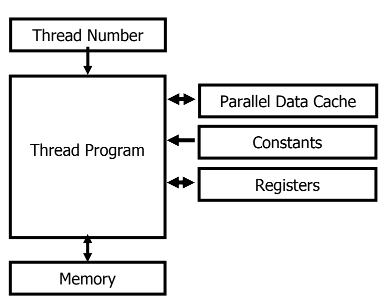
S05: High Performance Computing with CUDA

Programming CUDA

Ian Buck
NVIDIA

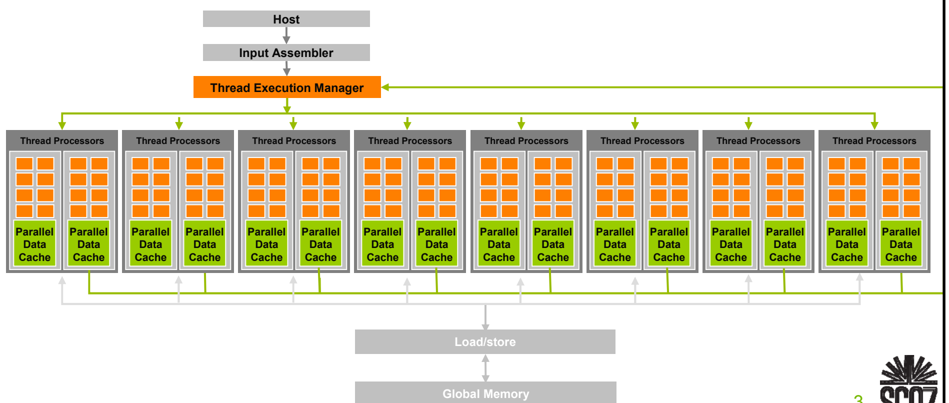
Enabling GPU Computing

- GPU Computing Arch
- CUDA
  - Targeted platform for GPU Computing



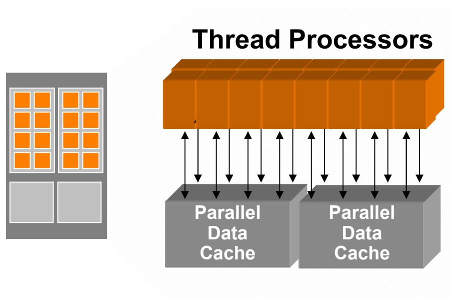
GPU Computing

- Processors execute computing threads
- Thread Execution Manager issues threads
- 128 Thread Processors
- Parallel Data Cache accelerates processing



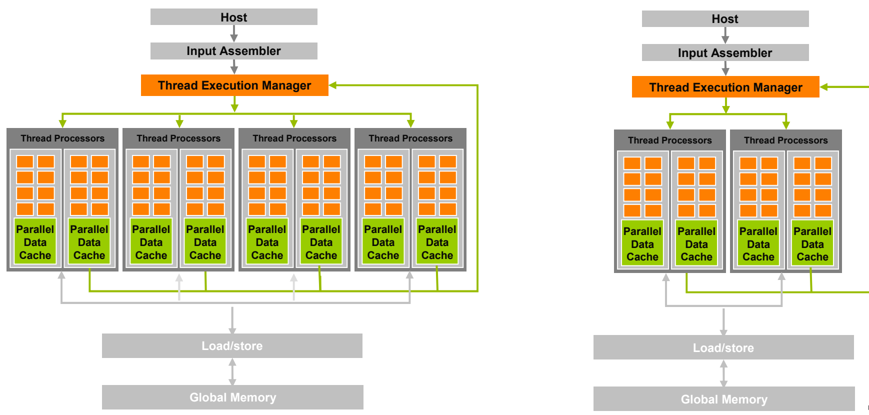
Thread Processor Group

- 128, 1.35 GHz processors
- 16KB Parallel Data Cache per group
- Scalar architecture
- IEEE 754 Precision

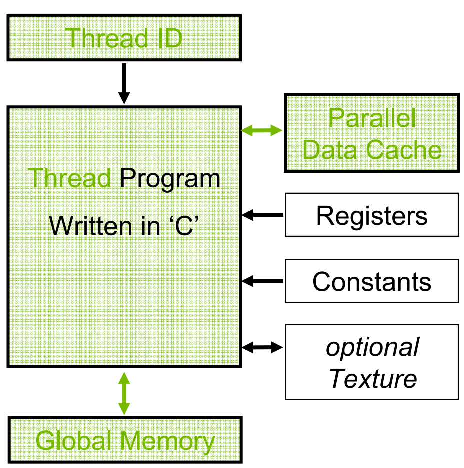


Scaling the Architecture

- Same program
- Scalable performance

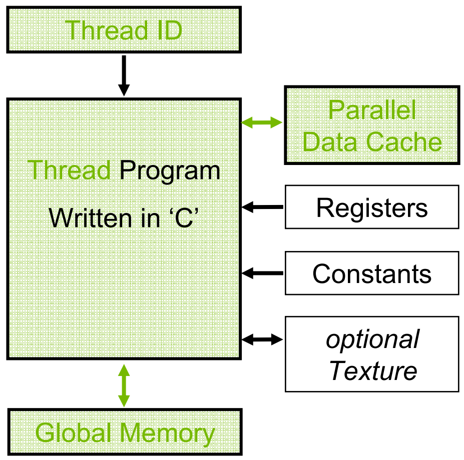


GPU Computing Model



- Dedicated computing mode
- Thread programs use 'C'
- On-chip shared memory
- General load/store

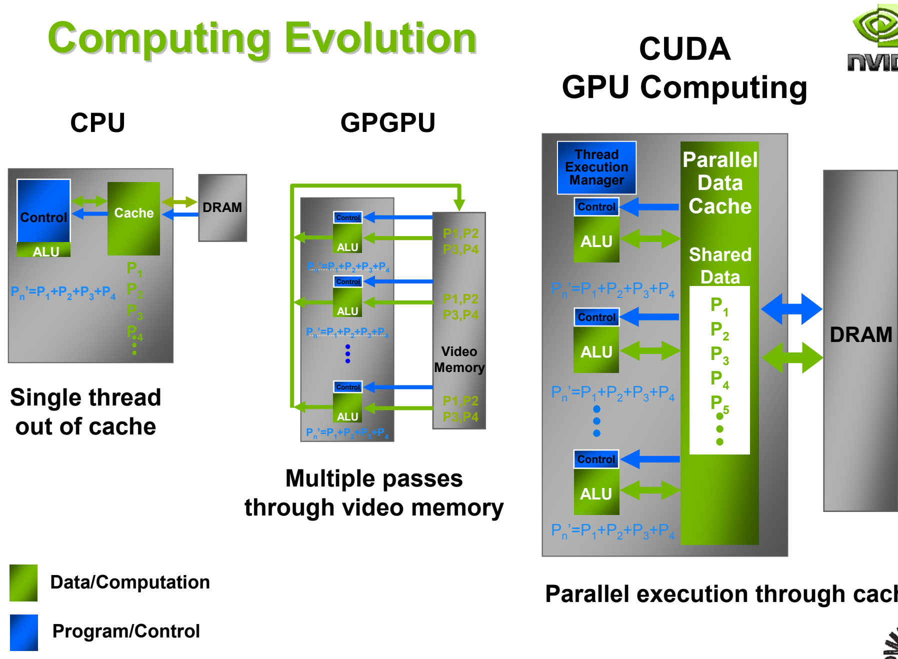
GPU Computing Model



Parallel Data Cache

- Dedicated on-chip memory
- Shared between threads for inter-thread communication
- Explicitly managed – software managed cache
- As Fast Registers

Computing Evolution



Single thread out of cache

Multiple passes through video memory

Parallel execution through cache

Programming Model:
A Massively Multi-threaded Processor

Move data-parallel application portions to the GPU

Differences between GPU and CPU threads

- **Lightweight threads**
- **GPU supports 1000's of threads**

Programming Model:
A Highly Multi-threaded Coprocessor

- **The GPU is viewed as a compute device that:**
  - **Is a coprocessor to the CPU or host**
  - **Has its own DRAM (device memory)**
  - **Runs many threads in parallel**
- **Data-parallel portions of an application execute on the device as kernels which run many cooperative threads in parallel**
- **Differences between GPU and CPU threads**
  - **GPU threads are extremely lightweight**
    - **Very little creation overhead**
  - **GPU needs 1000s of threads for full efficiency**
    - **Multi-core CPU needs only a few**

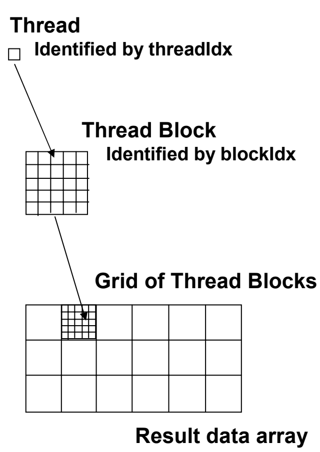
C on the GPU

- A simple, explicit programming language solution
- Extend only where necessary

```
__global__ void KernelFunc(...);
__device__ int GlobalVar;
__shared__ int SharedVar;

KernelFunc<<< 500, 128 >>>(...);
```


Execution Model



Multiple levels of parallelism

- Thread block
  - Up to 512 threads per block
  - Communicate through shared memory
  - Threads guaranteed to be resident
  - `threadIdx, blockIdx`
  - `__syncthreads()`
- Grid of thread blocks
  - `f<<<nblocks, nthreads>>>(a,b,c)`

C-Code Example to Add Arrays

CPU C program

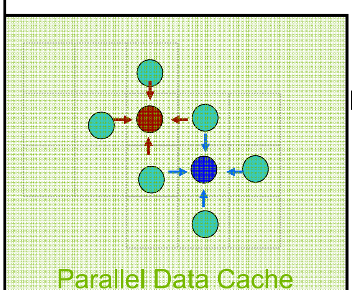
```
void add_matrix_cpu
              (float *a, float *b, float *c, int N)
{
    int i, j, index;
    for (i=0;i<N;i++) {
     for (j=0;j<N;j++) {
        index =i+j*N;
        c[index]=a[index]+b[index];
        }
    }
}
void main()
{
   .....
      add_matrix(a,b,c,N);
}
```

CUDA C program

```
__global__ void add_matrix_gpu
              (float *a, float *b, float *c, int N)
{
    int i=blockIdx.x*blockDim.x+threadIdx.x;
    int j=blockIdx.y*blockDim.y+threadIdx.y;
    int index =i+j*N;
    if( i <N && j <N) c[index]=a[index]+b[index];
}


void main()
{
    dim3 dimBlock (blocksize,blocksize);
    dim3 dimGrid (N/dimBlock.x,N/dimBlock.y);
    add_matrix_gpu<<<dimGrid,dimBlock>>>(a,b,c,N);
}
```


Example Algorithm - Fluids



Pressure depends on neighbors

Goal: Calculate PRESSURE in a fluid

Pressure = Sum of neighboring pressures

$P_n' = P_1 + P_2 + P_3 + P_4$

So the pressure for each particle is…

$Pressure_1 = P_1 + P_2 + P_3 + P_4$

$Pressure_2 = P_3 + P_4 + P_5 + P_6$

$Pressure_3 = P_5 + P_6 + P_7 + P_8$

$Pressure_4 = P_7 + P_8 + P_9 + P_{10}$

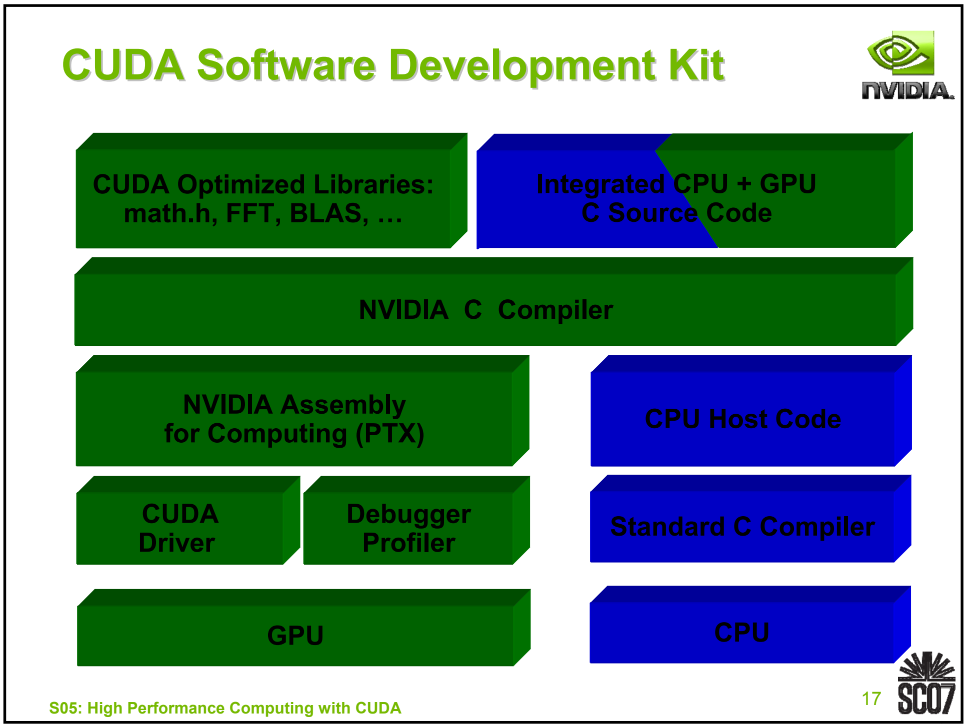
⋮

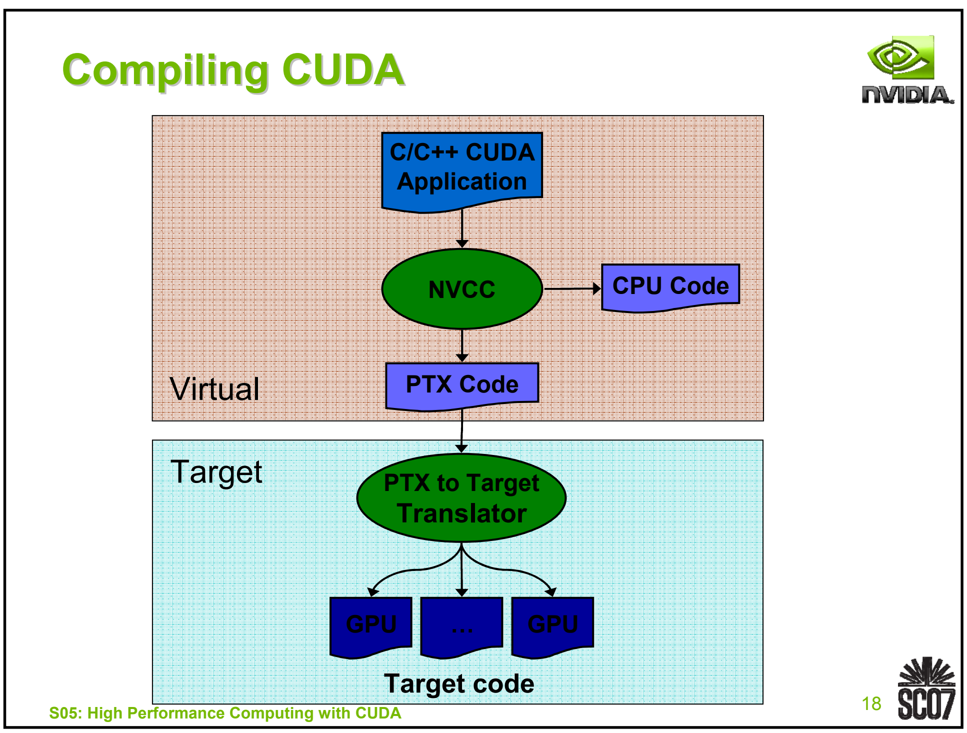
Divergence in Parallel Computing

- Removing divergence pain from parallel programming
- SIMD Pain
  - User required to SIMD-ify
  - User suffers when computation goes divergent
- GPUs: Decouple execution width from programming model
  - Threads can diverge freely
  - Inefficiency only when granularity exceeds native machine width
  - Hardware managed
  - Managing divergence becomes performance optimization
  - Scalable

Runtime Component: Memory Management

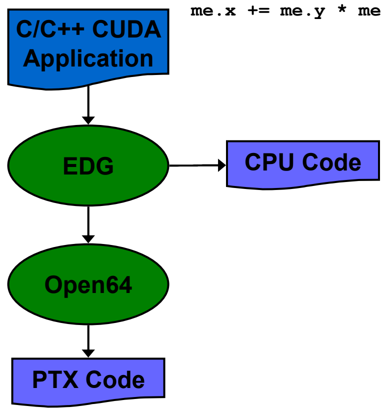
- Explicit GPU memory allocation
- Returns pointers to GPU memory
- Device memory allocation
  - `cudaMalloc(), cudaFree()`
- Memory copy from host to device, device to host, device to device
  - `cudaMemcpy(), cudaMemcpy2D(), ...`
- OpenGL & DirectX interoperability
  - `cudaGLMapBufferObject()`


CUDA Software Development Kit
CUDA Optimized Libraries: math.h, FFT, BLAS, …
Integrated CPU + GPU C Source Code
NVIDIA C Compiler
NVIDIA Assembly for Computing (PTX)
CPU Host Code
CUDA Driver
Debugger Profiler
Standard C Compiler
GPU
CPU
S05: High Performance Computing with CUDA
17


Compiling CUDA
C/C++ CUDA Application
NVCC
CPU Code
Virtual
PTX Code
Target
PTX to Target Translator
GPU
…
GPU
Target code
S05: High Performance Computing with CUDA
18

Standard C Compiler

```
float4 me = gx[gtid];
me.x += me.y * me.z;
```

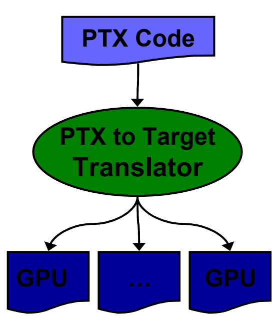


- EDG
  - Separates GPU and CPU code
- Open64
  - Generates GPU PTX assembly
- Parallel Thread eXecution (PTX)
  - Virtual Machine and ISA
  - Programming model
  - Execution resources and state

```
ld.global.v4.f32  {$f1,$f3,$f5,$f7}, [$r9+0];
mad.f32           $f1, $f5, $f3, $f1;
```


Virtual to Target ISA Translation

```
ld.global.v4.f32  {$f1,$f3,$f5,$f7},[$r9+0];
mad.f32           $f1,$f5,$f3,$f1;
```



Target code

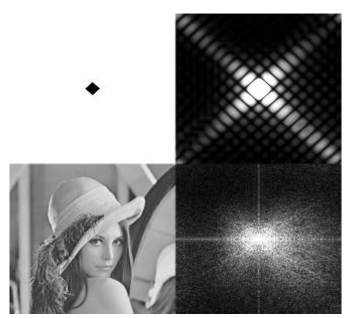
```
0x103c8009  0x0fffffff
0xd00e0609  0xa0c00780
0x100c8009  0x00000003
0x21000409  0x07800780
```

- Parallel Thread eXecution (PTX)
  - Virtual Machine and ISA
  - Distribution format for applications
  - Install-time translation
  - “fat binary” caches target-specific versions
- Target-specific optimization
  - ISA diffences
  - Resource allocation
  - Performance

CUBLAS Library

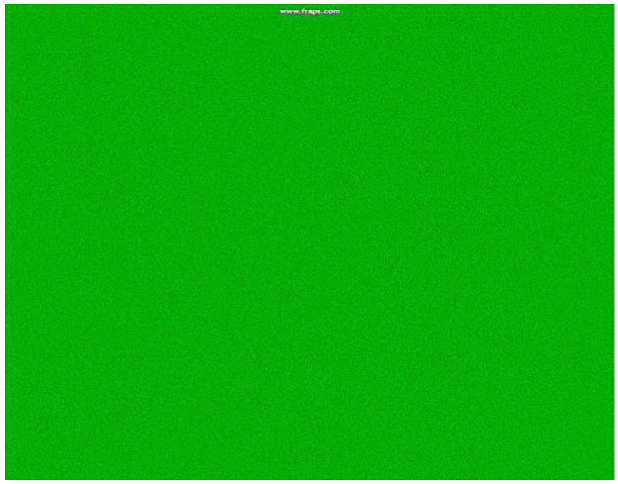
- Self-contained BLAS library
  - Application needs no direct interaction with CUDA driver
- Currently only a subset of BLAS core functions
  - Single/Real Routines, BLAS1 Complex, CGEMM
- Simple to use:
  - Create matrix and vector objects in GPU memory
  - Fill them with data
  - Call sequence of CUBLAS functions
  - Upload results back from GPU to host
- Column-major storage and 1-based indexing
  - For maximum compatibility with existing Fortran apps

CUFFT Library



- Efficient of FFT on CUDA
- Features
  - 1D, 2D, and 3D FFTs of complex and real-valued signal data
  - Batch execution for multiple 1D transforms in parallel
  - Transform sizes (for 1D) in the range [2, 16M]
  - Transform sizes (for 2D and 3D) in the range [2, 16384]

CUDA Stable Fluids Demo



CUDA port of:
Jos Stam, "Stable Fluids", In SIGGRAPH 99 Conference Proceedings, Annual Conference Series, August 1999, 121-128.

Single Precision Floating Point

	8-Series GPU	SSE	IBM Altivec	Cell SPE
Precision	IEEE 754	IEEE 754	IEEE 754	IEEE 754
Rounding modes for FADD and FMUL	Round to nearest and round to zero	All 4 IEEE, round to nearest, zero, inf, -inf	Round to nearest only	Round to zero/truncate only
Denormal handling	Flush to zero	Supported, 1000's of cycles	Supported, 1000's of cycles	Flush to zero
NaN support	Yes	Yes	Yes	No
Overflow and Infinity support	Yes	Yes	Yes	No infinity, only clamps to max norm
Flags	No	Yes	Yes	Some
Square root	Software only	Hardware	Software only	Software only
Division	Software only	Hardware	Software only	Software only
Reciprocal estimate accuracy	24 bit	12 bit	12 bit	12 bit
Reciprocal sqrt estimate accuracy	23 bit	12 bit	12 bit	12 bit
log2(x) and 2^x estimates accuracy	23 bit	No	12 bit	No

GPU Computing Roadmap

	2006	2007				2008
	Q4	Q1	Q2	Q3	Q4	
Processors	G80 November • First Compute					Floating Point • Double precision
CUDA/compiler		Beta February • Wide release	1.0 June • Production release		1.1 Nov • WinXP64 • Async	

More Info

http://www.nvidia.com/cuda