Lecture 3

SIMD and Vectorization
GPU Architecture

Today's lecture

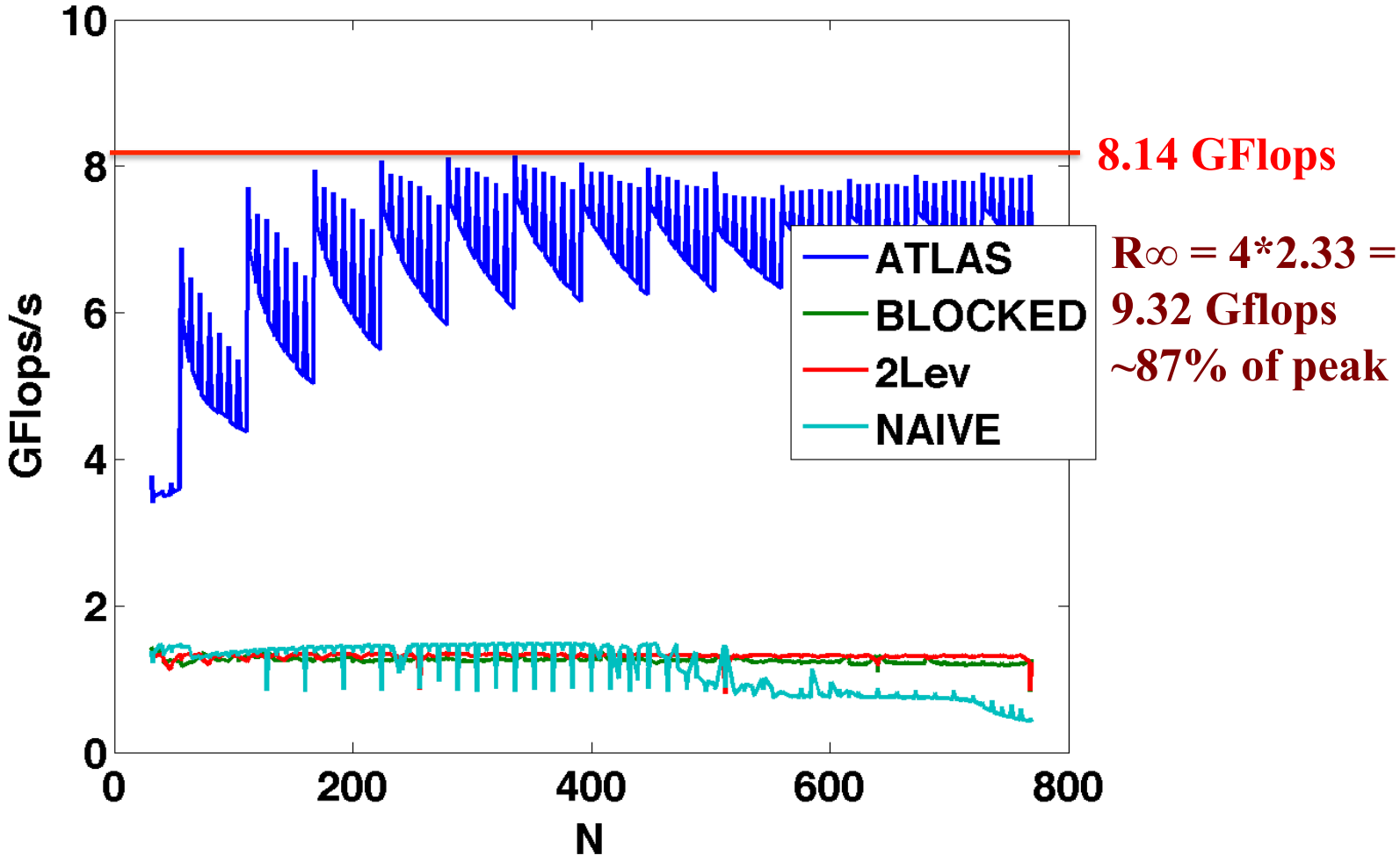
- Vectorization and SSE
- Computing with Graphical Processing Units (GPUs)

Performance programming for Mtx Multiply

- Hierarchical blocking
  - Multiple levels of cache and/or TLB
  - Cache friendly layouts
  - Register blocking (with unrolling)
- SSE intrinsics
- Autotuning
  - Computer generated variants & blocking factors
  - PHiPAC → ATLAS, in Matlab
  - Performance models not sufficiently accurate
  - Need to tune to matrix size
- See Jim Demmel's lecture
  www.cs.berkeley.edu/~demmel/cs267_Spr12/Lectures/lecture02_memhier_jwd12.ppt

Matrix Multiply optimizations

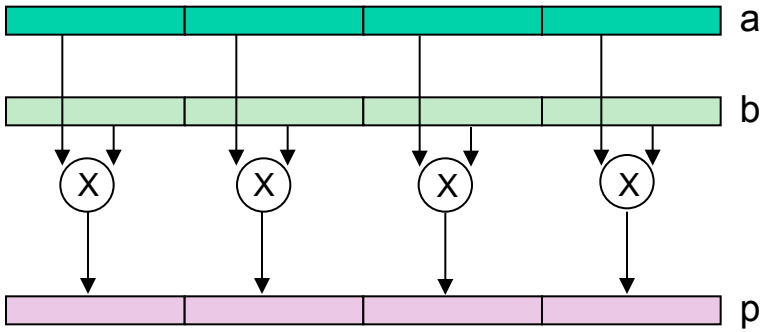
- Blocking for cache will boost performance but a lot more is needed to approach ATLAS' performance



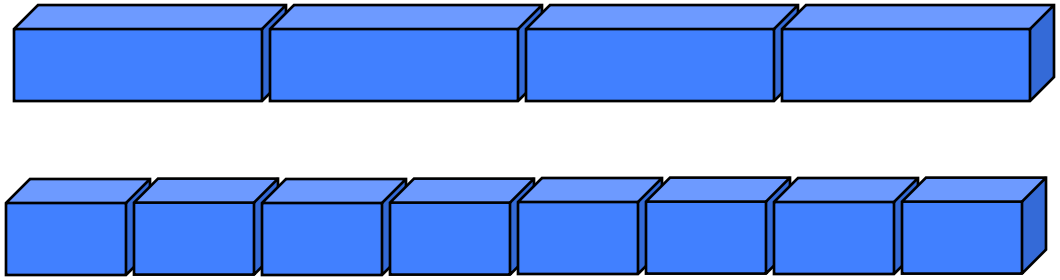
Streaming SIMD Extensions

- SIMD instruction set on short vectors
- SSE (AVX on Stampede, SSE4.1/4.2 on CSEClass, SSE3 on Bang)
- On Stampede: 16x256 bit vector registers

```
for i = 0:N-1 { p[i] = a[i] * b[i];}
```



1		1		1
4	=	2	*	2
2		2		1
6		3		2



4 doubles

8 floats

How do we use SSE & how does it perform?

- Low level: assembly language or libraries
- Higher level: a vectorizing compiler

```
g++ --std=c++11    -O3   -ftree-vectorizer-verbose=2
float b[N], c[N];
for (int i=0; i<N; i++)
      b[i] += b[i]*b[i] +  c[i]*c[i];
```

```
7: LOOP VECTORIZED.
vec.cpp:6: note: vectorized 1 loops in function..
```

- Performance

Single precision: With vectorization : 1.9 sec.
Without vectorization : 3.2 sec.

Double precision: With vectorization: 3.6 sec.
Without vectorization : 3.3 sec.

http://gcc.gnu.org/projects/tree-ssa/vectorization.html

How does the vectorizer work?

- Original code

```
float b[N], c[N];
for (int i=0; i<N; i++)
      b[i] += b[i]*b[i] +  c[i]*c[i];
```

- Transformed code

```
for (i = 0; i < 1024; i+=4)
      a[i:i+3] = b[i:i+3] + c[i:i+3];
```

- Vector instructions

```
for (i = 0; i < 1024; i+=4){
     vB = vec_ld( &b[i] );
    vC = vec_ld( &c[i] );
     vA = vec_add( vB, vC );
     vec_st( vA, &a[i] );
}
```

What prevents vectorization

- Interrupted flow out of the loop

```
for (i=0; i<n; i++) {
    a[i] = b[i] + c[i];
    maxval = (a[i] > maxval ? a[i] : maxval);
    if (maxval > 1000.0) break;
}
```

Loop not vectorized/parallelized: multiple exits

- This loop will vectorize

```
for (i=0; i<n; i++) {
    a[i] = b[i] + c[i];
    maxval = (a[i] > maxval ? a[i] : maxval);
}
```

C++ intrinsics

- The compiler may not be able to handle all situations, such as conditionals
- Library intrinsics map directly onto machine instructions (one or more)
- Supported by gcc and other compilers
- The interface provides 128 bit data types and operations on those datatypes
- Data may need to be aligned

SSE Pragmatics

- AVX: 16 YMM data registers (256 bit)
  (Don’t use the MMX 64 bit registers)
- SSE4: 8 XMM registers (128 bits)
- Vector operations (add, subtract, etc)
- Data transfer (load/store)
- Shuffling (handles conditionals)
- See the Intel intrisics guide:
  software.intel.com/sites/landingpage/IntrinsicsGuide
- May need to invoke compiler options
  depending on level of optimization

Blocking for registers in matrix multiply

- We can apply blocking to the registers, too
- In SSE4: 2x2 matrix multiply
- Store array values on the stack

C_{00} += $A_{00}B_{00}$ + $A_{01}B_{10}$
C_{10} += $A_{10}B_{00}$ + $A_{11}B_{10}$
C_{01} += $A_{00}B_{01}$ + $A_{01}B_{11}$
C_{11} += $A_{10}B_{01}$ + $A_{11}B_{11}$

$$\begin{pmatrix} A00 & A01 \\ A10 & A11 \end{pmatrix}\begin{pmatrix} B00 & B01 \\ B10 & B11 \end{pmatrix}$$

Rewrite as SIMD algebra

C00_C01 += A00_A00 * B00_B01
C10_C11 += A10_A10 * B00_B01
C00_C01 += A01_A01 * B10_B11
C10_C11 += A11_A11 * B10_B11

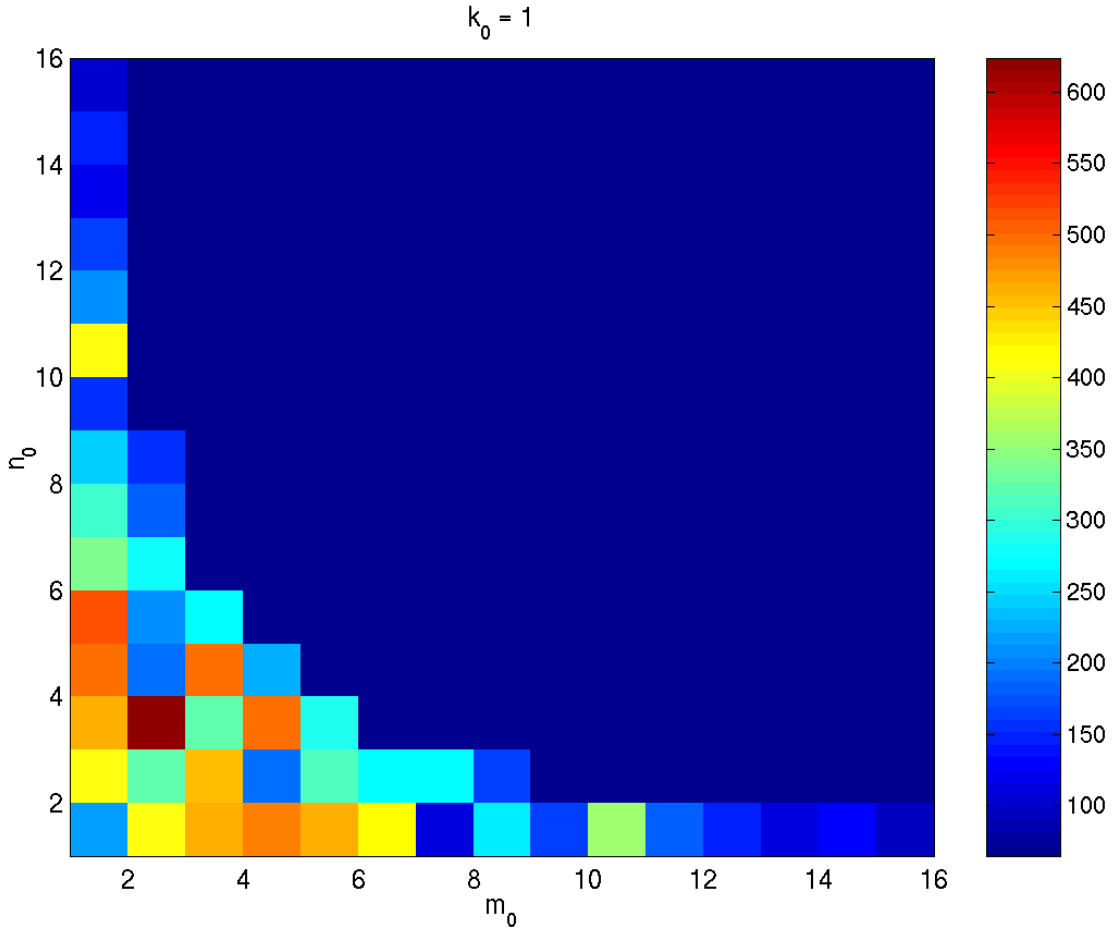
2x2 Matmul with SSE instrinsics

```
#include <emmintrin.h>
void square_dgemm (int N, double* A, double* B, double* C){
__m128d c1 = _mm_loadu_pd( C+0*N);      //load unaligned block in C
__m128d c2 = _mm_loadu_pd( C+1*N);
for( int i = 0; i < 2; i++ ){
    __m128d a1 = _mm_load1_pd( A+i+0*N);  //load i-th column of A (A0x,A0x)
    __m128d a2 = _mm_load1_pd( A+i+1*N);                    (A1x,A1x)
    __m128d   b = _mm_load_pd( B+i*N);   //load aligned i-th  row of B
    c1 =_mm_add_pd( c1, _mm_mul_pd( a1, b ) );   //rank-1 update
    c2 =_mm_add_pd( c2, _mm_mul_pd( a2, b ) );
}
_mm_storeu_pd( C+0*N, c1 );                  //store unaligned block in C
_mm_storeu_pd( C+1*N, c2 );
```

```
C00_C01 += A00_A00 * B00_B01
C10_C11 += A10_A10 * B00_B01
C00_C01 += A01_A01 * B10_B11
C10_C11 += A11_A11 *  B10_B11
```

$$\begin{pmatrix} A00 & A01 \\ A10 & A11 \end{pmatrix}\begin{pmatrix} B00 & B01 \\ B10 & B11 \end{pmatrix}$$

A search space



A 2-D slice of a 3-D register-tile search space. The dark blue region was pruned. (Platform: Sun Ultra-IIi, 333 MHz, 667 Mflop/s peak, Sun cc v5.0 compiler)

Jim Demmel

Loop Unrolling

- Common loop optimization strategy
- Duplicate the body of the loop

```
for (int i=0; i < n ; i++)
    z[i] = x[i] + y[i];
```

```
for (int i=0; i < n ; i++4){
    z[i+0] = x[i+0] + y[i+0];
    z[i+1] = x[i+1] + y[i+1];
    z[i+2] = x[i+2] + y[i+2];
    z[i+3] = x[i+3] + y[i+3];
   }
```

- Register utilization, instruction scheduling
- May be combined with “jamming:” unroll and jam
- Not always advantageous

Today's lecture

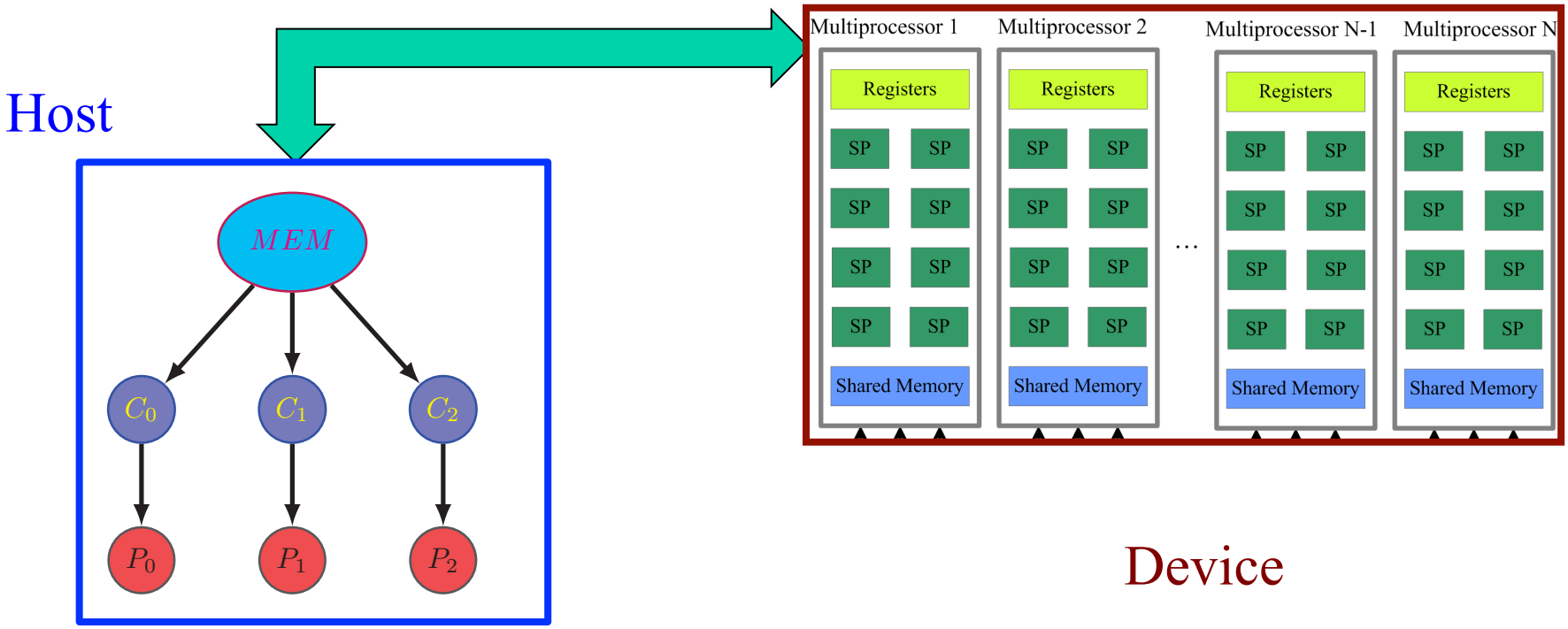
- Vectorization and SSE
- Computing with Graphical Processing Units (GPUs)

Recall processor design trends

- No longer possible to use growing population of transistors to boost single processor performance
  - Can no longer increase the clock speed
  - Instead, we replicate the cores
- An opportunity: Specialize the processing core
  - Simplified design, pack more onto the chip
  - Boost performance
  - Reduce power
- Simplified core
  - Remove architectural enhancements like branch caches
  - Constrain memory access and control flow
  - Partially expose the memory hierarchy
- Embrace technological trends

Heterogeneous processing with Graphical Processing Units

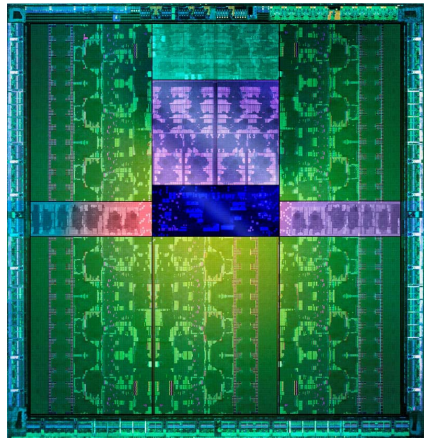
- Specialized *many-core* processor
- Explicit data motion
  - between *host* and *device*
  - inside the device



Stampede's NVIDIA Tesla Kepler K20 (GK110)

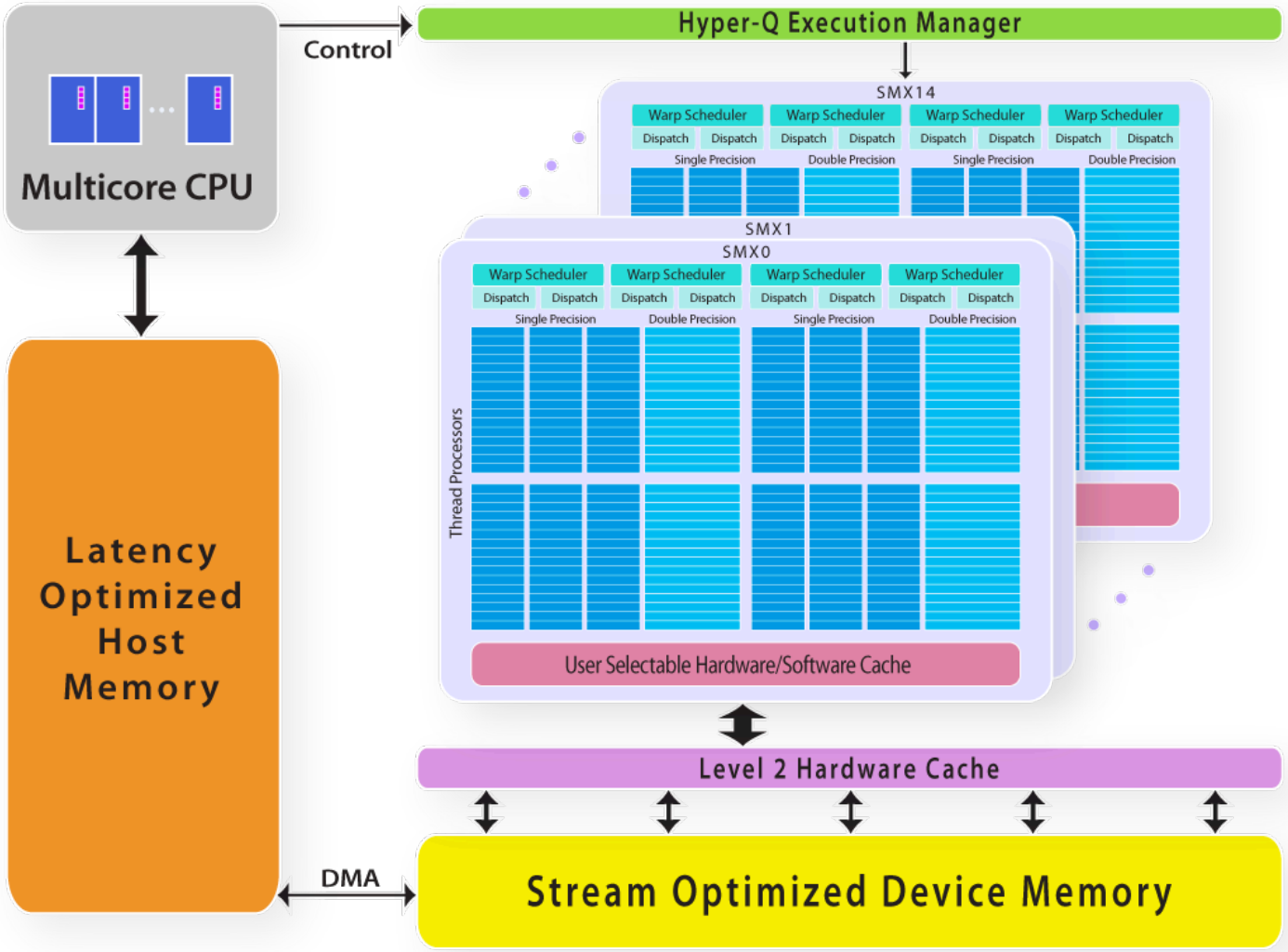
- Hierarchically organized clusters of streaming multiprocessors
  - 13 streaming processors @ 706 MHz
    (down from 1.296 GHz on GeForce 280)
  - Peak performance : 1.17 Tflops/s Double Precision
- SIMT parallelism: long vectors
- 5 GB "device" memory (frame buffer) @ 208 GB/s
- See http://international.download.nvidia.com/pdf/kepler/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf

Nvidia



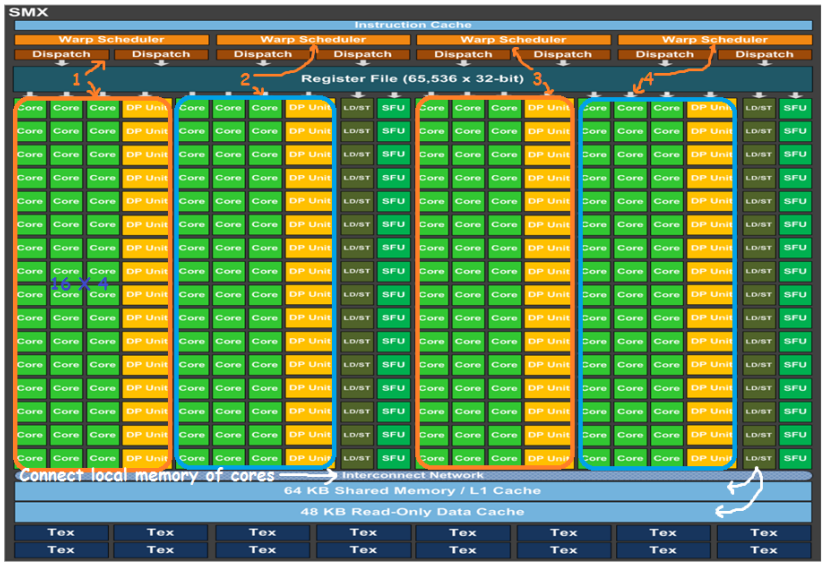
7.1B transistors

Overview of Kepler GK110



SMX Streaming processor

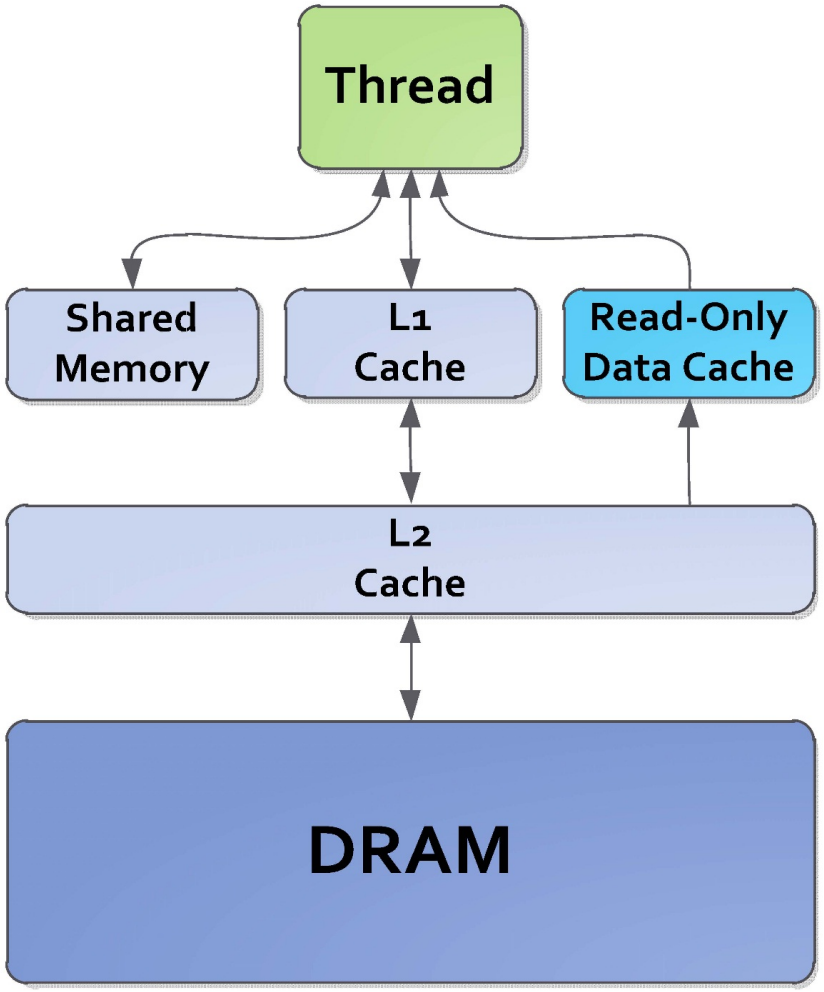
- Processor organized into SMX streaming processors, AKA *vector units*
- Stampede's K20s (GK110 GPU) have 13 SMXs (2496 cores)
- Each vector unit
  - 192 SP cores, 64 DP cores, 32 SFUs, 32 Load/Store units
  - each scalar cores: fused multiply adder, truncates intermediate result
  - 64KB on chip memory configurable as Shared memory + L1 Cache
  - 64K x 32-bit registers (256KB) up to 255/thread
  - 1 FMA /cycle = 2 flops / cyc / DP core * 64 DP/SMX * 13 SMX = 1664 flops/cyc @0.7006 Ghz = 1.165 TFLOPS



Nvidia

Kepler's Memory Hierarchy

- DRAM takes hundreds of cycles to access
- Can partition the on-chip Shared memory L,1$ cache
  {¾ + ¼}
  {¾ + ¼}
  {½ + ½}
- L2 Cache (768 KB)



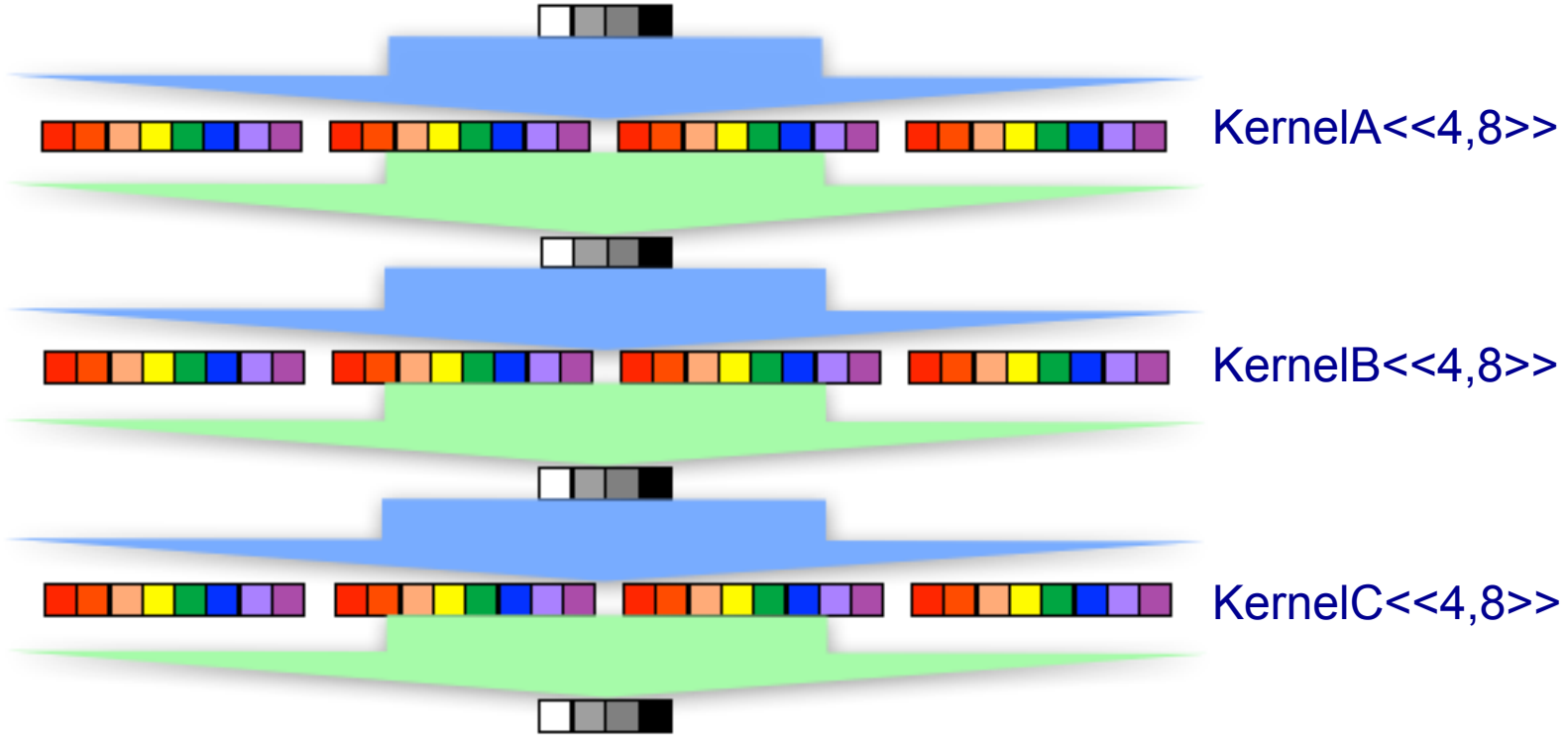
B. Wilkinson

Additional features

- Direct data exchange between threads in the same warp
- High speed atomics suitable for inner loops (e.g. summation)
- Dynamic parallelism: launch new grids from GPU
- GPUDirect – RDMA (direct) access to device memory from other devices, including NICS
- HyperQ: multiple host threads can launch work on the device simultaneously
- Quad warp scheduler, 128 threads can be issued an executed simultaneously
- L2 Cache (768 KB)
- See http://www.anandtech.com/show/6446/nvidia-launches-tesla-k20-k20x-gk110-arrives-at-last/3

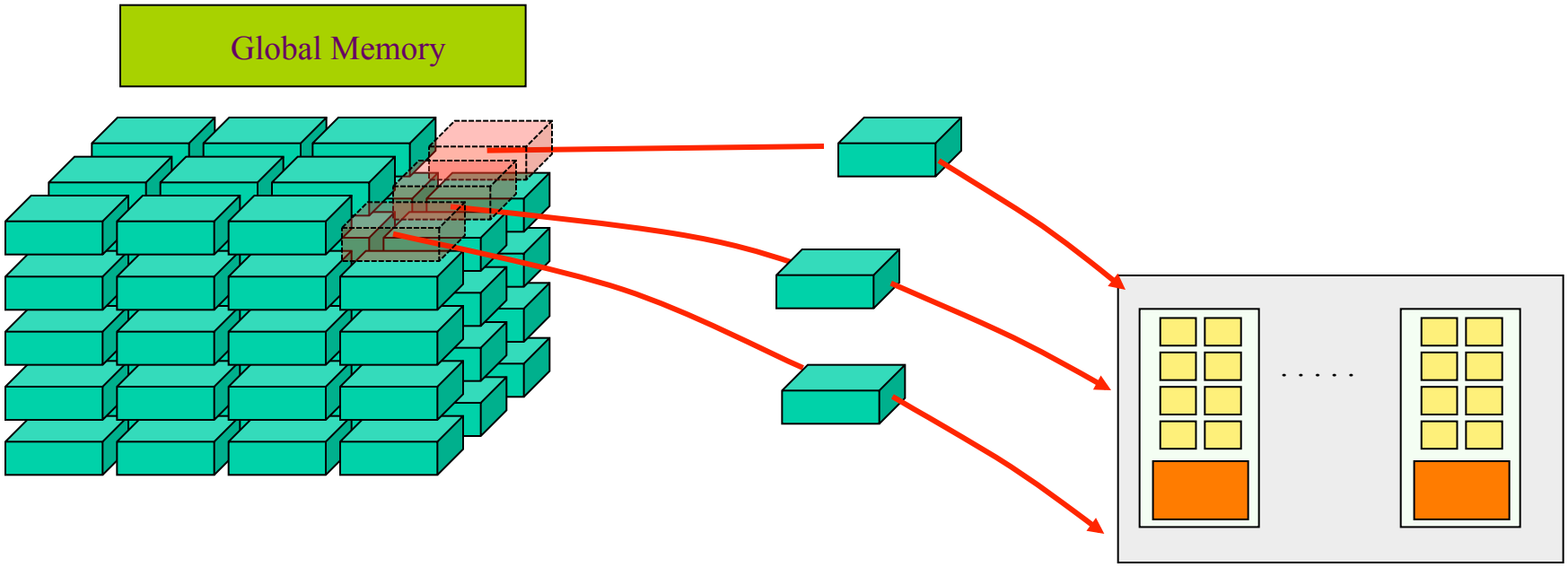
CUDA

- Programming environment with extensions to C
- Under control of the *host*, invoke sequences of multithreaded kernels on the *device* (GPU)
- Many lightweight threads
- CUDA: programming environment + C extensions



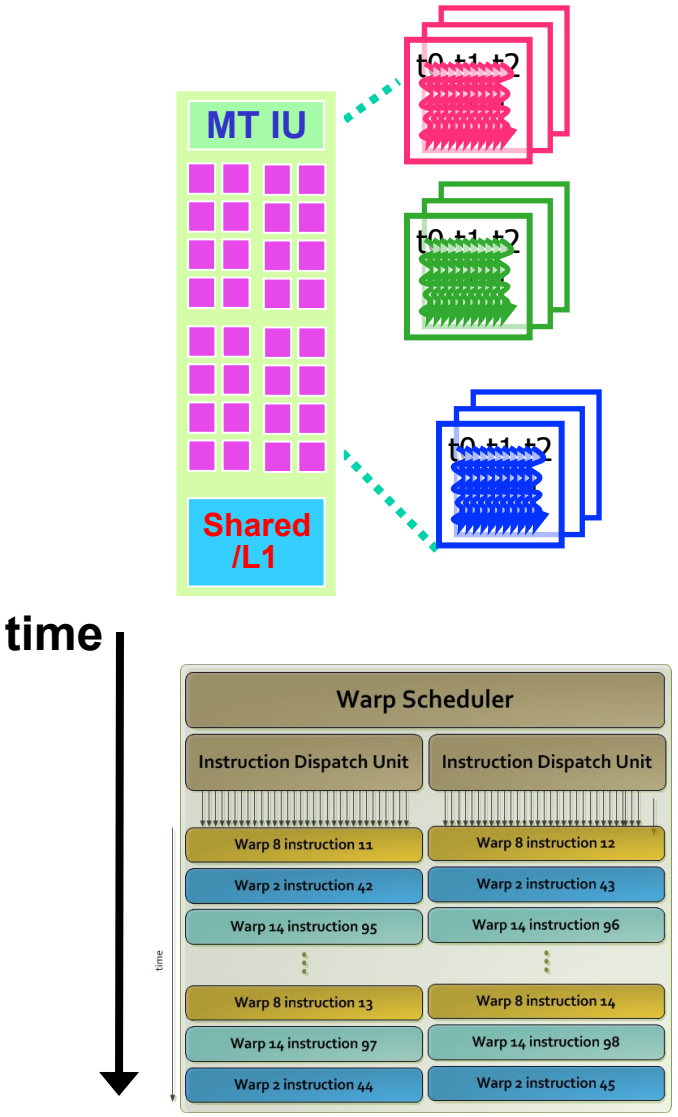
Thread execution model

- Kernel call spawns virtualized, hierarchically organized threads Grid ⊃ Block ⊃ Thread
- Hardware handles dispatching, 0 overhead
- Compiler re-arranges loads to hide latencies
- Global synchronization: kernel invocation



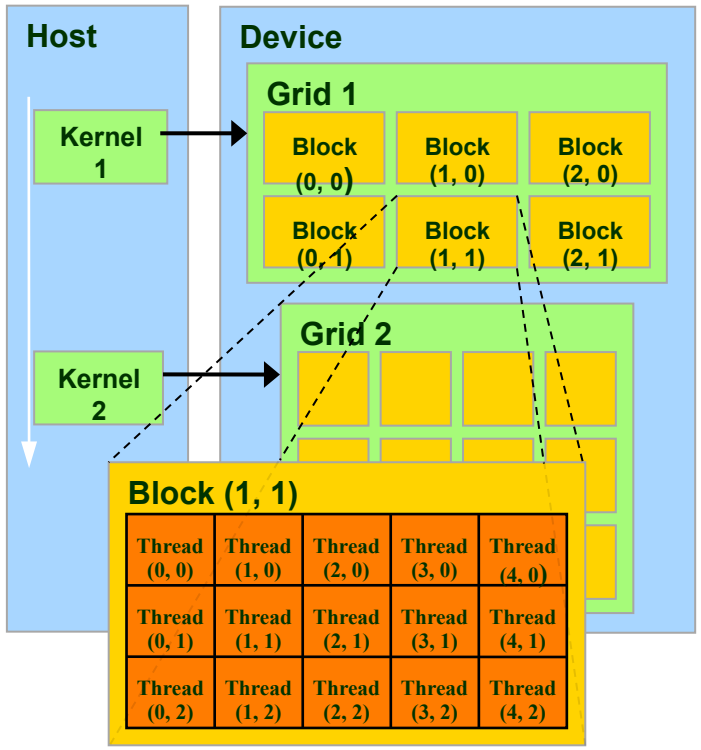
Warp Scheduling

- Threads assigned to an SMX in units of a thread block, multiple blocks
- Each block divided into *warps* of 32 (SIMD) threads, a schedulable unit
  - A warp becomes eligible for execution when all its operands are available
  - Dynamic instruction reordering: eligible warps selected for execution using a prioritized scheduling policy
  - All threads in a Warp execute the same instruction, branches serialize execution
- Multiple warps simultaneously active, hiding data transfer delays
- All registers in all the warps are available, 0 overhead scheduling
- Hardware is free to assign blocks to any SMX
- There are 4 warp schedulers/SMX



Hierarchical Thread Organization

- Thread organization
  - Grid ⊃ Block ⊃ Thread
  - Specify number and geometry of threads in a block and similarly for blocks
- Each thread uniquely specified by block & thread ID
- Programmer determines the mapping of virtual thread IDs to global memory locations
  - $\prod: Z^n \rightarrow Z^2 \times Z^3$
  - $\Theta(\Pi_\iota), \forall \Pi_\iota \in \Pi$

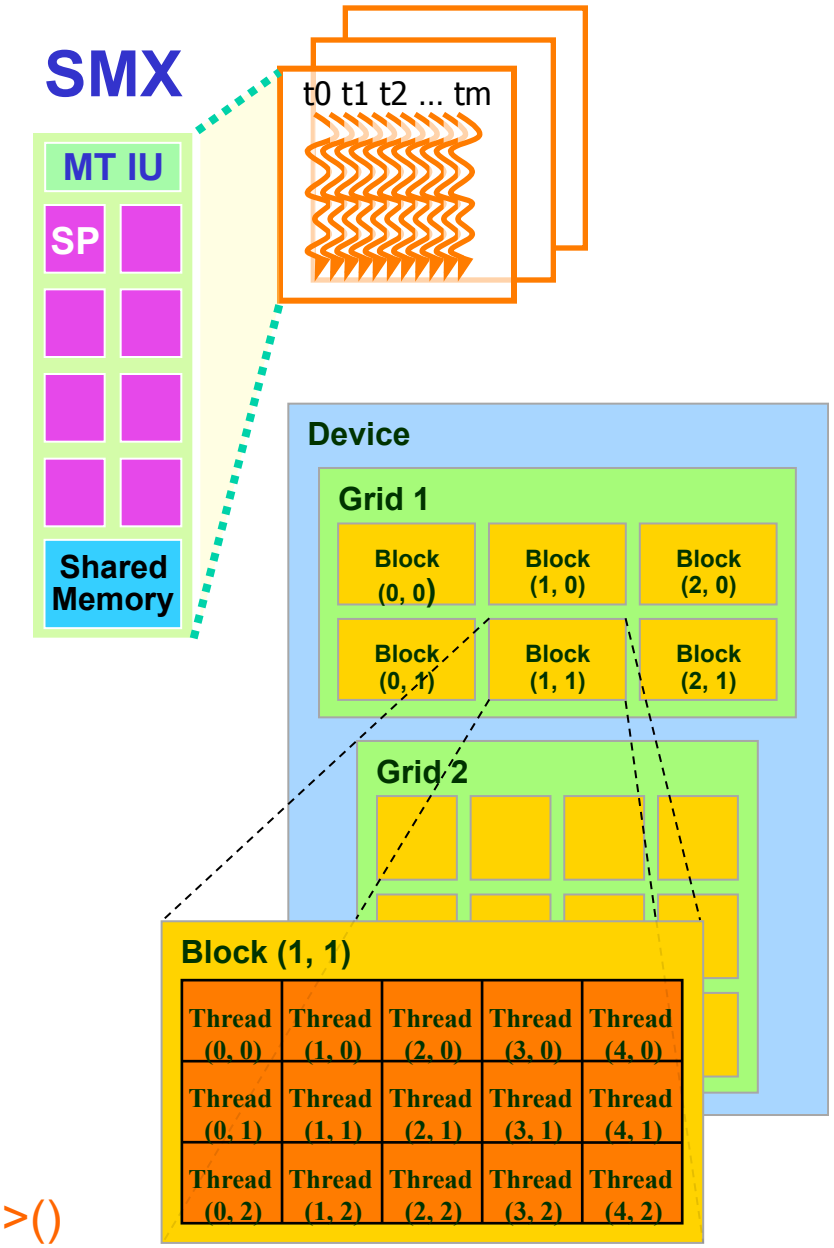


KernelA<<<2,3>,<3,5>>>()
Grid Block

DavidKirk/NVIDIA & Wen-mei Hwu/UIUC

Thread execution

- Thread Blocks
  - Unit of workload assignment
  - Each thread has its own set of registers
  - All have access to a fast on-chip *shared memory*
  - Synchronization only among all threads in a block
  - Threads in different blocks communicate via slow global memory
  - Processor groups threads into *warps* of 32 threads
- SIMT parallelism: all threads in a warp execute the same instruction
  - All branches followed
  - Instructions disabled
  - Divergence, serialization



KernelA<<<2,3>,<3,5>>>()

Grid Block

DavidKirk/NVIDIA & Wen-mei Hwu/UIUC

Coding example – Increment Array

Serial Code

```
void incrementArrayOnHost(float *a, int N){
  int i;
  for (i=0; i < N; i++) a[i] = a[i]+1.f;
}
```

```
#include <cuda.h>
__global__ void incrementOnDevice(float *a, int N)
{
  int idx = blockIdx.x*blockDim.x + threadIdx.x;
  if (idx<N) a[idx] = a[idx]+1.f;
}

incrementOnDevice <<< nBlocks, blockSize >>> (a_d, N);
```

Rob Farber, Dr Dobb's Journal

Managing memory

```
float *a_h, *b_h;          // pointers to host memory
float *a_d;                // pointer to device memory

cudaMalloc((void **) &a_d, size);

for (i=0; i<N; i++) a_h[i] = (float)i;  // init host data

cudaMemcpy(a_d, a_h, sizeof(float)*N,
                  cudaMemcpyHostToDevice);
```

Computing and returning result

```
int bSize = 4;
int nBlocks = N/bSize + (N%bSize == 0?0:1);
incrementOnDevice <<< nBlocks, bSize >>> (a_d, N);

// Retrieve result from device and store in b_h
  cudaMemcpy(b_h, a_d, sizeof(float)*N,
                       cudaMemcpyDeviceToHost);

// check results
  for (i=0; i<N; i++) assert(a_h[i] == b_h[i]);

// cleanup
  free(a_h); free(b_h);
  cudaFree(a_d);
```

Experiments - increment benchmark

- Total time: timing taken from the host, includes copying data to the device
- Device only: time taken on device only

N = 8388480, block size = 128, times in *milliseconds*, cseclass02

Reps = 10	100	1000	10^4	10^5	
3.3	36	358	3.58s	35.8s	Device time
71	102	429	3.64s	35.9s	Kernel launch + data xfer
92	730	7.06s	--	--	Host
6.8	52	500			a[i] = 1 + sin(a[i]) : Device)
6.4s	23.9s	200s			Sine function (Host)

Measuring performance

- Two ways
  - Use an ordinary timer, e.g. gettimeofday()
  - Use Cuda events/elapsed time (#ifdef CUDA_TIMER)
- See incrArray
- Note that kernel invocation is asynchronous

```
cudaThreadSynchronize();
double t_device_compute = -getTime();
        incr<<< nBlocks, bSize >>> (a_d, N);
cudaThreadSynchronize();
 t_device_compute +=getTime();
```

CUDA Error Handling

Cuda error: Can't run kernel: invalid device function.

- Cuda can silently fail, you can observe misleading performance
- E.g. if you specify an invalid grid / thread block dimensions
- Note: the last error can be cleared by successive kernel calls, so check frequently

```
cudaMalloc((void **) &a_d, size);
checkCUDAError("Unable to allocate storage on the device");
```

- Consult checkCUDAError() in utils.cu (incrArr)
- What about asynchronous calls?
- cf CUDA Programming Guide, “Error Handling”

Getting information about the binary

- Compiler will report a kernel's register usage along with that of local, shared and constant memory

--ptxas-options=-v

```
incrementArrays (float *a, int N)
int idx = blockIdx.x*blockDim.x + threadIdx.x;
if (idx<N) a[idx] = a[idx]+1.f;
```

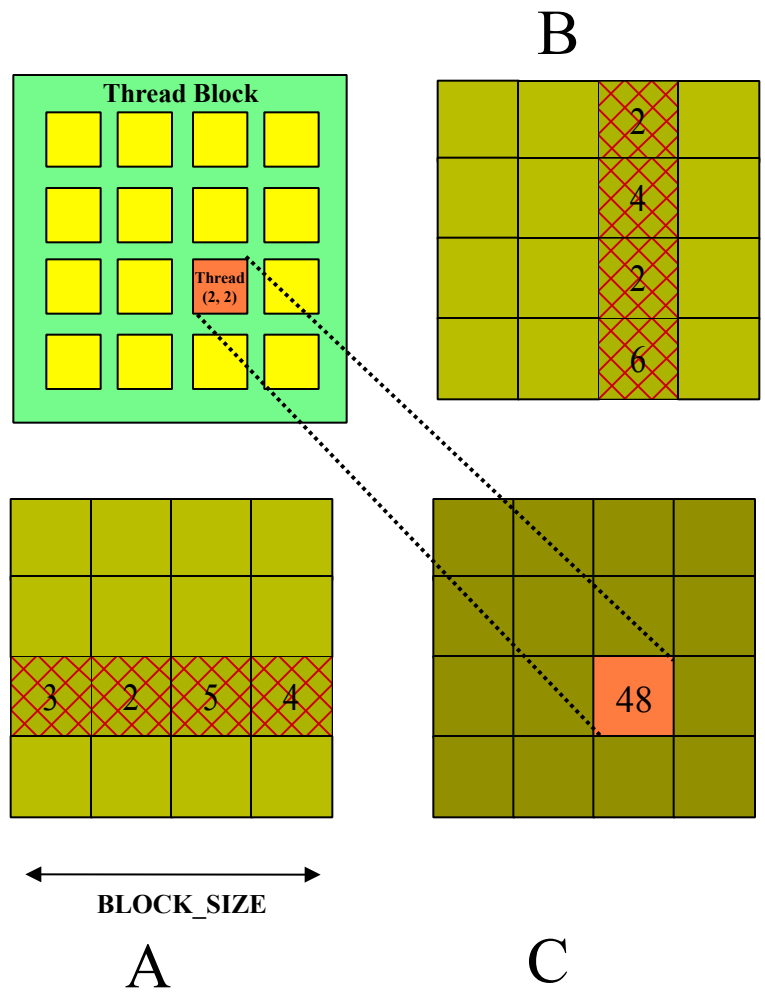
```
ptxas info : Compiling entry function
'_Z22incrementArrayOnDevicePfii' for 'sm_21'
ptxas info    : Used 6 registers, 48 bytes cmem[0]
```

Today's lecture

- CUDA Programming
- Matrix Multiplication on the GPU

Naïve kernel implementation

- Each thread computes one element of C
  - Loads a row of matrix A
  - Loads a column of matrix B
  - Computes a dot product
- Every value of A and B is loaded N times from global memory



Courtesy DavidKirk/NVIDIA and Wen-mei Hwu/UIUC

Naïve Kernel

```
__global__ void
matMul(DOUBLE* C, DOUBLE* A, DOUBLE* B) {
  int I  =  blockIdx.x*blockDim.x + threadIdx.x;
  int J =  blockIdx.y*blockDim.y + threadIdx.y;
  int N =  blockDim.y*gridDim.y; // Assume a square matrix
  if ((I < N) && (J < N)){
        float _c = 0;
        for (unsigned int k = 0; k < N; k++) {
            float a = A[I * N + k];
            float b = B[k * N + J];
            _c += a * b;
        }
        C[I * N + J] = _c;
   }
}
```

```
for (unsigned int i = 0; i < N; i++)
        for (unsigned int j = 0; j < N; j++) {
            DOUBLE sum = 0;
            for (unsigned int k = 0; k < N; k++)
                sum += A[i * N + k] * B[k * N + j];
            C[i * N + j] = (DOUBLE) sum;
        }
```

CUDA code on the host side

```
unsigned int n2 = N*N*sizeof(DOUBLE);
DOUBLE *h_A = (DOUBLE*) malloc(n2);
DOUBLE *h_B = (DOUBLE*) malloc(n2);
// Check that allocations went OK
 assert(h_A);  assert(h_B);

genMatrix(h_A, N, N);  genMatrix(h_B, N, N);  // Initialize matrices

DOUBLE *d_A, *d_B, *d_C;
cudaMalloc((void**) &d_A, n2); ... &d_A  ...  &d_B
checkCUDAError("Error allocating device memory arrays");

 // copy host memory to device
 cudaMemcpy(d_A, h_A, n2, cudaMemcpyHostToDevice);
 checkCUDAError("Error copying data to device");
 cudaMemcpy(d_B, h_B, n2, cudaMemcpyHostToDevice);
 checkCUDAError("Error copying data to device");
```

Host code - continued

```
// setup execution configurations
    dim3 threads(ntx, nty,1);        // ntx & nty are user input
    dim3 grid(N / threads.x, N / threads.y);

     // launch the kernel
     matMul<<< grid, threads >>>(d_C, d_A, d_B);

     // retrieve result
     cudaMemcpy(h_C, d_C, n2, cudaMemcpyDeviceToHost);
     checkCUDAError("Unable to retrieve result from device");

// Free device storage
     assert(cudaSuccess ==cudaFree(d_A));
     assert(cudaSuccess ==cudaFree(d_B));
     assert(cudaSuccess ==cudaFree(d_C));
```

Execution Configurations

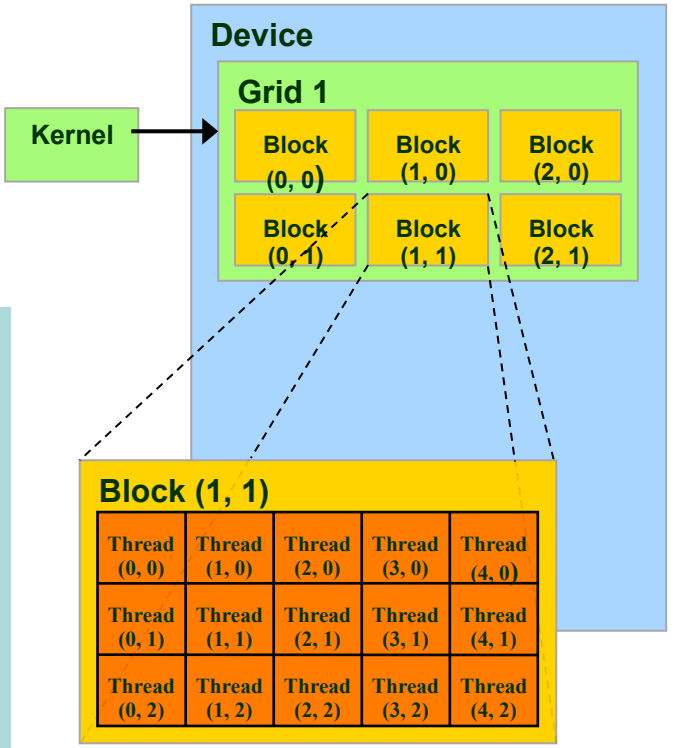
- Grid ⊃ Block ⊃ Thread
- Expressed with *configuration variables*

```
__global__ void Kernel (...);

dim3 DimGrid(2,3);       // 6 thread blocks

dim3 DimBlock(3,5,1);  // 15 threads /block

Kernel<<< DimGrid, DimBlock, >>>(...);
```



DavidKirk/NVIDIA & Wen-mei Hwu/UIUC

Performance

- Baseline [N=512, double precision]
  - Lilliput, C1060, 2.0 GHz Intel Xeon E5504, 4MB L3, peak 8.0 GF / core
  - Forge, M2070 14×32 cores
  - 21 GF on 4 CPU cores (MPI), 25 Gflops for N=2K

Gflops dp, C1060	**9.8**	**8.5**	**7.4**	**5.9**	**5.3**	**5.1**	**3.0**	**2.7**
Geometry	2×256	2×128	2×64	4×128	4×64 2×32	4×32	8×64	8×32

Gflops sp, C1060	**8.6**	**7.7**	**6.2**	**4.6**	**3.9**	**3.5**	**2.0**	**1.8**
Geometry	2×256	2×128	2×32 2×64	4×128	4×64	4×32	8×64	8×32

Gflops sp **Dirac** dp	50,49,46,..., 9.5 69,69,68,…, 6.6
Geometry	2×256, 2×128, 2×64,…, 16×16 2×256, 2×128, 2×64,…, 16×16