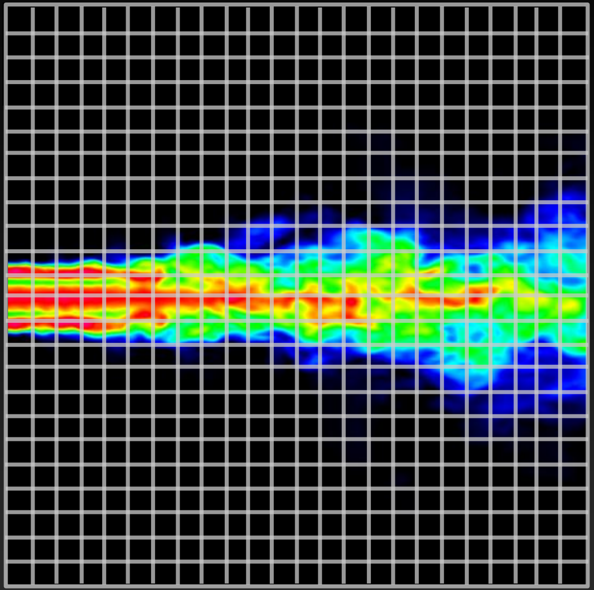
Autonomous, Dynamic Parallelism

Dynamic Parallelism

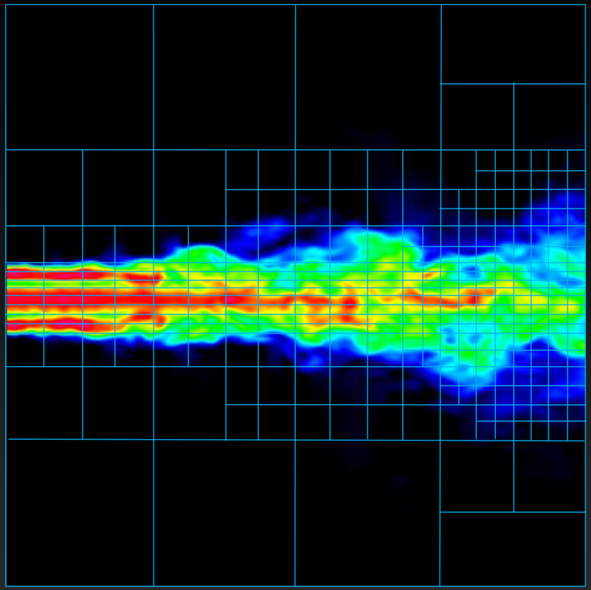
Coarse grid	Fine grid	*Dynamic grid*



Higher Performance
Lower Accuracy



Lower Performance
Higher Accuracy



Target performance where accuracy is required

Supported on GK110 GPUs

Dynamic Parallelism

- **Kernel launches grids**
- **Syntax is identical to host**
- **CUDA Runtime functions in `cudadevrt` library**

```
__global__ void childKernel()
{
    printf("Hello %d", threadIdx.x);
}

__global__ void parentKernel()
{
   childKernel<<<1,10>>>();
   cudaDeviceSynchronize();
   printf("World!\n");
}
```

```
int main(int argc, char *argv[])
{
  parentKernel<<<1,1>>>();
  cudaDeviceSynchronize();
  return 0;
}
```

Dynamic Parallelism :: nested parallelism

- Return traffic to the host after each algorithm step is <u>not</u> required to be a good case for Dynamic Parallelism
  - We often illustrate Dynamic Parallelism that way, but that’s just one example

- Look for cases of general nested parallelism as well
  - E.g., apps that don’t have enough parallelism exposed at any one place, even though in aggregate there is much more

Dynamic (Nested) Parallelism Example

```
void f(void)
{
    for (int i = 0 ; i < 12 ; i++)
        v[i].doSomething();
}

V::doSomething(void)
{
    for (int j = 0 ; j < 100 ; j++)
        x[j].innerSomething();
}

X::innerSomething(void)
{
    for (int k = 0 ; k < 29 ; k++)
        y[k].evaluate();
}
```

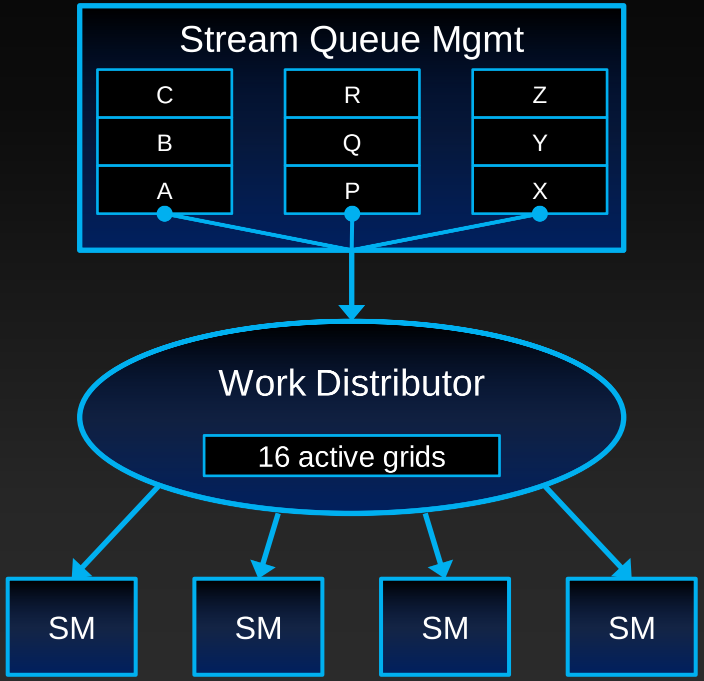
- `evaluate()` is called a total of 34800 times
- But parallelism is only exposed as 29 calls at a time
- Choices: flatten C++ hierarchy
  - Lose abstraction
  - What if functions are virtual?
- Dynamic Parallelism makes this much simpler

Dynamic (Nested) Parallelism Example

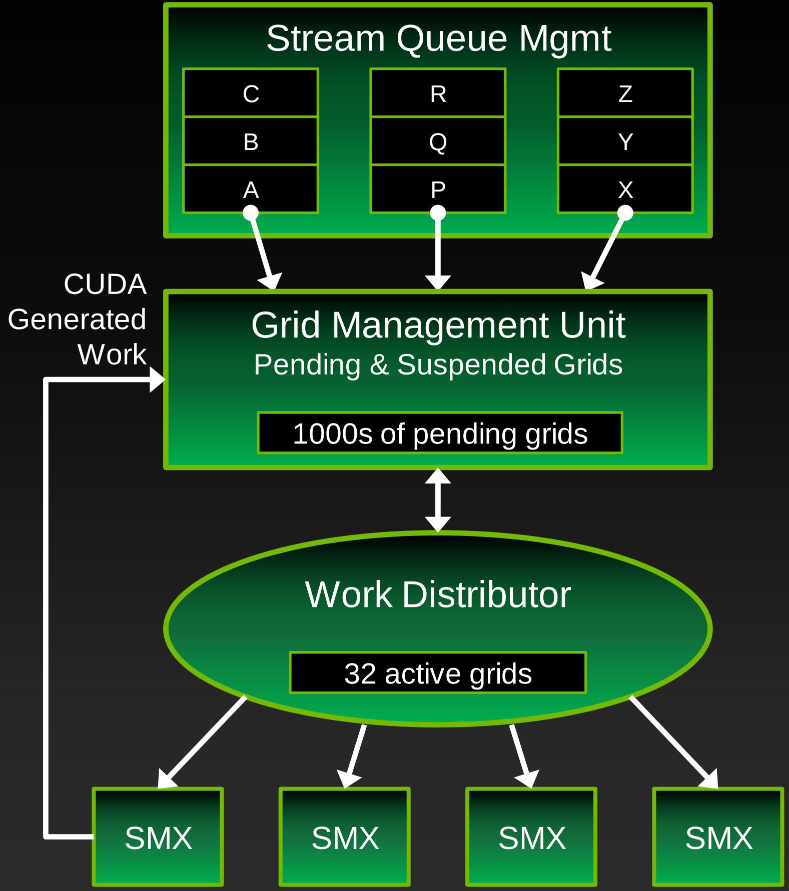
```
void f(void)
{
    for (int i = 0 ; i < 12 ; i++)
        v[i].doSomething();
}

V::doSomething(void)
{
    for (int j = 0 ; j < 100 ; j++)
        x[j].innerSomething();
}

X::innerSomething(void)
{
    for (int k = 0 ; k < 29 ; k++)
        y[k].evaluate();
}
```

```
void f(void)
{
    V::doSomething_krnl<<<1,12>>>(v);
}

__global__ V::doSomething_krnl(V *v)
{
    X::innerSomething_krnl<<<1,100>>>
        (v[threadIdx.x].x);
}

__global__ X::innerSomething_krnl(X *x)
{
    Y::evaluate_krnl<<<1,29>>>
        (x[threadIdx.x].y);
}
```

Grid Management



Fermi



Kepler GK110

Hyper-Q Enables Efficient Scheduling

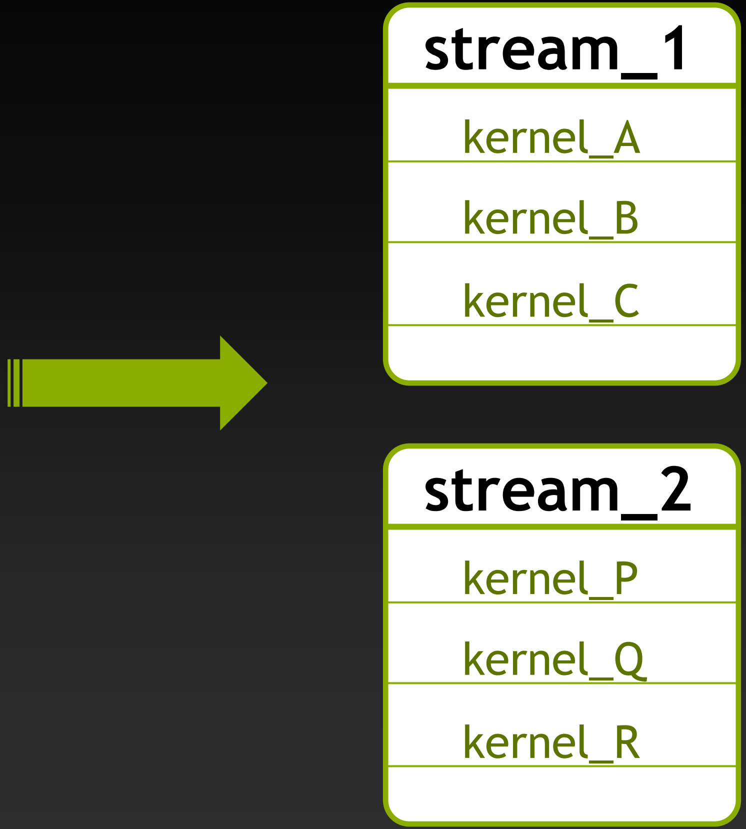
- Grid Management Unit selects most appropriate task from up to 32 hardware queues (CUDA streams)
- Improves scheduling of concurrently executed grids
- Particularly interesting for MPI applications when combined with CUDA Proxy, but *not limited to MPI applications*

Hyper-Q for non-MPI apps

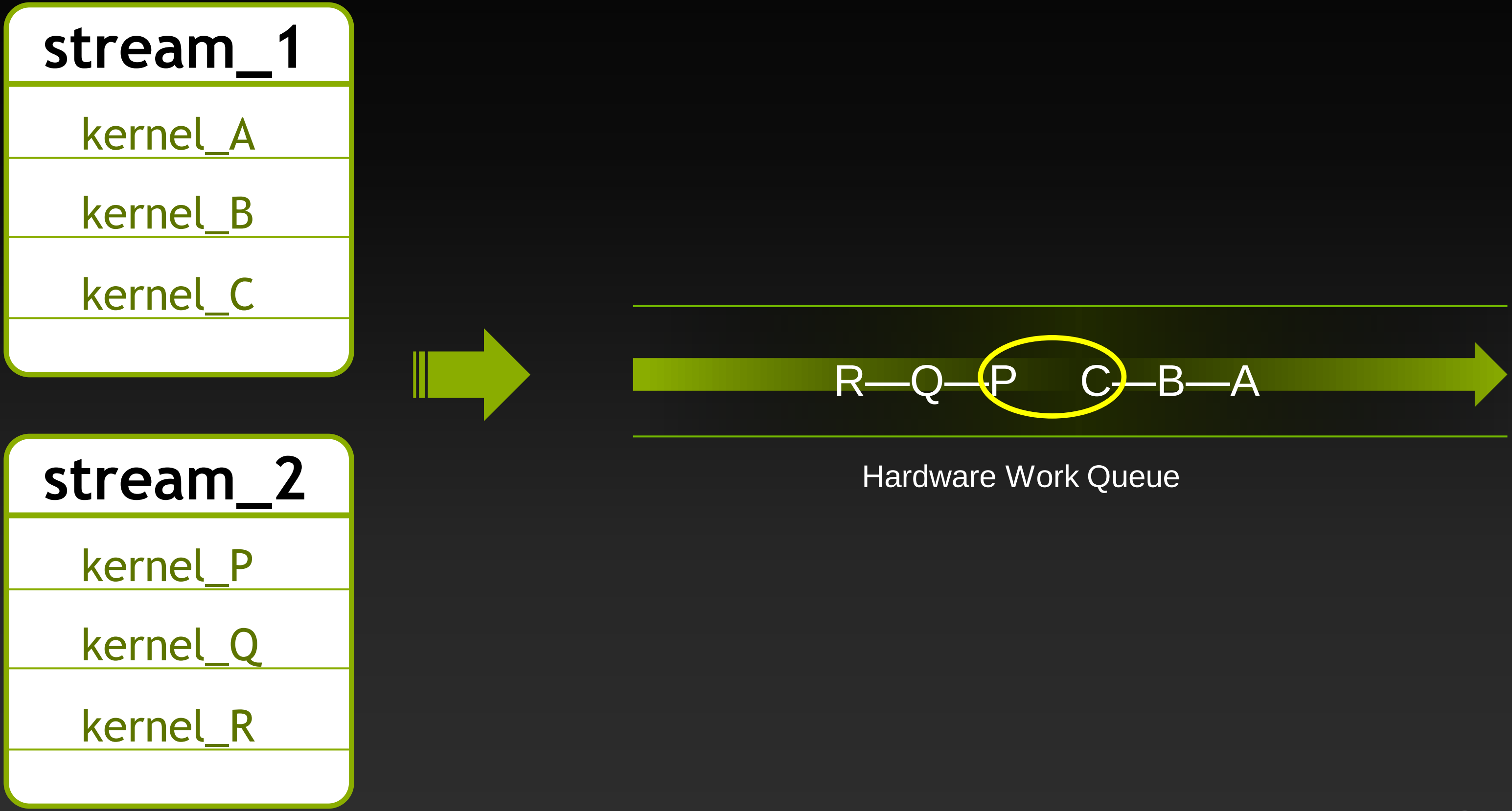
- One process: No proxy required!
  - Automatically utilized
  - One or many host threads no problem
  - Just need multiple CUDA streams
  - Removes false dependencies among CUDA streams that reduce effective concurrency on Fermi and GK104 GPUs

Stream Dependencies Example

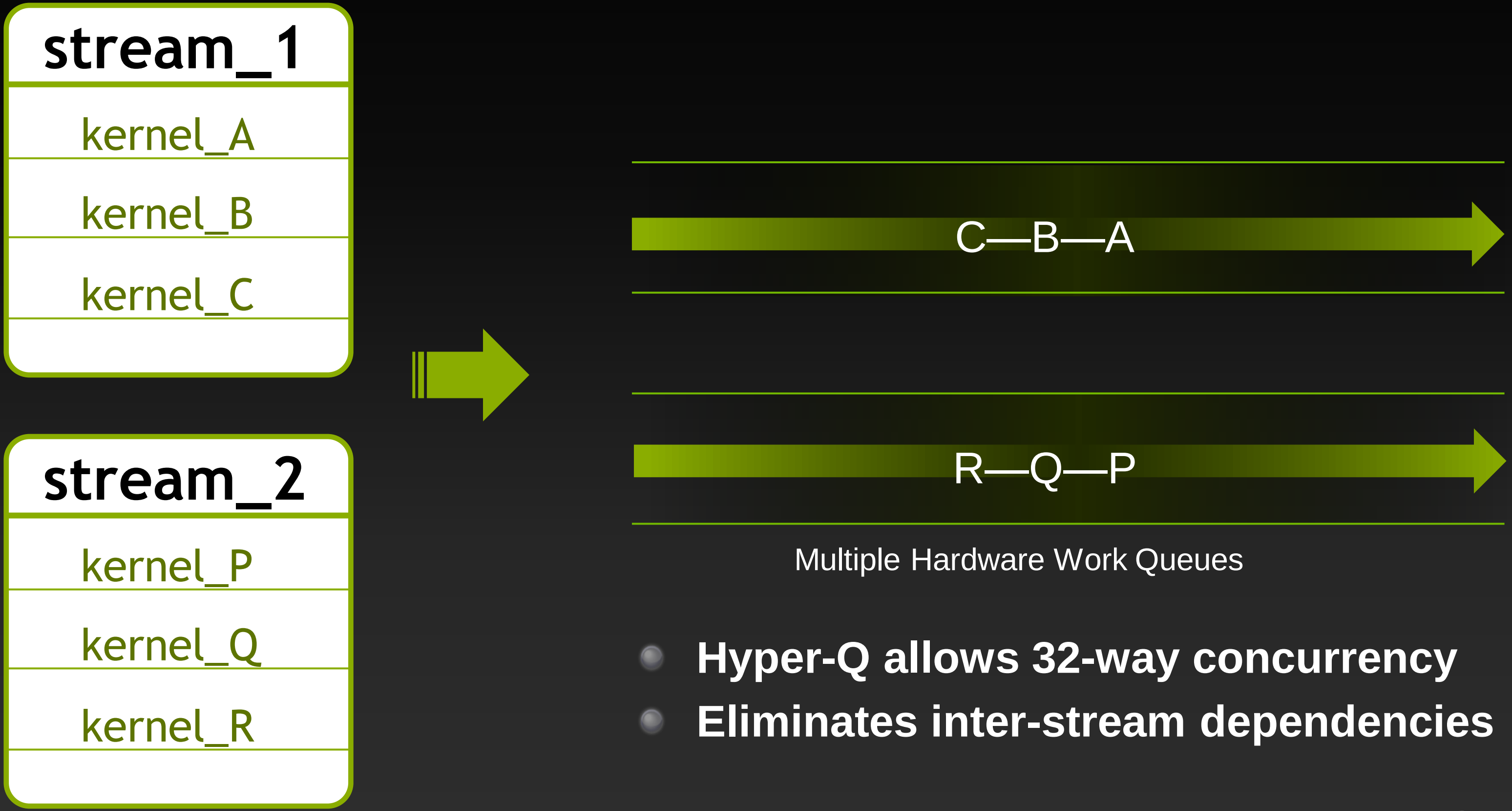
```
void foo(void)
{
    kernel_A<<<g,b,s, stream_1>>>();
    kernel_B<<<g,b,s, stream_1>>>();
    kernel_C<<<g,b,s, stream_1>>>();
}

void bar(void)
{
    kernel_P<<<g,b,s, stream_2>>>();
    kernel_Q<<<g,b,s, stream_2>>>();
    kernel_R<<<g,b,s, stream_2>>>();
}
```



Stream Dependencies without Hyper-Q

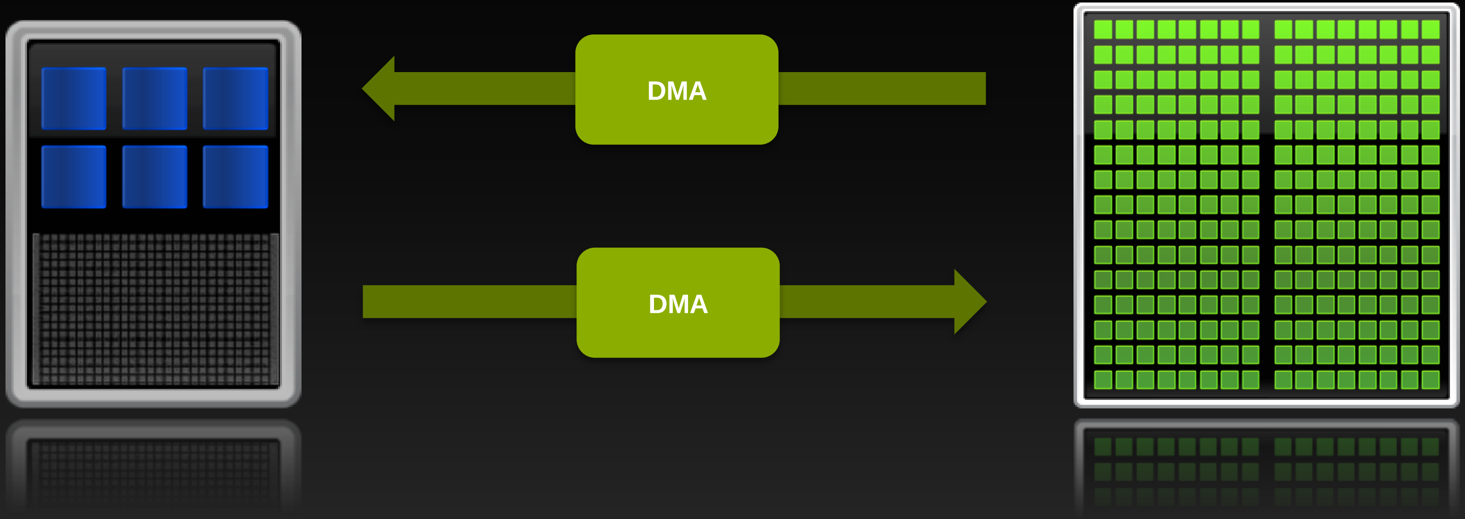


Stream Dependencies with Hyper-Q



- **Hyper-Q allows 32-way concurrency**
- **Eliminates inter-stream dependencies**

Hyper-Q Example: Building a Pipeline

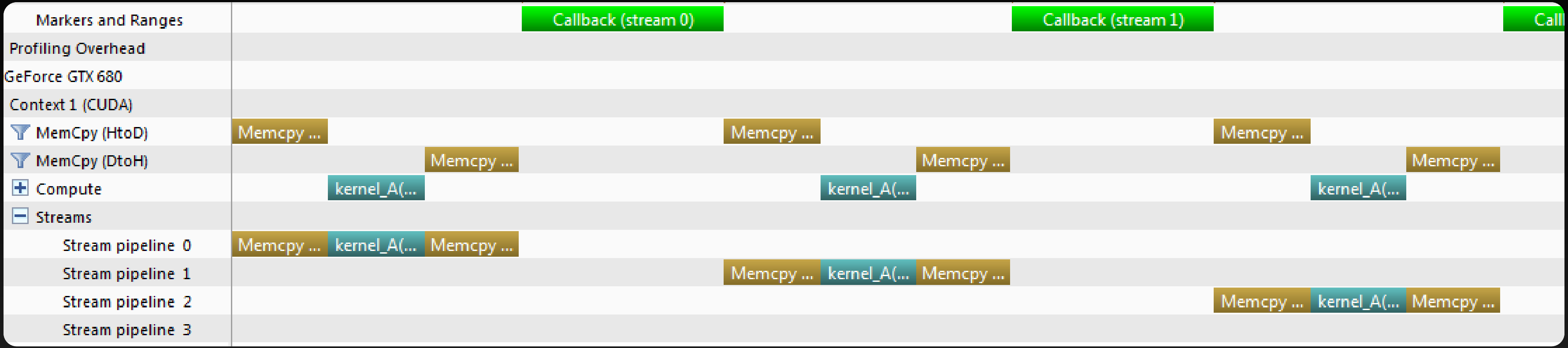


- Heterogeneous system: overlap work and data movement
- Kepler + CUDA 5: Hyper-Q and CPU Callbacks

Pipeline Code

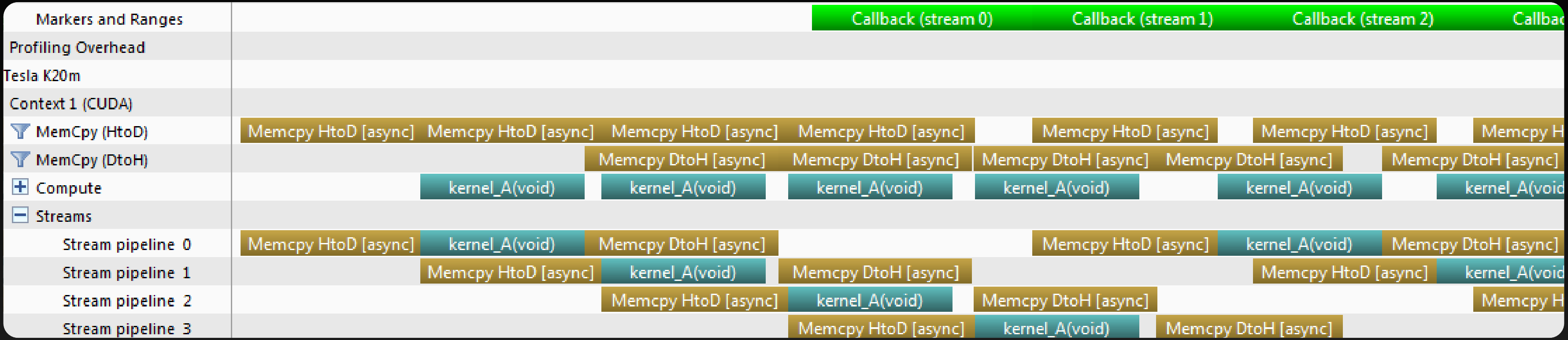
```
for (unsigned int i = 0 ; i < nIterations ; ++i)
{
    // Copy data from host to device
    chk(cudaMemcpyAsync(d_data, h_data, cpybytes, cudaMemcpyHostToDevice,
                        *r_streams.active()));

    // Launch device kernel A
    kernel_A<<<gdim, bdim, 0, *r_streams.active()>>>();

    // Copy data from device to host
    chk(cudaMemcpyAsync(h_data, d_data, cpybytes, cudaMemcpyDeviceToHost,
                        *r_streams.active()));

    // Launch host post-process
    chk(cudaStreamAddCallback(*r_streams.active(), cpu_callback,
                              r_streamids.active(), 0));

    // Rotate streams
    r_streams.rotate(); r_streamids.rotate();
}
```

Pipeline Without Hyper-Q



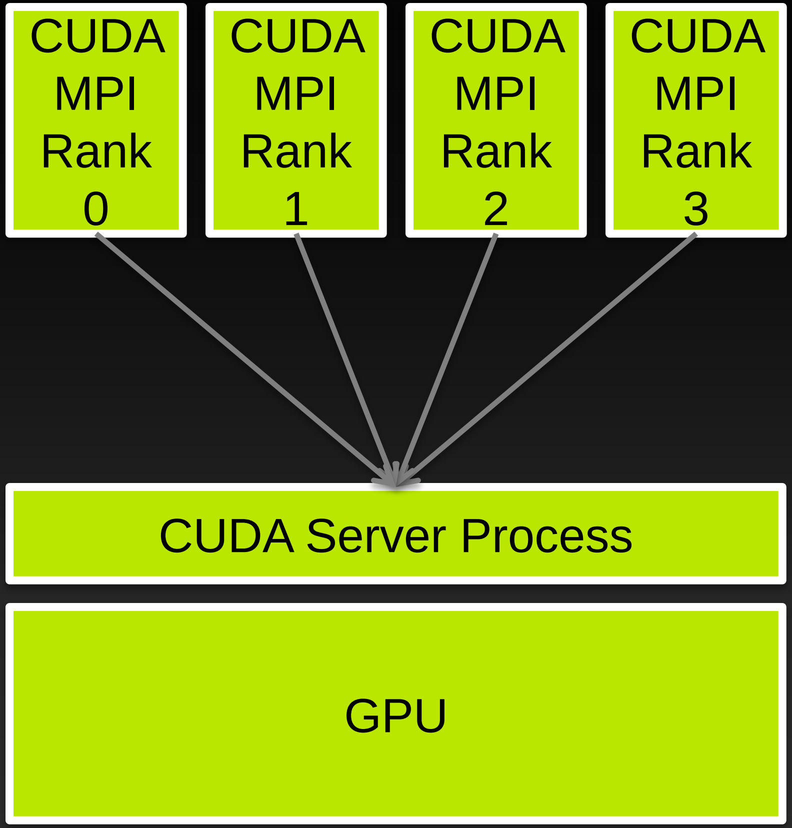
- False dependencies prevent overlap
- Breadth-first launch gives overlap, but more complex code

Pipeline With Hyper-Q



- Full overlap of all engines
- Simple to program

Multi-Process Server Required for Hyper-Q / MPI



- `$ mpirun -np 4 my_cuda_app`
  - No application re-compile to share the GPU
- No user configuration needed
  - Can be preconfigured by SysAdmin

- MPI Ranks using CUDA are clients
- Server spawns on-demand per user

- One job per user
  - No isolation between MPI ranks
  - Exclusive process mode enforces single *server*
- One GPU per rank
  - No cudaSetDevice()
  - only CUDA device 0 is visible

CUDA 5.5
Available Now

CUDA 5.5 Overview

- Linux RPM/DEB installers - bit.ly/cudacast-5
- Stream Priorities
- Dynamic Parallelism performance improvements
- MPS on Linux
- Single-GPU Debugging on Linux - bit.ly/cudacast-4
- Multi-user and remote debugging
- New Visual Profiler guided optimization
- CUFFT API Enhancements - bit.ly/cudacast-8 (CUFFTW)
- LLVM based Compiler SDK

Where to learn more

- Search for “GK110 White Paper” in your favorite search engine
- Search for “kepler” on the GTC on-demand site
    - www.gputechconf.com/gtcnew/on-demand-gtc.php
- Documentation on docs.nvidia.com
- Forums:
    - Stackoverflow.com using the CUDA tag
    - devtalk.nvidia.com

- Email me: mebersole@nvidia.com