EECS 570
Lecture 4
GPU and DLP
Winter 2018
Prof. Narayanasamy

http://www.eecs.umich.edu/courses/eecs570/

Slides developed in part by Profs. Adve, Falsafi, Martin, Roth, Nowatzyk, and Wenisch of EPFL, CMU, UPenn, U-M, UIUC.
Readings

For today:

- Kim et al, Performance Analysis and Tuning for General Purpose Graphics Processing Units (GPGPU), Ch. 1

For Monday 1/23:

Data-Level Parallelism
How to Compute This Fast?

• Performing the **same** operations on **many** data items
  ☑ Example: SAXPY

```plaintext
for (I = 0; I < 1024; I++) {
  Z[I] = A*X[I] + Y[I];
}
```

L1: `ldf [X+r1]-->f1`  // I is in r1
    `mulf f0,f1-->f2`  // A is in f0
    `ldf [Y+r1]-->f3`
    `addf f2,f3-->f4`
    `stf f4->[Z+r1]`
    `addi r1,4-->r1`
    `blti r1,4096,L1`

• Instruction-level parallelism (ILP) - fine grained
  ☑ Loop unrolling with static scheduling –or– dynamic scheduling
  ☑ Wide-issue superscalar (non-)scaling limits benefits

• Thread-level parallelism (TLP) - coarse grained
  ☑ Multicore

• Can we do some “medium grained” parallelism?
Data-Level Parallelism

• Data-level parallelism (DLP)
  - Single operation repeated on multiple data elements
    - SIMD (Single-Instruction, Multiple-Data)
  - Less general than ILP: parallel insns are all same operation
  - Exploit with vectors

• Old idea: Cray-1 supercomputer from late 1970s
  - Eight 64-entry x 64-bit floating point “Vector registers”
    - 4096 bits (0.5KB) in each register! 4KB for vector register file
  - Special vector instructions to perform vector operations
    - Load vector, store vector (wide memory operation)
    - Vector+Vector addition, subtraction, multiply, etc.
    - Vector+Constant addition, subtraction, multiply, etc.
    - In Cray-1, each instruction specifies 64 operations!
  - ALUs were expensive, did not perform 64 ops in parallel!
• One way to exploit data level parallelism: vectors
  □ Extend processor with vector “data type”
  □ Vector: array of 32-bit FP numbers
    ○ Maximum vector length (MVL): typically 8–64
  □ Vector register file: 8–16 vector registers (v0–v15)
Today's Vectors / SIMD
Example Vector ISA Extensions (SIMD)

• Extend ISA with floating point (FP) vector storage ...
  □ Vector register: fixed-size array of 32- or 64- bit FP elements
  □ Vector length: For example: 4, 8, 16, 64, ...

• ... and example operations for vector length of 4
  □ Load vector: \texttt{ldf.v} [X+r1] \rightarrow v1
    \begin{align*}
    \texttt{ldf} & \ [X+r1+0] \rightarrow v1_0 \\
    \texttt{ldf} & \ [X+r1+1] \rightarrow v1_1 \\
    \texttt{ldf} & \ [X+r1+2] \rightarrow v1_2 \\
    \texttt{ldf} & \ [X+r1+3] \rightarrow v1_3
    \end{align*}
  □ Add two vectors: \texttt{addf.vv} v1,v2 \rightarrow v3
    \begin{align*}
    \texttt{addf} & \ v1_i,v2_i \rightarrow v3_i \text{ (where } i \text{ is 0,1,2,3)}
    \end{align*}
  □ Add vector to scalar: \texttt{addf.vs} v1,f2,v3
    \begin{align*}
    \texttt{addf} & \ v1_i,f2 \rightarrow v3_i \text{ (where } i \text{ is 0,1,2,3)}
    \end{align*}

• Today’s vectors: short (128 bits), but fully parallel
Example Use of Vectors - 4-wide

Operations

- Load vector: `ldf.v [X+r1] -> v1`
- Multiply vector to scalar: `mulf.vs v1, f2 -> v3`
- Add two vectors: `addf.vv v1, v2 -> v3`
- Store vector: `stf.v v1 -> [X+r1]`

• Performance?
  - Best case: 4x speedup
  - But, vector instructions don’t always have 1-cycle throughput
    - Execution width (implementation) vs vector width (ISA)
Vector Datapath & Implementation

- Vector insn. are just like normal insn... only “wider”
  - Single instruction fetch (no extra N^2 checks)
  - Wide register read & write (not multiple ports)
  - Wide execute: replicate FP unit (same as superscalar)
  - Wide bypass (avoid N^2 bypass problem)
  - Wide cache read & write (single cache tag check)

- Execution width (implementation) vs vector width (ISA)
  - E.g. Pentium 4 and “Core 1” executes vector ops at half width
  - “Core 2” executes them at full width

- Because they are just instructions...
  - ...superscalar execution of vector instructions
  - Multiple n-wide vector instructions per cycle
Intel’s SSE2/SSE3/SSE4...

- **Intel SSE2 (Streaming SIMD Extensions 2) - 2001**
  - 16 128bit floating point registers (\texttt{xmm0–xmm15})
  - Each can be treated as 2x64b FP or 4x32b FP (“packed FP”)
    - Or 2x64b or 4x32b or 8x16b or 16x8b ints (“packed integer”)
    - Or 1x64b or 1x32b FP (just normal scalar floating point)
  - Original SSE: only 8 registers, no packed integer support

- Other vector extensions
  - AMD 3DNow!: 64b (2x32b)
  - PowerPC AltiVEC/VMX: 128b (2x64b or 4x32b)

- Intel’s AVX-512
  - Intel’s “Haswell” and Xeon Phi brought 512-bit vectors to x86
Other Vector Instructions

- These target specific domains: e.g., image processing, crypto
  - Vector reduction (sum all elements of a vector)
  - Geometry processing: 4x4 translation/rotation matrices
  - Saturating (non-overflowing) subword add/sub: image processing
  - Byte asymmetric operations: blending and composition in graphics
  - Byte shuffle/permute: crypto
  - Population (bit) count: crypto
  - Max/min/argmax/argmin: video codec
  - Absolute differences: video codec
  - Multiply-accumulate: digital-signal processing
  - Special instructions for AES encryption

- More advanced (but in Intel’s Xeon Phi)
  - Scatter/gather loads: indirect store (or load) from a vector of pointers
  - Vector mask: predication (conditional execution) of specific elements
Using Vectors in Your Code
Using Vectors in Your Code

• Write in assembly
  ❑ Ugh

• Use “intrinsic” functions and data types
  ❑ For example: _mm_mul_ps() and “__m128” datatype

• Use vector data types
  ❑ typedef double v2df __attribute__((vector_size (16)));

• Use a library someone else wrote
  ❑ Let them do the hard work
  ❑ Matrix and linear algebra packages

• Let the compiler do it (automatic vectorization, with feedback)
  ❑ GCC’s “-ftree-vectorize” option, -ftree-vectorizer-verbose=n
  ❑ Limited impact for C/C++ code (old, hard problem)
SAXPY Example: Best Case

• Code
void saxpy(float* x, float* y,
    float* z, float a,
    int length) {
    for (int i = 0; i < length; i++) {
        z[i] = a*x[i] + y[i];
    }
}

• Scalar
.L3:
    movss (%rdi,%rax), %xmm1
    mulss %xmm0, %xmm1
    addss (%rsi,%rax), %xmm1
    movss %xmm1, (%rdx,%rax)
    addq $4, %rax
    cmpq %rcx, %rax
    jne .L3

• Auto Vectorized
.L6:
    movaps (%rdi,%rax), %xmm1
    mulps %xmm2, %xmm1
    addps (%rsi,%rax), %xmm1
    movaps %xmm1, (%rdx,%rax)
    addq $16, %rax
    incl %r8d
    cmpl %r8d, %r9d
    ja .L6

+ Scalar loop to handle last few iterations (if length % 4 != 0)
+ “mulps”: multiply packed ‘single’
SAXPY Example: Actual

- **Code**

```c
void saxpy(float* x, float* y, float* z, float a, int length) {
    for (int i = 0; i < length; i++) {
        z[i] = a * x[i] + y[i];
    }
}
```

- **Scalar**

```
.L3:
movss (%rdi,%rax), %xmm1
mulss %xmm0, %xmm1
addss (%rsi,%rax), %xmm1
movss %xmm1, (%rdx,%rax)
addq $4, %rax
cmpq %rcx, %rax
jne .L3
```

- **Auto Vectorized**

```
.L8:
movaps %xmm3, %xmm1
movaps %xmm3, %xmm2
movlps (%rdi,%rax), %xmm1
movlps (%rsi,%rax), %xmm2
movhps 8(%rdi,%rax), %xmm1
movhps 8(%rsi,%rax), %xmm2
mulps %xmm4, %xmm1
incl %r8d
addps %xmm2, %xmm1
movaps %xmm1, (%rdx,%rax)
addq $16, %rax
cmpl %r9d, %r8d
jb .L8
```

- + Explicit alignment test
- + Explicit aliasing test
Bridging “Best Case” and “Actual”

• Align arrays

```c
typedef float afloat __attribute__((__aligned__(16)));
void saxpy(afloat* x,
          afloat* y,
          afloat* z,
          float a, int length) {
    for (int i = 0; i < length; i++) {
        z[i] = a*x[i] + y[i];
    }
}
```

• Avoid aliasing check

```c
typedef float afloat __attribute__((__aligned__(16)));
void saxpy(afloat* __restrict__ x,
          afloat* __restrict__ y,
          afloat* __restrict__ z, float a, int length)
```

• Even with both, still has the “last few iterations” code
Reduction Example

• Code

```c
void saxpy(float* x, float* y, float* z, float a, int length) {
    for (int i = 0; i < length; i++) {
        z[i] = a*x[i] + y[i];
    }
}
```

• Scalar

```assembly
.L3:
    movss (%rdi,%rax), %xmm1
    mulss %xmm0, %xmm1
    addss (%rsi,%rax), %xmm1
    movss %xmm1, (%rcx,%rax)
    addq $4, %rax
    cmpq %rcx, %rax
    jne .L3
```

• Auto Vectorized

```assembly
.L7:
    movaps (%rdi,%rax), %xmm0
    incl %ecx
    subps (%rsi,%rax), %xmm0
    addq $16, %rax
    addps %xmm0, %xmm1
    cmpl %ecx, %r8d
    ja .L7
    haddps %xmm1, %xmm1
    haddps %xmm1, %xmm1
    movaps %xmm1, %xmm0
    je .L3
```

○ “haddps”: Packed Single-FP Horizontal Add

SSE2 on Pentium 4
Tomorrow’s “CPU” Vectors
Beyond Today’s Vectors

- Today’s vectors are limited
  - Wide compute
  - Wide load/store of consecutive addresses
  - Allows for “SOA” (structures of arrays) style parallelism

- Looking forward (and backward)...
  - **Vector masks**
    - Conditional execution on a per-element basis
    - Allows vectorization of conditionals
  - **Scatter/gather**
    - $a[i] = b[y[i]]$  
      - $b[y[i]] = a[i]$  
    - Helps with sparse matrices, “AOS” (array of structures) parallelism

- Together, enables a different style vectorization
  - Translate arbitrary (parallel) loop bodies into vectorized code (later)
Vector Masks (Predication)

- **Vector Masks**: 1 bit per vector element
  - Implicit predicate in all vector operations
    ```
    for (I=0; I<N; I++) if (maskI) { vop... }
    ```
  - Usually stored in a “scalar” register (up to 64-bits)
  - Used to vectorize loops with conditionals in them
    ```
    cmp_eq.v, cmp_lt.v, etc.: sets vector predicates
    for (I=0; I<32; I++)
      if (X[I] != 0.0) Z[I] = A/X[I];
    ```

```plaintext
ldf.v [X+r1] -> v1
cmp_ne.v v1,f0 -> r2 // 0.0 is in f0
divf.sv {r2} v1,f1 -> v2 // A is in f1
stf.v {r2} v2 -> [Z+r1]
```
Scatter Stores & Gather Loads

• How to vectorize:
  ```
  for(int i = 1, i<N, i++) {
    int bucket = val[i] / scalefactor;
    found[bucket] = 1;
  }
  ```
  □ Easy to vectorize the divide, but what about the load/store?

• Solution: hardware support for vector “scatter stores”
  □ stf.v v2->[r1+v1]
  □ Each address calculated from r1+v1
    stf v20->[r1+v10],   stf v21->[r1+v11],
    stf v22->[r1+v12],   stf v23->[r1+v13]

• Vector “gather loads” defined analogously
  □ ldf.v [r1+v1]->v2

• Scatter/gathers slower than regular vector load/store ops
  □ Still provides throughput advantage over non-vector version
New Developments in “CPU” Vectors
Emerging Features

- Past vectors were limited
  - Wide compute
  - Wide load/store of consecutive addresses
  - Allows for “SOA” (structures of arrays) style parallelism

- Looking forward (and backward)...
  - Vector masks
    - Conditional execution on a per-element basis
    - Allows vectorization of conditionals
  - Scatter/gather
    - $a[i] = b[y[i]]$, $b[y[i]] = a[i]$
    - Helps with sparse matrices, “AOS” (array of structures) parallelism

- Together, enables a different style vectorization
  - Translate arbitrary (parallel) loop bodies into vectorized code (later)
Vector Masks (Predication)

- **Vector Masks**: 1 bit per vector element
  - Implicit predicate in all vector operations
    for (I=0; I<N; I++) if (maskI) { vop... }
  - Usually stored in a “scalar” register (up to 64-bits)
  - Used to vectorize loops with conditionals in them
    cmp_eq.v, cmp_lt.v, etc.: sets vector predicates
    
    for (I=0; I<32; I++)
       if (X[I] != 0.0) Z[I] = A/X[I];

    ldf.v [X+r1] -> v1
    cmp_ne.v v1,f0 -> r2     // 0.0 is in f0
    divf.sv {r2} v1,f1 -> v2   // A is in f1
    stf.v {r2} v2 -> [Z+r1]
Scatter Stores & Gather Loads

• How to vectorize:
  
  ```c
  for(int i = 1, i<N, i++) {
      int bucket = val[i] / scalefactor;
      found[bucket] = 1;
  }
  ```

  ❐ Easy to vectorize the divide, but what about the load/store?

• Solution: hardware support for vector “scatter stores”
  
  ❐ stf.v v2->[r1+v1]

  ❐ Each address calculated from r1+v1i
  
  stf v20->[r1+v10], stf v21->[r1+v11],
  stf v22->[r1+v12], stf v23->[r1+v13]

• Vector “gather loads” defined analogously
  
  ❐ ldf.v [r1+v1]->v2

• Scatter/gathers slower than regular vector load/store ops
  
  ❐ Still provides throughput advantage over non-vector version
Today’s GPU’s “SIMT” Model
Graphics Processing Units (GPU)

- Killer app for parallelism: graphics (3D games)

- A quiet revolution and potential build-up
  - Calculation: 367 GFLOPS vs. 32 GFLOPS
  - Memory Bandwidth: 86.4 GB/s vs. 8.4 GB/s
  - Until recently, programmed through graphics API

- GPU in every desktop, laptop, mobile device
  - massive volume and potential impact

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2009 ECE 498AL, University of Illinois, Urbana-Champaign
What is Behind such an Evolution?

- The GPU is specialized for compute-intensive, highly data parallel computation (exactly what graphics rendering is about)
  - So, more transistors can be devoted to data processing rather than data caching and flow control

- The fast-growing video game industry exerts strong economic pressure that forces constant innovation
GPUs and SIMD/Vector Data Parallelism

• Graphics processing units (GPUs)
  ❑ How do they have such high peak FLOPS?
  ❑ Exploit massive data parallelism

• “SIMT” execution model
  ❑ Single instruction multiple threads
  ❑ Similar to both “vectors” and “SIMD”
  ❑ A key difference: better support for conditional control flow

• Program it with CUDA or OpenCL
  ❑ Extensions to C
  ❑ Perform a “shader task” (a snippet of scalar computation) over many elements
  ❑ Internally, GPU uses scatter/gather and vector mask operations
Context: History of Programming GPUs

• “GPGPU”
  - Originally could only perform “shader” computations on images
  - So, programmers started using this framework for computation
  - Puzzle to work around the limitations, unlock the raw potential

• As GPU designers notice this trend...
  - Hardware provided more “hooks” for computation
  - Provided some limited software tools

• GPU designs are now fully embracing compute
  - More programmability features to each generation
  - Industrial-strength tools, documentation, tutorials, etc.
  - Can be used for in-game physics, etc.
  - A major initiative to push GPUs beyond graphics (HPC)
GPU Architectures

- NVIDIA G80 – extreme SIMD parallelism in shader units
Throughput Computing: Hardware Basics

Justin Hensley
Advanced Micro Devices, Inc
Graphics Product Group

adapted from Kayvon Fatahalian’s SIGGRAPH’08 talk
What does a modern graphics API do?

Vertex Assembly → Vertex Shader → Geometry Assembly → Geometry Shader → Scan Conversion → Pixel Shader → Blend → Display
A Simple Program - Diffuse Shader

```c
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;
float4 diffuseShader(float3 norm, float2 uv)
{
    float3 kd;
    kd = myTex.Sample(mySamp, uv);
    kd *= clamp( dot(lightDir, norm), 0.0, 1.0);
    return float4(kd, 1.0);
}
```

Each invocation is independent, but no explicitly exposed parallelism
Shader is compiled

1 Unshaded fragment in

```c
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;
float4 diffuseShader(float3 norm, float2 uv)
{
    float3 kd;
    kd = myTex.Sample(mySamp, uv);
    kd *= clamp(dot(lightDir, norm), 0.0, 1.0);
    return float4(kd, 1.0);
}
```

1 Shaded fragment out

```c
<diffuseShader>:
sample r0, v4, t0, s0
mul r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, 1(0.0), 1(1.0)
mul o0, r0, r3
mul o1, r1, r3
mul o2, r2, r3
mov o3, 1(1.0a)
```
Exploit data parallelism! - add two cores

Each invocation is independent!
Add even more cores - four cores

adapted from Kayvon Fatahalian’s SIGGRAPH’08 talk
How about even more cores - 16 cores

adapted from Kayvon Fatahalian’s SIGGRAPH’08 talk
128 cores?

How do you feed all these cores?

Think data parallel! - Graphics requires hardware process *lots* of “items” that share the same shader.
Back to the simple core...

• How do you feed all these cores?

• Share cost of fetch / decode across many ALUs

• **SIMD** Processing
Back to the simple core...

- How do you feed all these cores?
- Share cost of fetch / decode across many ALUs
- **SIMD** Processing
  - Single
  - Instruction
  - Multiple
  - Data
Back to the simple core...

- How do you feed all these cores?
- Share cost of fetch / decode across many ALUs
- **SIMD** Processing
  - Single

**SIMD Processing does not imply SIMD instructions!**
Back to a single core...

```c
<diffuseShader>:
sample r0, v4, t0, s0
mul  r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, 1(0.0), 1(1.0)
mul  o0, r0, r3
mul  o1, r1, r3
mul  o2, r2, r3
mov  o3, 1(1.0)
```
128-Fragments in parallel

16 cores ➔ 128 ALUs (16 cores * 8 ALUs)
➔ 16 independent instruction streams

adapted from Kayvon Fatahalian’s SIGGRAPH’08 talk
128-things in parallel

• X cores can work on primitives (triangles)
  – "geometry shader"
• Y cores can work on vertices
  – "vertex shader"
• Z cores can work on fragments
  – "pixel shader"
• N cores can work on data/work/etc
  – "compute kernels"/"compute shaders"
• Which cores working on what data changes over time
What about branching?

```c
if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}
```

<resume unconditional shader code>
What about branching?

```c
if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}
```

<resume unconditional shader code>
What about branching?

Not all ALUs do useful work!
Worst case: 1/8 performance

if \( x > 0 \) {
    y = pow(x, \text{exp});
    y *= \text{Ks};
    \text{refl} = y + \text{Ka};
} \text{else} {
    x = 0;
    \text{refl} = \text{Ka};
}
What about branching?

```
<unconditional shader code>
if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}
<resume unconditional shader code>
```
How to handle stalls?

• Memory access latency = 100’s to 1000’s of cycles
  – Stalls occur when a core cannot run the next instruction

• GPUs don’t have the large / fancy caches and logic that helps avoid stall because of a dependency on a previous operation.

• But we have LOTS of independent fragments.
  – Interleave processing of many fragments on a single core to avoid stalls caused by high latency operations.
Hiding Memory Stalls

Time (clocks)

Frag 1 ... 8

Fetch/Decode
ALU ALU ALU ALU
ALU ALU ALU ALU
Ctx Ctx Ctx Ctx
Ctx Ctx Ctx Ctx
Shared Ctx Data

adapted from Kayvon Fatahalian’s SIGGRAPH’08 talk
Hiding Memory Stalls

Time (clocks)

Frag 1 ... 8
Frag 9 ... 16
Frag 17 ... 24
Frag 25 ... 32

Fetch/Decode

ALU ALU ALU ALU
ALU ALU ALU ALU

adapted from Kayvon Fatahalian’s SIGGRAPH’08 talk
Hiding Memory Stalls

Time (clocks)

Frag 1 ... 8
Frag 9 ... 16
Frag 17 ... 24
Frag 25 ... 32

Stall

Runnable

adapted from Kayvon Fatahalian’s SIGGRAPH’08 talk
Hiding Memory Stalls

Time (clocks)

Frag 1 ... 8
Frag 9 ... 16
Frag 17 ... 24
Frag 25 ... 32

Stall
Runnable

adapted from Kayvon Fatahalian’s SIGGRAPH’08 talk
Hiding Memory Stalls

Time (clocks)

Frag 1 ... 8
Runnables
Stall

Frag 9 ... 16
Runnables
Stall

Frag 17 ... 24
Runnables
Stall

Frag 25 ... 32

adapted from Kayvon Fatahalian’s SIGGRAPH’08 talk
Throughput computing

Increase run time of one group
To maximum throughput of many groups

adapted from Kayvon Fatahalian’s SIGGRAPH’08 talk
Latency Hiding with “Thread Warps”

- **Warp**: A set of threads that execute the same instruction (on different data elements)

- **Fine-grained multithreading**
  - One instruction per thread in pipeline at a time (No branch prediction)
  - Interleave warp execution to hide latencies

- Register values of all threads stay in register file

- No OS context switching

- Memory latency hiding
  - Graphics has millions of pixels

Slide credit: Tor Aamodt
Warp-based SIMD vs. Traditional SIMD

• Traditional SIMD contains a single thread
  - Lock step
  - Programming model is SIMD (no threads) → SW needs to know vector length
  - ISA contains vector/SIMD instructions

• Warp-based SIMD consists of multiple scalar threads executing in a SIMD manner (i.e., same instruction executed by all threads)
  - Does not have to be lock step
  - Each thread can be treated individually (i.e., placed in a different warp) → programming model not SIMD
    - SW does not need to know vector length
    - Enables memory and branch latency tolerance
  - ISA is scalar → vector instructions formed dynamically
CUDA In One Slide

Thread

<table>
<thead>
<tr>
<th>per-thread local memory</th>
</tr>
</thead>
</table>

Block

| per-block shared memory |

Local barrier

Global barrier

Kernel `foo()`

Kernel `bar()`

per-device global memory
CUDA Devices and Threads

- A compute **device**
  - Is a coprocessor to the CPU or **host**
  - Has its own DRAM (**device memory**)
  - Runs many **threads in parallel**
  - Is typically a **GPU** but can also be another type of parallel processing device

- Data-parallel portions of an application are expressed as device **kernels** which run on many threads

- 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
Thread Batching: Grids and Blocks

- A kernel is executed as a **grid of thread blocks**
  - All threads share data memory space

- A **thread block** is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory

- Two threads from two different blocks cannot cooperate
Execution Model

• Each thread block is executed by a single multiprocessor
  ❑ Synchronized using shared memory

• Many thread blocks are assigned to a single multiprocessor
  ❑ Executed concurrently in a time-sharing fashion
  ❑ Keep GPU as busy as possible

• Running many threads in parallel can hide DRAM memory latency
  ❑ Global memory access : 2~300 cycles
CUDA Device Memory Space Overview

- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory

- The host can R/W global, constant, and texture memories
Example: Vector Addition Kernel

// Pair-wise addition of vector elements
// One thread per addition

__global__ void vectorAdd(float* iA, float* iB, float* oC)
{
    int idx = threadIdx.x
    + blockDim.x * blockIdx.x;
    oC[idx] = iA[idx] + iB[idx];
}

Courtesy NVIDIA
Example: Vector Addition Host Code

```c
float* h_A = (float*) malloc(N * sizeof(float));
float* h_B = (float*) malloc(N * sizeof(float));
// ... initialize h_A and h_B

// allocate device memory
float* d_A, d_B, d_C;
 cudaMalloc( (void**) &d_A, N * sizeof(float) );
cudaMalloc( (void**) &d_B, N * sizeof(float) );
cudaMalloc( (void**) &d_C, N * sizeof(float) );

// copy host memory to device
 cudaMemcpy( d_A, h_A, N * sizeof(float), cudaMemcpyHostToDevice );
cudaMemcpy( d_B, h_B, N * sizeof(float), cudaMemcpyHostToDevice );

// execute the kernel on N/256 blocks of 256 threads each
 vectorAdd<<<N/256, 256>>>( d_A, d_B, d_C);
```
CUDA-Strengths

• Easy to program (small learning curve)

• Success with several complex applications
  □ At least 7X faster than CPU stand-alone implementations

• Allows us to read and write data at any location in the device memory

• More fast memory close to the processors (registers + shared memory)
CUDA-Limitations

• Some hardwired graphic components are hidden

• Better tools are needed
  □ Profiling
  □ Memory blocking and layout
  □ Binary Translation

• Difficult to find optimal values for CUDA execution parameters
  □ Number of thread per block
  □ Dimension and orientation of blocks and grid
  □ Use of on-chip memory resources including registers and shared memory