Slide Set 18: GPUs

CPEN411

based on slides from Tor M. Aamodt

NVIDIA Tegra X1 die photo

What is a GPU?

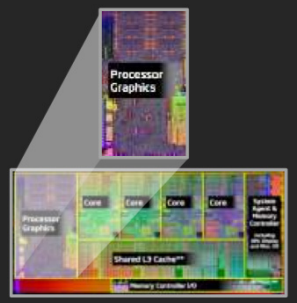
- GPU = Graphics Processing Unit
  - Accelerator for raster based graphics (OpenGL, DirectX)
  - Highly programmable (Turing complete)
  - Commodity hardware
  - 100's of ALUs; 10's of 1000s of concurrent threads

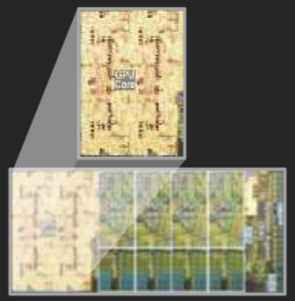
The GPU is Ubiquitous

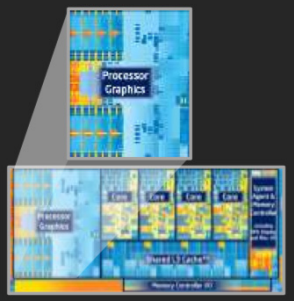
THE FUTURE BELONGS TO THE APU:
BETTER GRAPHICS, EFFICIENCY AND COMPUTE

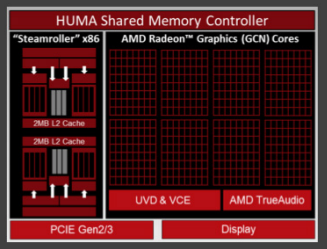
AMD

"SANDY BRIDGE"	"IVY BRIDGE"	"HASWELL"	2014 AMD A-SERIES/CODENAMED "KAVERI"
		(Estimated)	
17% GPU*	**27%** GPU*	**31%** GPU*	**47%** GPU







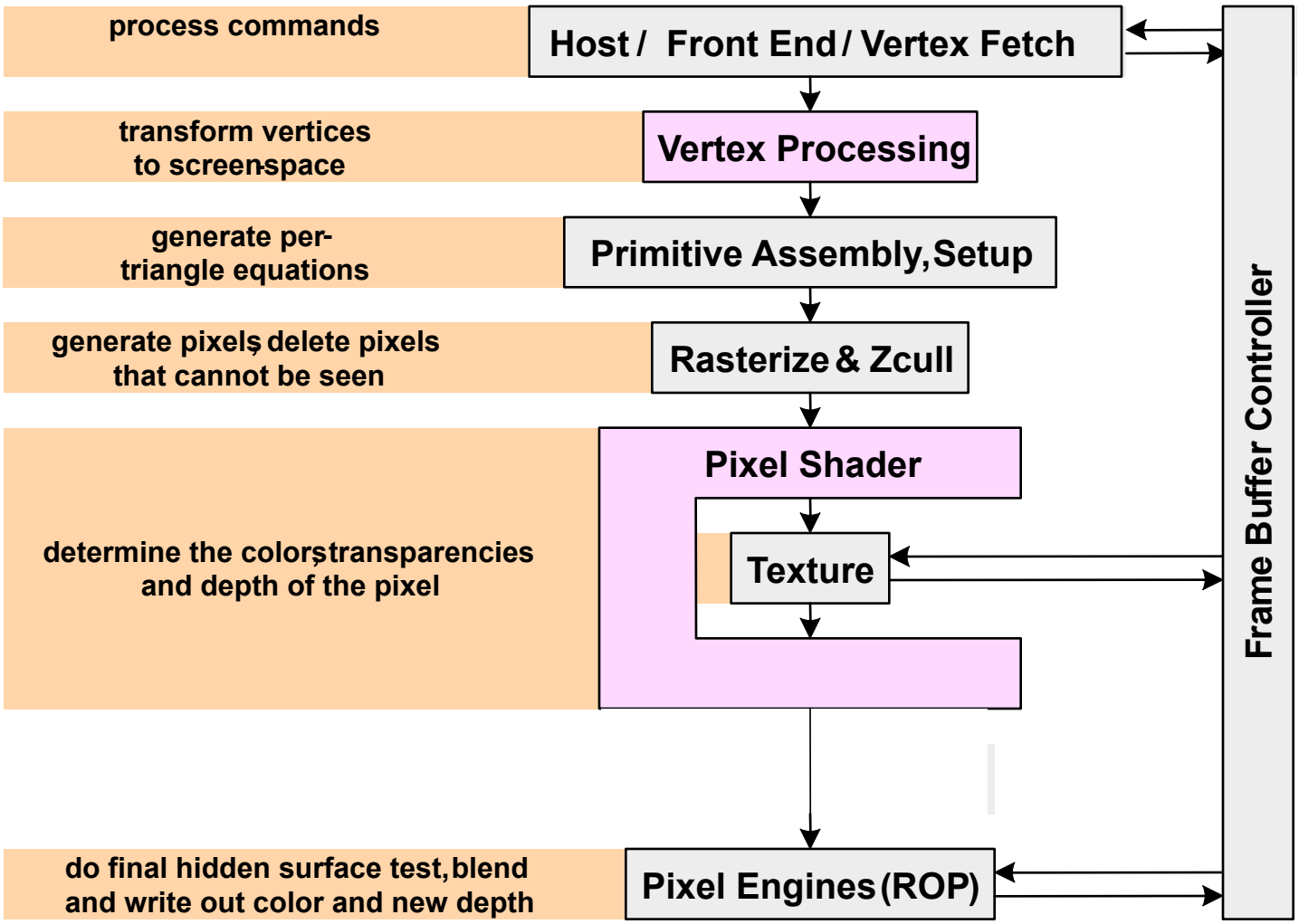


DELIVERS BREAKTHROUGHS IN APU-BASED:

- **Compute**
  - (OpenCL™, Direct Compute)
- **Gaming**
  - (DirectX®, OpenGL, Mantle)
- **Experiences**
  - (Audio, Ultra HD, Devices, New Interactivity)

[APU13 keynote]

GPU: The Life of a Triangle



[David Kirk / Wen-mei Hwu]

Not just for graphics

- Supercomputing – Green500.org Nov 2014

  "the top three slots of the Green500 were powered by three different accelerators with number one, L-CSC, being powered by AMD FirePro™ S9150 GPUs; number two, Suiren, powered by PEZY-SC many-core accelerators; and number three, TSUBAME-KFC, powered by NVIDIA K20x GPUs. Beyond these top three, the next 20 supercomputers were also accelerator-based."

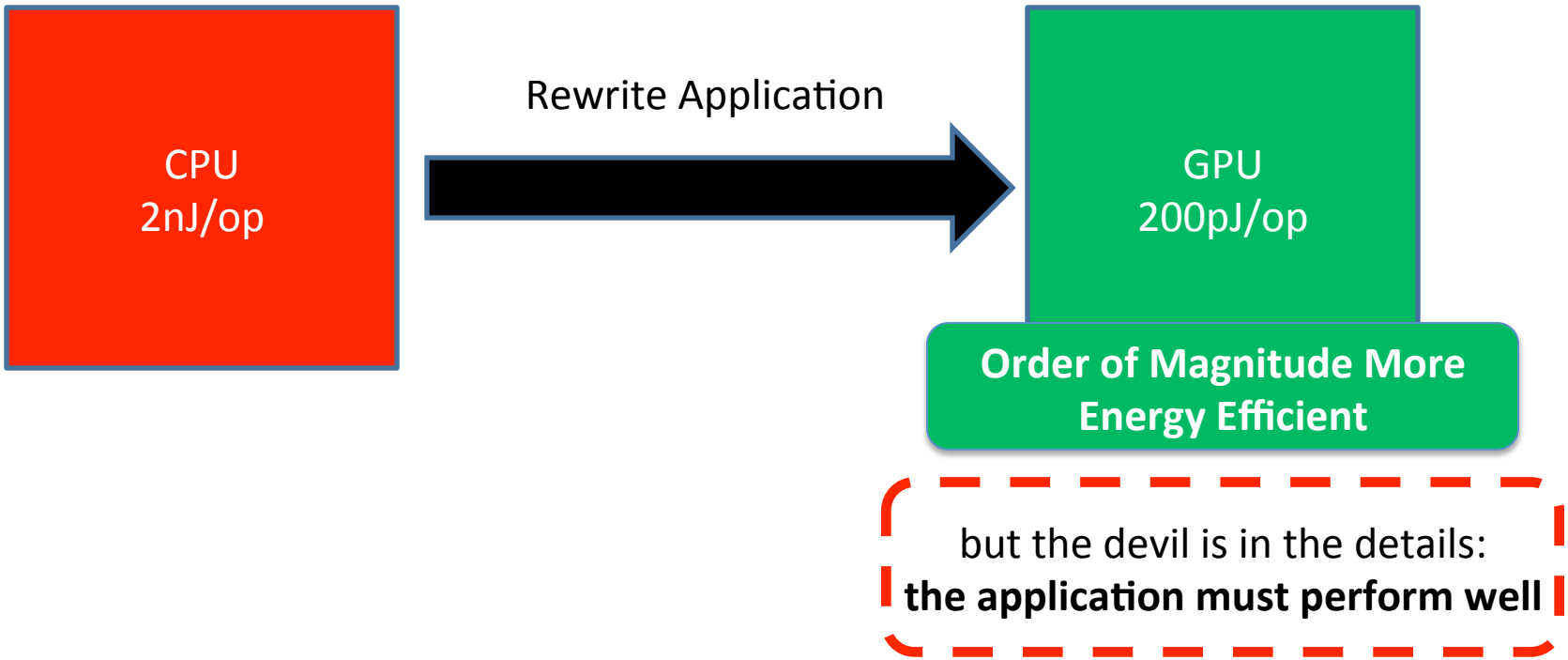
- Deep Belief Networks map *very* well to GPUs (e.g., Google keynote at 2015 GPU Tech Conf.)

  http://blogs.nvidia.com/blog/2015/03/18/google-gpu/

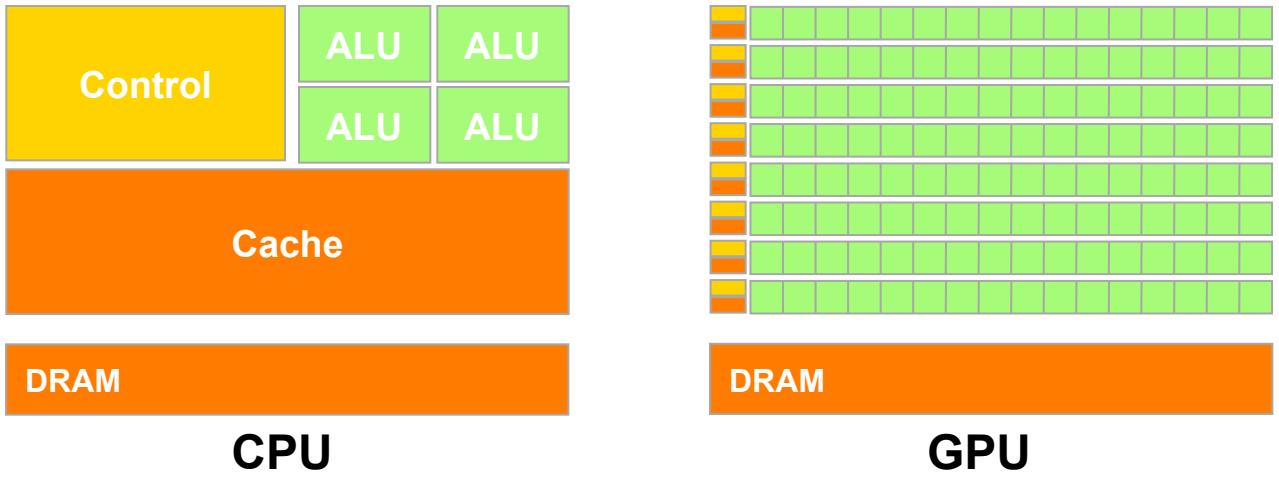
  http://www.ustream.tv/recorded/60071572

Why use a GPU for computing?

- GPU uses larger fraction of silicon for computation than CPU.
- At peak performance GPU uses order of magnitude less energy per operation than CPU.



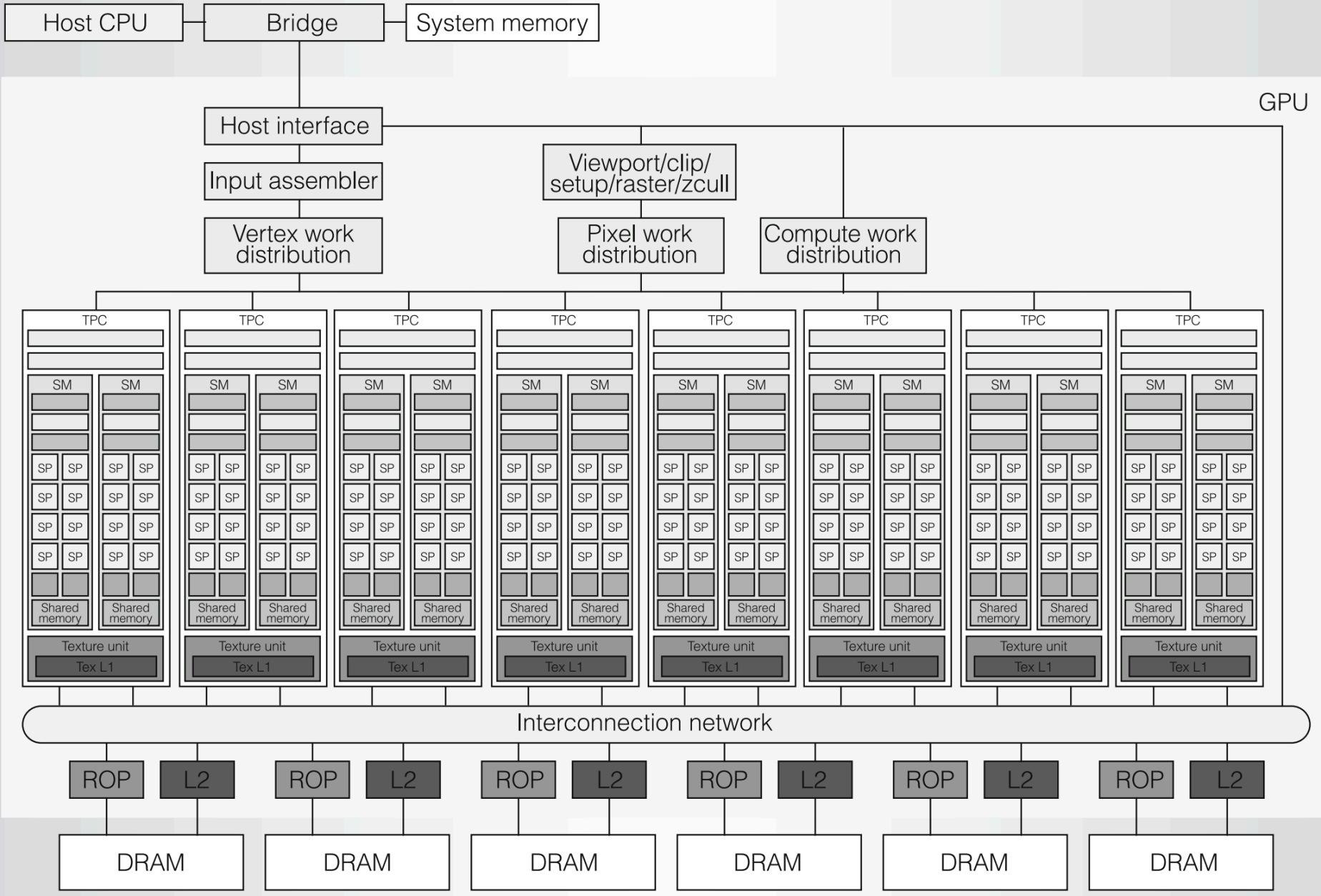
GPU uses larger fraction of silicon for computation than CPU?



[NVIDIA]

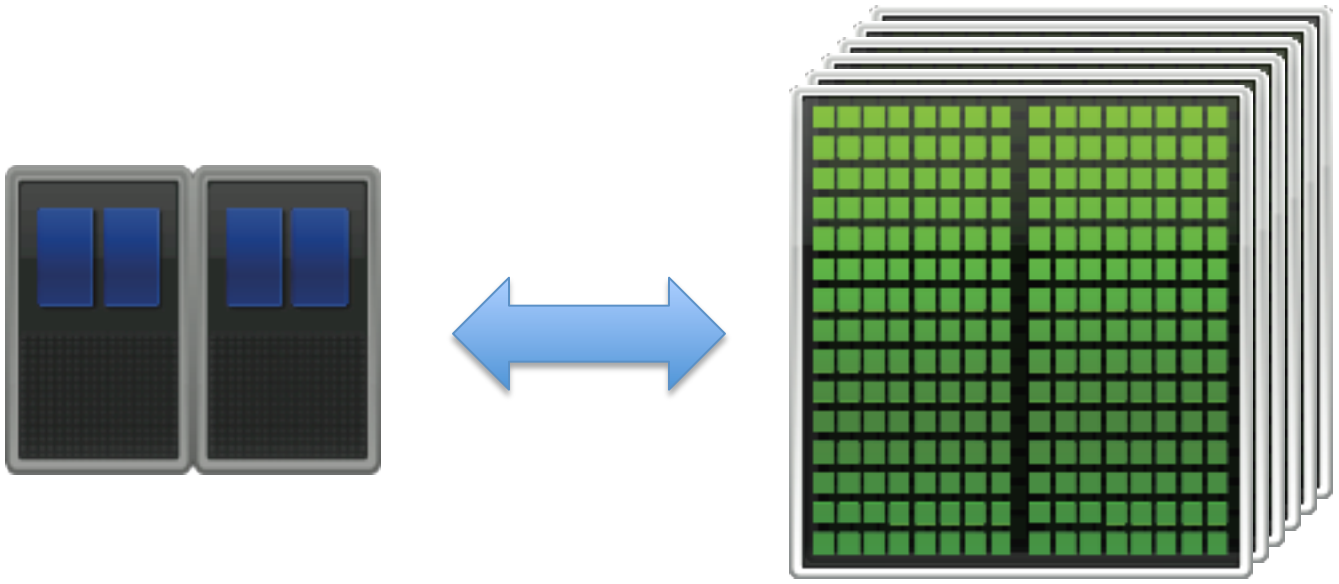
GPUs vs. Vector Processors

- GPU hardware similar to vector processors
- different *programming model*:
  - clever HW+compiler runs *scalar* threads on SIMD HW
  - recall that Tomasulo+ROB executes *sequential* code on *parallel* (OoO superscalar) hardware
- leaves the (hard) work of *detecting parallelism* to the programmer



NVidia Tesla architecture (Lindholm et al, 2008)

GPU Compute Programming Model

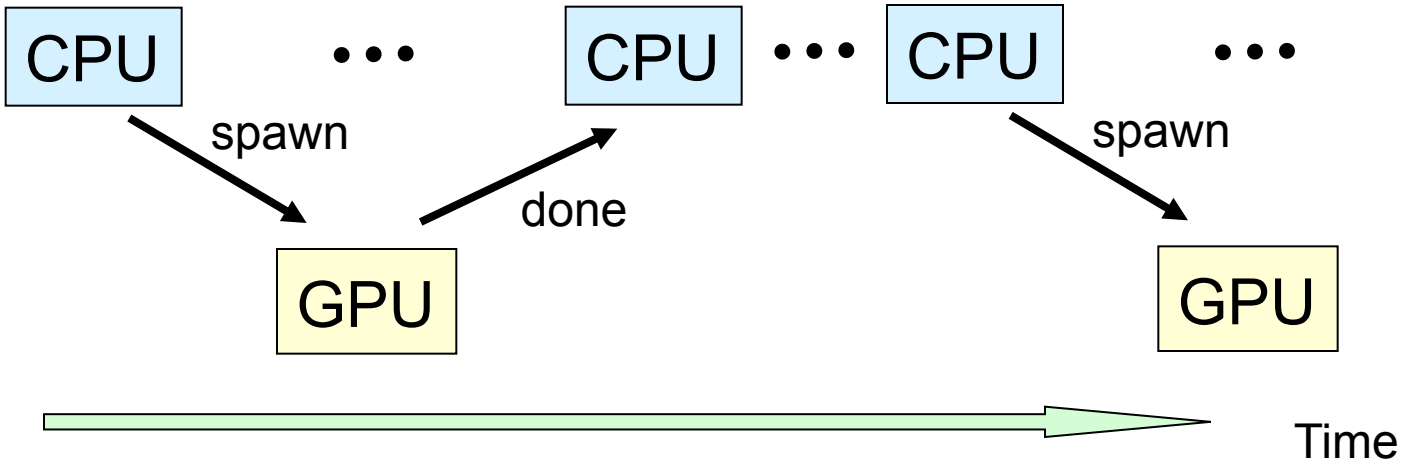


CPU GPU

How is this system programmed (today)?

GPGPU Programming Model

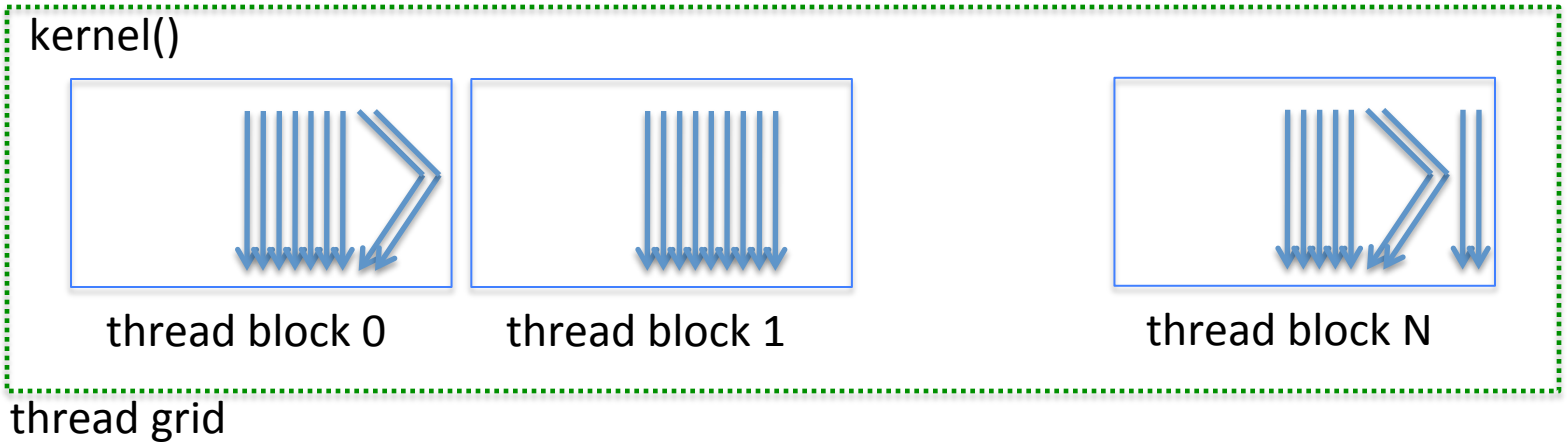
- CPU “Off-load” parallel kernels to GPU



  - Transfer data to GPU memory
  - GPU HW spawns threads
  - Need to transfer result data back to CPU main memory

CUDA/OpenCL Threading Model

CPU spawns fork-join style “grid” of parallel threads

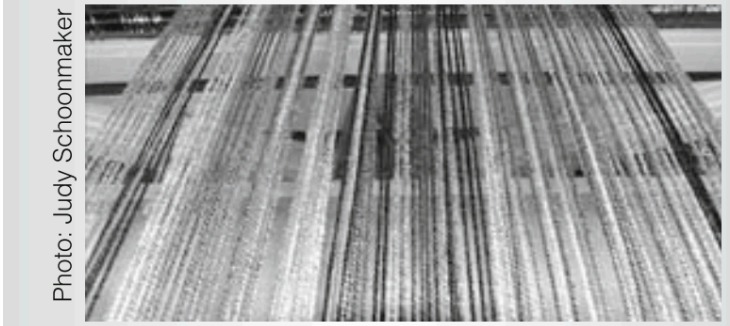


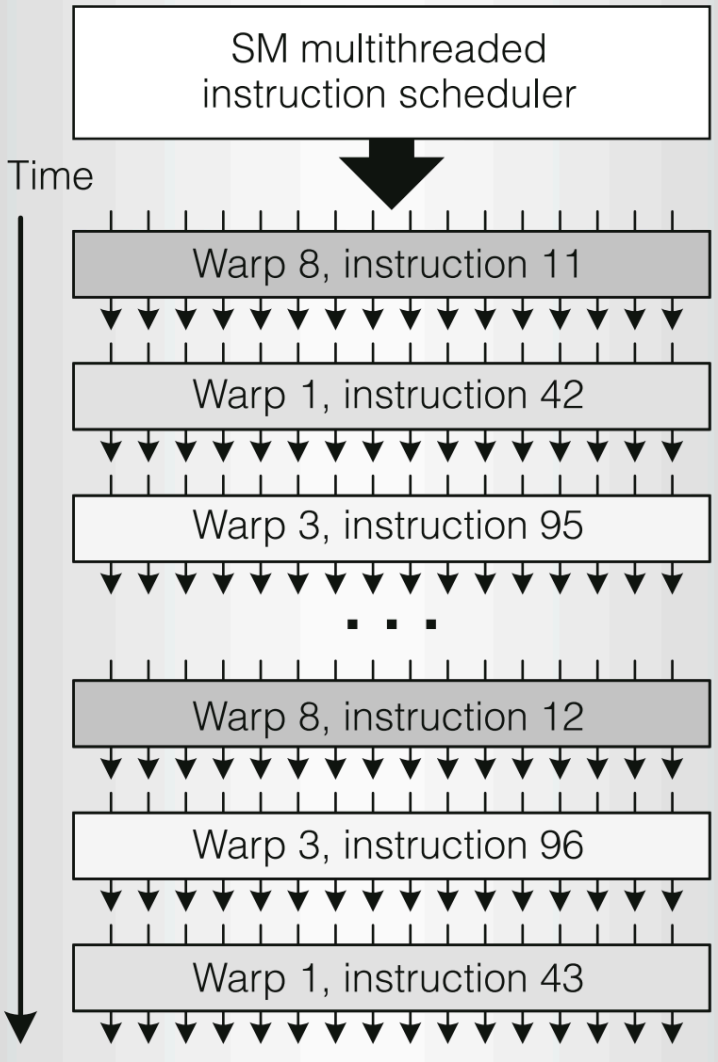
- spawns more threads than GPU can run (some may wait)
- organize threads into “blocks” (up to 1024 threads per block)
- threads can communicate/synchronize with other threads in block
- threads/Blocks have an identifier (can be 1, 2 or 3 dimensional)
- each kernel spawns a “grid” containing 1 or more thread blocks.
- motivation: Write parallel software once and run on future hardware

SIMT Execution

- programmer sees **scalar** threads
- GPU bundles threads into warps (wavefronts) and runs them in lockstep on SIMD hardware
- a warp groups 32–64 threads
- **no inter-thread dependencies**
  - can schedule threads into warps
- care about **throughput** not latency
  - 1000s of threads can cover long memory latencies

(Lindholm et al, 2008)

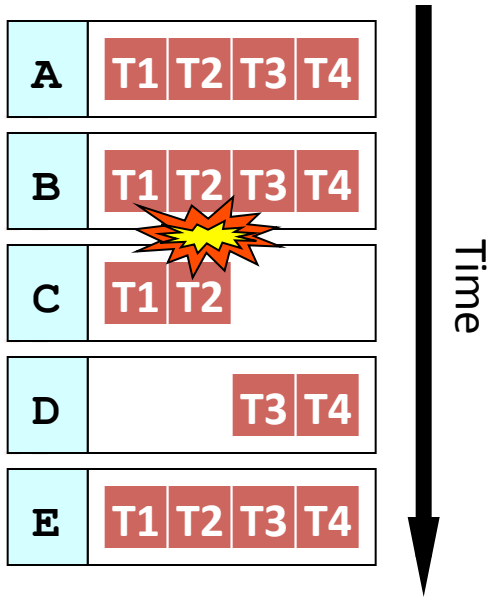




Branch divergence

- Challenge: How to handle branch operations when different threads in a warp follow a different path through program?
- Solution: **serialize** different paths (we'll see how later)

```
foo[] = {4,8,12,16};
A: v = foo[threadIdx.x];
B: if (v < 10)
C:     v = 0;
   else
D:     v = 10;
E: w = bar[threadIdx.x]+v;
```



CUDA Syntax Extensions

- Declaration specifiers

  __global__ void foo(...); // kernel entry point (runs on GPU)

  __device__ void bar(...); // function callable from a GPU thread

- Syntax for kernel launch

  foo<<<500, 128>>>(...); // 500 thread blocks, 128 threads each

- Built in variables for thread identification

  dim3 threadIdx; dim3 blockIdx; dim3 blockDim;

Example: Original C Code

```
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];
}

int main() {
  // omitted: allocate and initialize memory
  saxpy_serial(n, 2.0, x, y); // Invoke serial SAXPY kernel
  // omitted: using result
}
```

CUDA Code

```
__global__ void saxpy(int n, float a, float *x, float *y) {
   int i = blockIdx.x*blockDim.x + threadIdx.x;
   if(i<n) y[i]=a*x[i]+y[i];
}
```

Runs on GPU

```
int main() {
  // omitted: allocate and initialize memory
  int nblocks = (n + 255) / 256;

  cudaMalloc((void**) &d_x, n);
  cudaMalloc((void**) &d_y, n);
  cudaMemcpy(d_x,h_x,n*sizeof(float),cudaMemcpyHostToDevice);
  cudaMemcpy(d_y,h_y,n*sizeof(float),cudaMemcpyHostToDevice);
  saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);
  cudaMemcpy(h_y,d_y,n*sizeof(float),cudaMemcpyDeviceToHost);
  // omitted: using result
}
```

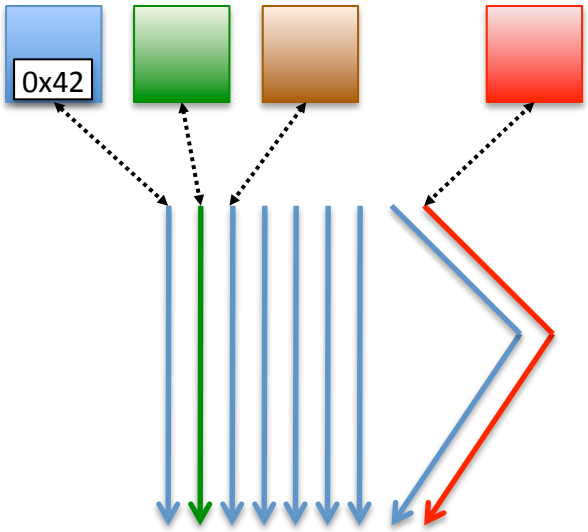
GPU Memory Address Spaces

- GPU has three *address spaces* to support increasing visibility of data among threads: **local**, **shared**, **global**

- plus two more (read-only) address spaces: **constant** and **texture**

Local (private) Address Space

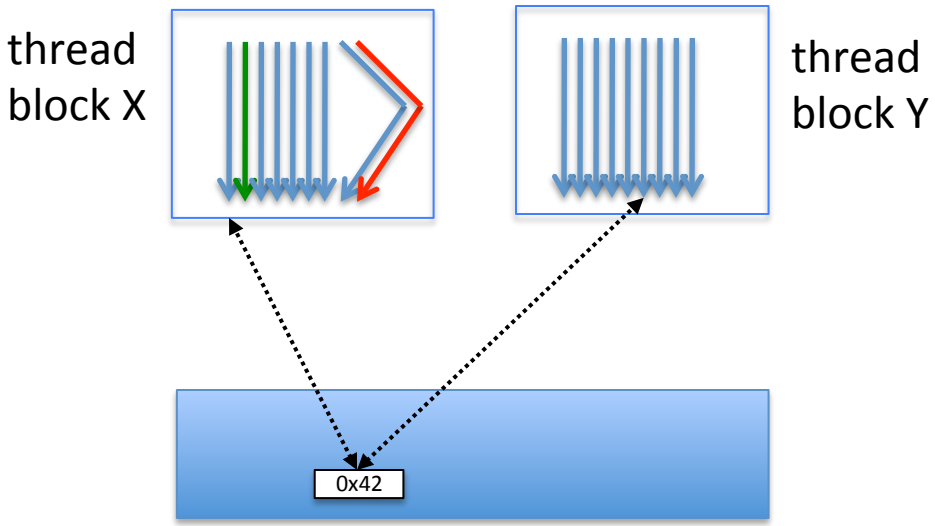
each thread has own “local memory” (CUDA)
a.k.a. “private memory” (OpenCL).



note: Location at address 100 for thread 0 is different from location at address 100 for thread 1

contains local variables private to a thread

Global Address Spaces



Each thread in the different thread blocks (even from different kernels) can access a region called “global memory” (CUDA/OpenCL).

Commonly in GPGPU workloads threads write their own portion of global memory. Avoids need for synchronization—slow; also unpredictable thread block scheduling.

History of “global memory”

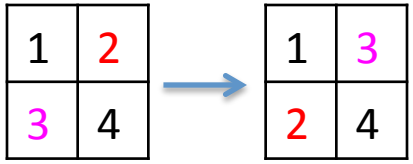
- prior to NVIDIA GeForce 8800 and CUDA 1.0, access to memory was through texture reads and raster operations for writing.

- problem: address of memory access was highly constrained function of thread ID.

- CUDA 1.0 enabled access to arbitrary memory location in a flat memory space called “global”

Example: Transpose (CUDA SDK)

```
__global__ void transposeNaive(float *odata, float* idata, int width, int height)
{
  int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;  // TILE_DIM = 16
  int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;

  int index_in  = xIndex + width * yIndex;
  int index_out = yIndex + height * xIndex;
  for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) { // BLOCK_ROWS = 16
    odata[index_out+i] = idata[index_in+i*width];
  }
}
```

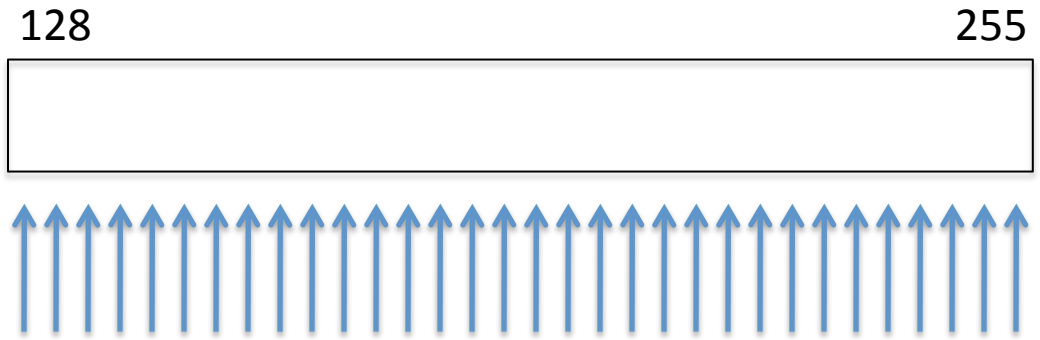


NOTE: “xIndex”, “yIndex”, “index_in”, “index_out”, and “i” are in local memory
(local variables are register allocated but stack lives in local memory)

“odata” and “idata” are pointers to global memory
(both allocated using calls to cudaMalloc -- not shown above)

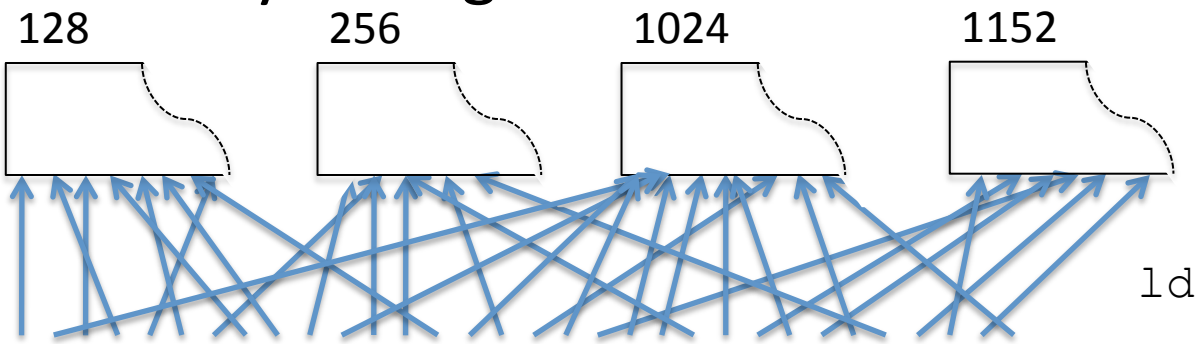
“Coalescing” global accesses

- Not same as CPU write combining/buffering:
- Aligned accesses request single 128B cache blk




```
ld.global r1,0(r2)
```

- Memory Divergence:




```
ld.global r1,0(r2)
```

Example: Transpose (CUDA SDK)

```
__global__ void transposeNaive(float *odata, float* idata, int width, int height)
{
  int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;

  int index_in  = xIndex + width * yIndex;
  int index_out = yIndex + height * xIndex;
  for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
    odata[index_out+i] = idata[index_in+i*width];
  }
}
```

Assume height=16 and consider i=0:

Thread x=0,y=0 has xIndex=0, yIndex=0 so accesses odata[0]
Thread x=1,y=0 has xIndex=1, yIndex=0 so accesses odata[16]

Write to global memory highlighted above is not “coalesced”.

Redundant Global Memory Accesses

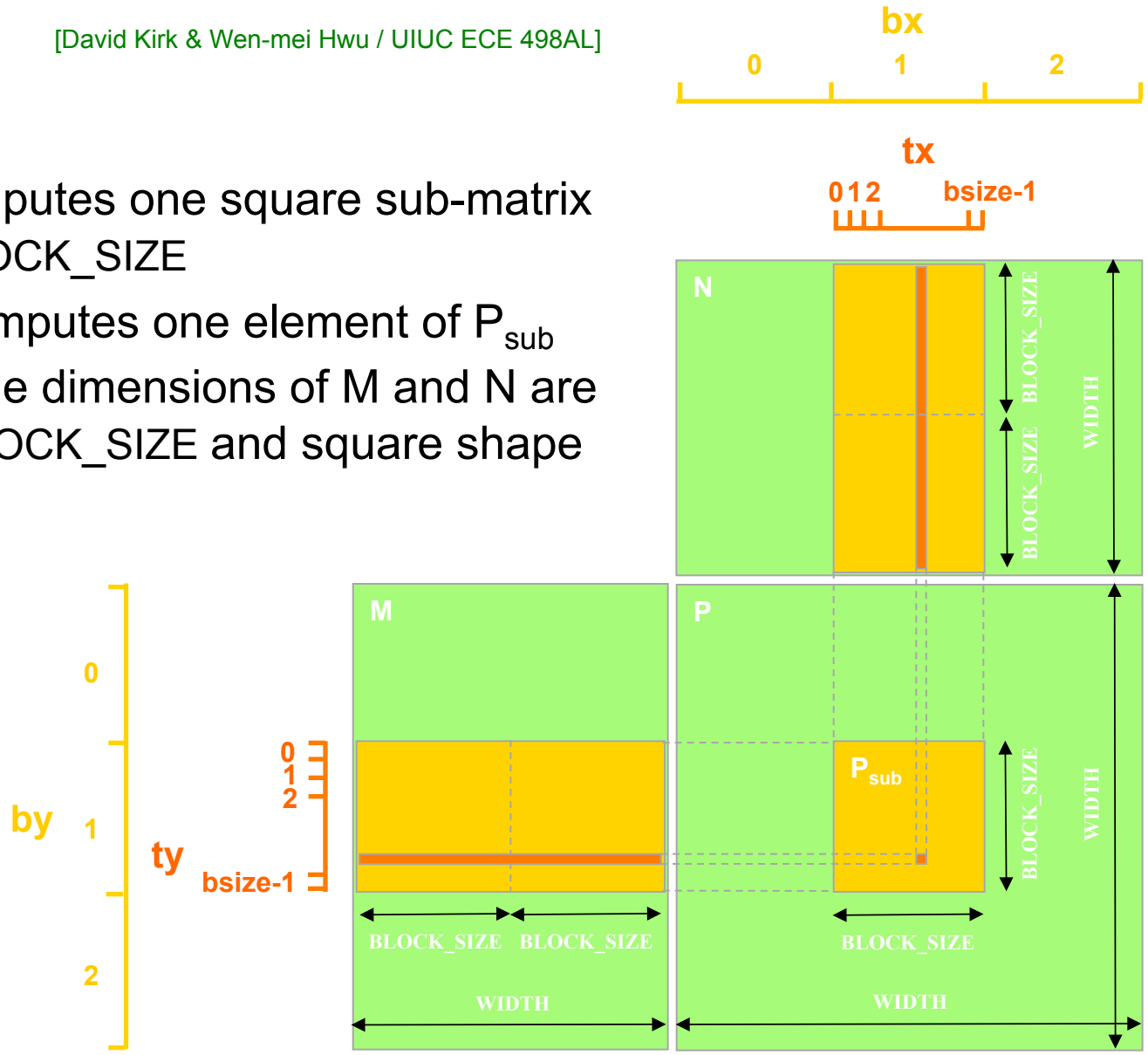
```
__global__ void matrixMul (float *C, float *A, float *B, int N)
{
  int xIndex = blockIdx.x * BLOCK_SIZE + threadIdx.x;
  int yIndex = blockIdx.y * BLOCK_SIZE + threadIdx.y;

  float sum = 0;

  for (int k=0; k<N; i++)
    sum += A[yIndex][k] * B[k][xIndex];

  C[yIndex][xIndex] = sum;
}
```

E.g., both thread x=0,y=0 and thread x=32, y=0 access A[0][0] potentially causing two accesses to off-chip DRAM. In general, each element of A and B is redundantly fetched O(N) times.

Tiled Multiply Using Thread Blocks

[David Kirk & Wen-mei Hwu / UIUC ECE 498AL]

- One block computes one square sub-matrix P_{sub} of size BLOCK_SIZE
- One thread computes one element of P_{sub}
- Assume that the dimensions of M and N are multiples of BLOCK_SIZE and square shape

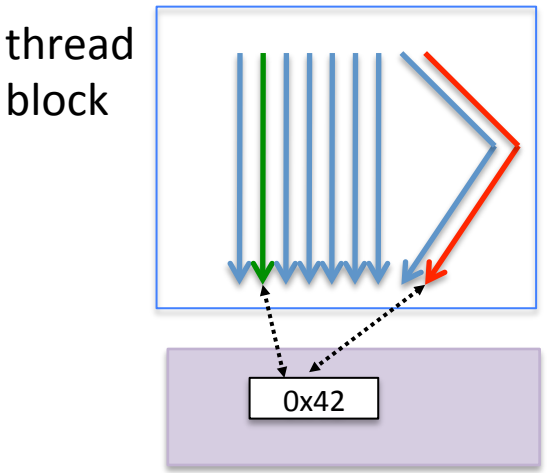


History of “shared memory”

- Prior to NVIDIA GeForce 8800 and CUDA 1.0, threads could not communicate with each other through on-chip memory.

- “Solution”: small (16-48KB) programmer managed scratchpad memory shared between threads within a thread block.

Shared (Local) Address Space



Each thread in the same thread block (work group) can access a memory region called “shared memory” (CUDA) “local memory” (OpenCL).

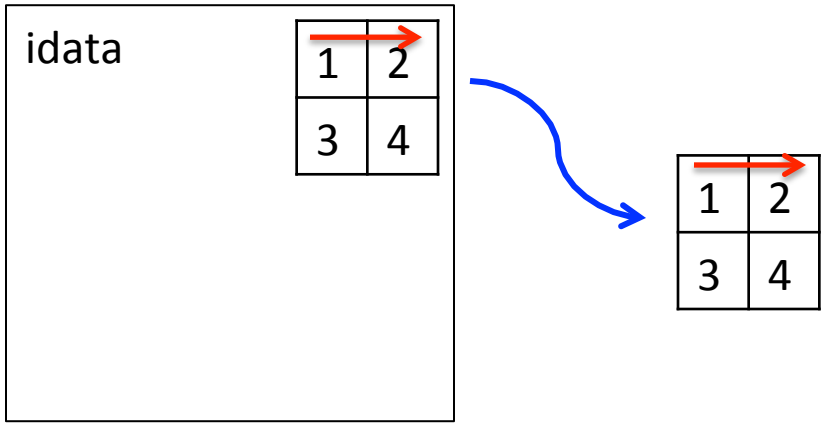
Shared memory address space is limited in size (16 to 48 KB).

Used as a software managed “cache” to avoid off-chip memory accesses.

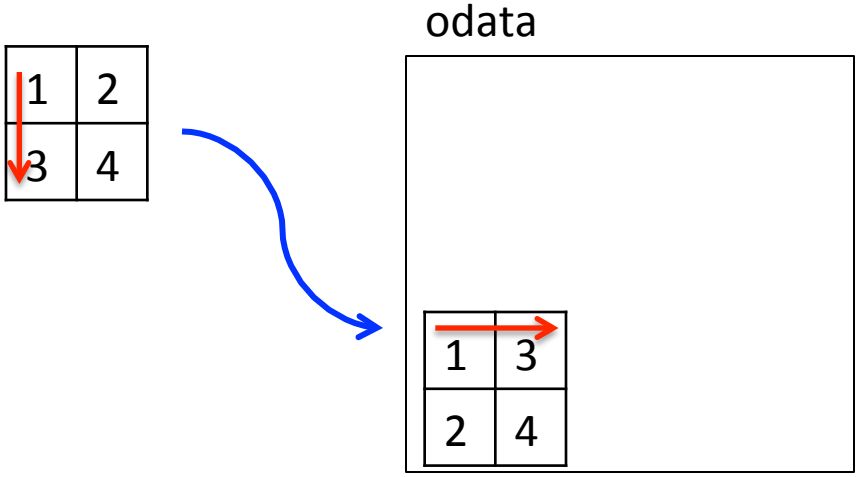
Synchronize threads in a thread block using __syncthreads();

Optimizing Transpose for Coalescing

Step 1: Read block of data into shared memory



Step 2: Copy from shared memory into global memory using coalesce write



Optimizing Transpose for Coalescing

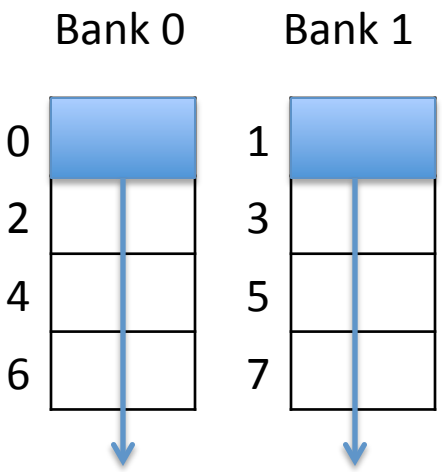
```
__global__ void transposeCoalesced(float *odata, float *idata, int width, int height)
{
  __shared__ float tile[TILE_DIM][TILE_DIM];

  int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
  int index_in = xIndex + (yIndex)*width;

  xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
  yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
  int index_out = xIndex + (yIndex)*height;

  for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
    tile[threadIdx.y+i][threadIdx.x] = idata[index_in+i*width];
  }

  __syncthreads();  // wait for all threads in block to finish above for loop

  for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
    odata[index_out+i*height] = tile[threadIdx.x][threadIdx.y+i];
  }
}
```

GOOD: Coalesced write

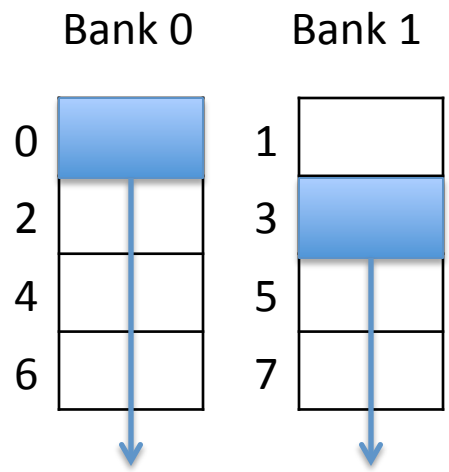
BAD: Shared memory bank conflicts

Review: Bank Conflicts

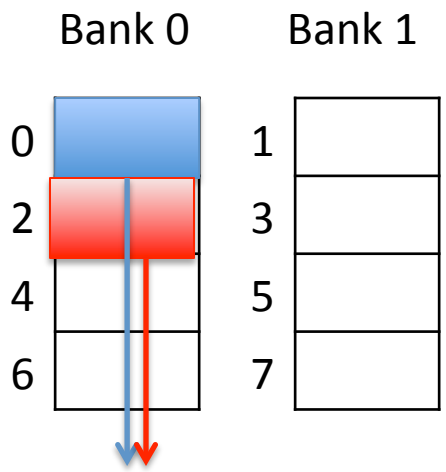
- To increase bandwidth common to organize memory into multiple banks.
- Independent accesses to different banks can proceed in parallel



Example 1: Read 0, Read 1
(can proceed in parallel)



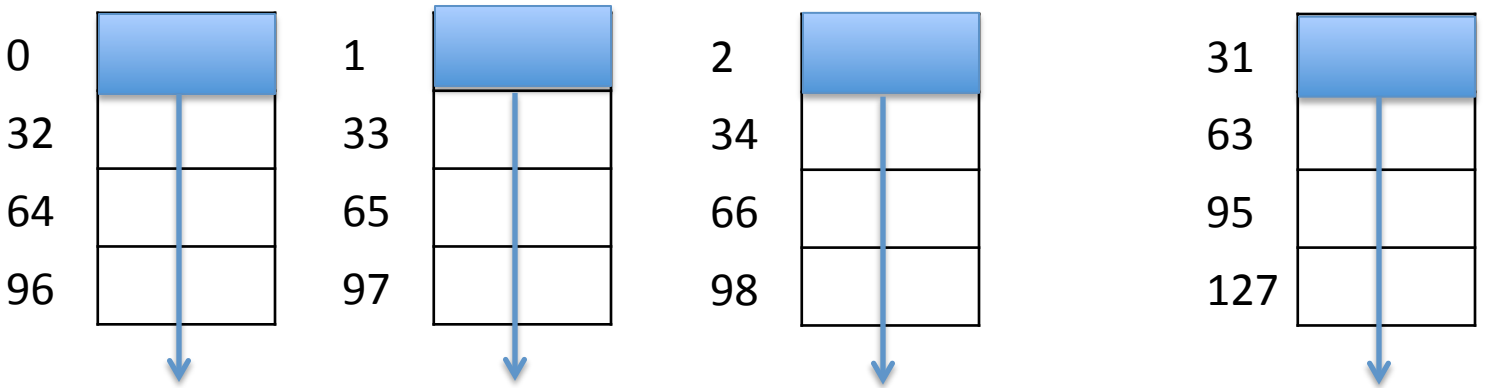
Example 2: Read 0, Read 3
(can proceed in parallel)



Example 3: Read 0, Read 2
(bank conflict)

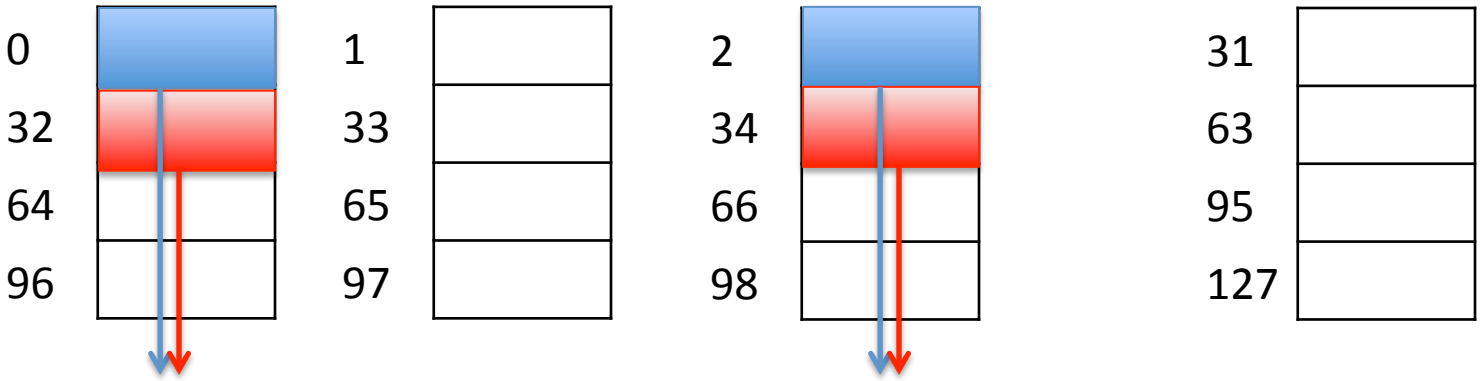
Shared Memory Bank Conflicts

```
__shared__ int A[BSIZE];
…
A[threadIdx.x] = … // no conflicts
```



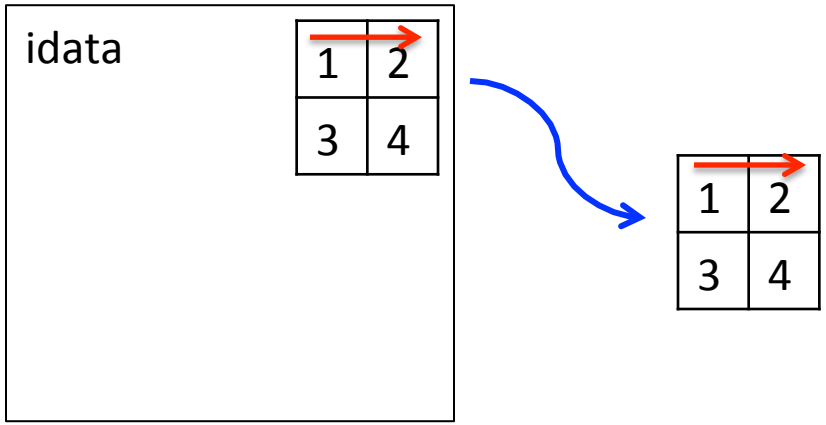
Shared Memory Bank Conflicts

```
__shared__ int A[BSIZE];
…
A[2*threadIdx.x] = // 2-way conflict
```

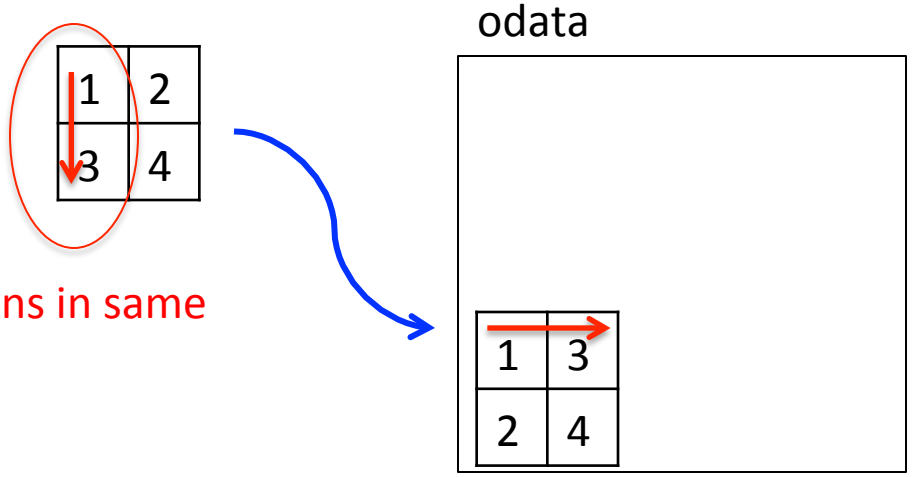


Optimizing Transpose for Coalescing

Step 1: Read block of data into shared memory



Step 2: Copy from shared memory into global memory using coalesce write



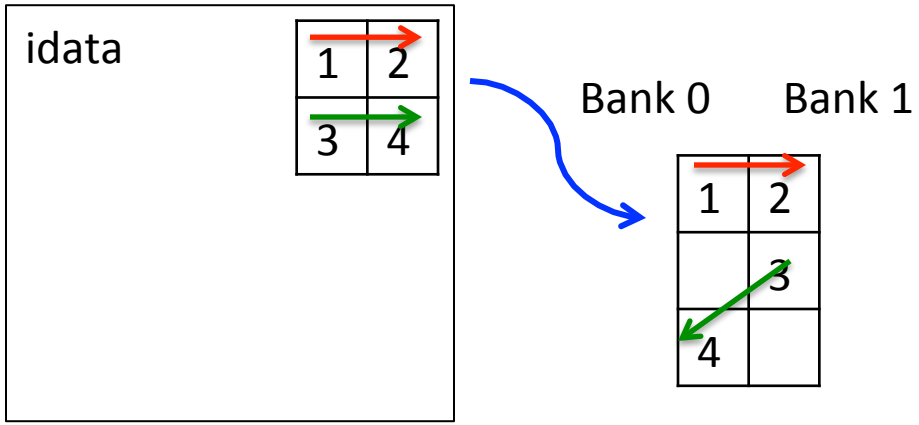
Problem: Access two locations in same shared memory bank.

+ Eliminate Bank Conflicts

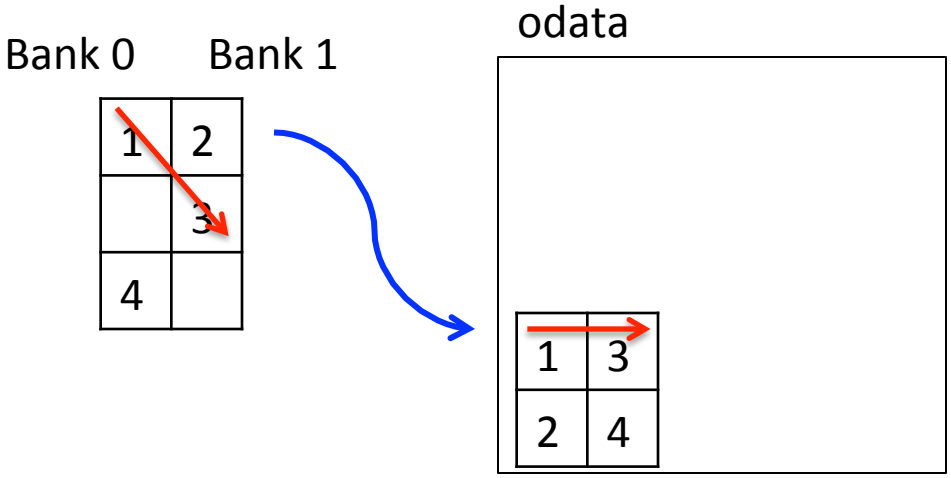
```
__global__ void transposeNoBankConflicts (float *odata, float *idata, int width, int height)
{
  __shared__ float tile[TILE_DIM][TILE_DIM+1];

  int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
  int index_in = xIndex + (yIndex)*width;

  xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
  yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
  int index_out = xIndex + (yIndex)*height;

  for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
    tile[threadIdx.y+i][threadIdx.x] = idata[index_in+i*width];
  }

  __syncthreads();

  for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
    odata[index_out+i*height] = tile[threadIdx.x][threadIdx.y+i];
  }
}
```

Optimizing Transpose for Coalescing

Step 1: Read block of data into shared memory



Step 2: Copy from shared memory into global memory using coalesce write

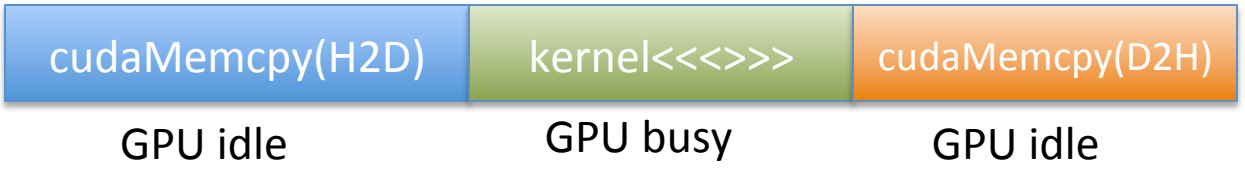


CUDA Streams

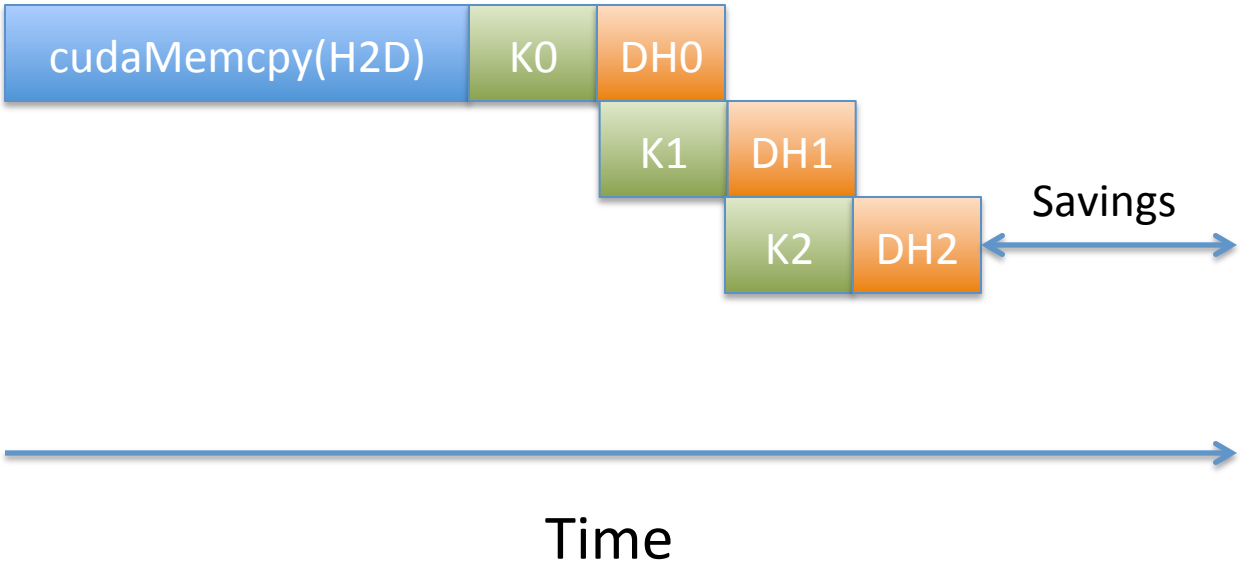
- CUDA (and OpenCL) provide the capability to overlap computation on GPU with memory transfers using “Streams” (Command Queues)
- A Stream orders a sequence of kernels and memory copy “operations”.
- Operations in one stream can overlap with operations in a different stream.

How Can Streams Help?

Serial:



Streams:



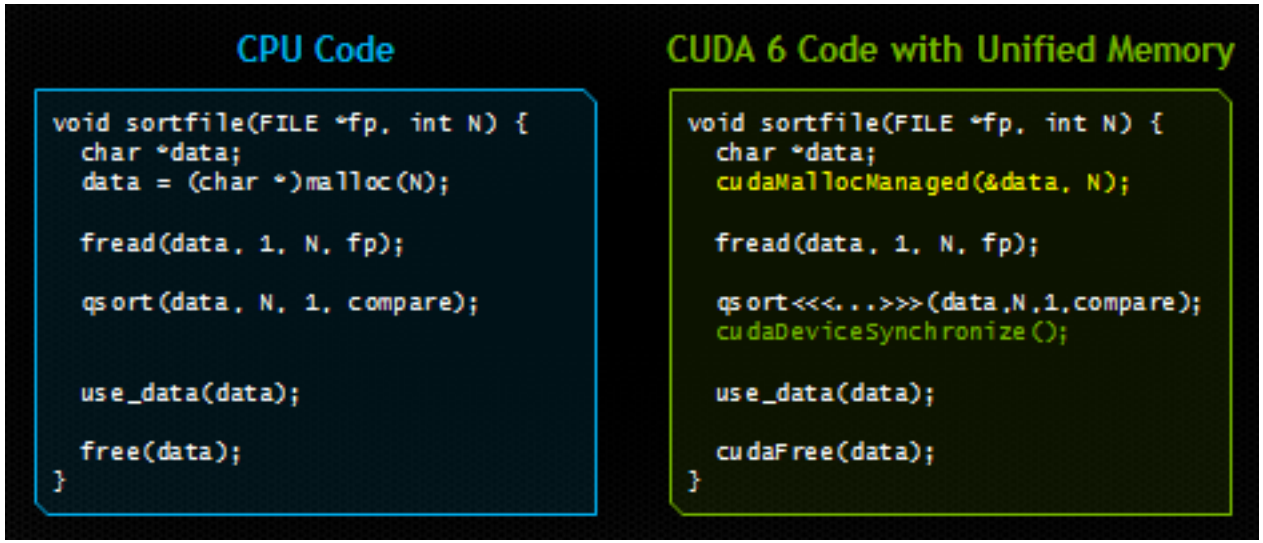
http://on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf

CUDA Streams

```
cudaStream_t streams[3];
for(i=0; i<3; i++)
  cudaStreamCreate(&streams[i]);  // initialize streams

for(i=0; i<3; i++) {
  cudaMemcpyAsync(pD+i*size,pH+i*size,size,
    cudaMemcpyHostToDevice,stream[i]);              // H2D
  MyKernel<<<grid,block,0,stream[i]>>>(pD+i,size); // compute
  cudaMemcpyAsync(pD+i*size,pH+i*size,size,
    cudaMemcpyDeviceToHost,stream[i]);              // D2H
}
```

Recent Features in CUDA

- Dynamic Parallelism (CUDA 5): Launch kernels from within a kernel. Reduce work for e.g., adaptive mesh refinement.
- Unified Memory (CUDA 6): Avoid need for explicit memory copies between CPU and GPU



http://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/

See also, Gelado, et al. ASPLOS 2010.