# CS252
# Graduate Computer Architecture
# Lecture 25

## Disks and Queueing Theory
## GPUs
## April 25th, 2011

John Kubiatowicz
Electrical Engineering and Computer Sciences
University of California, Berkeley

http://www.eecs.berkeley.edu/~kubitron/cs252

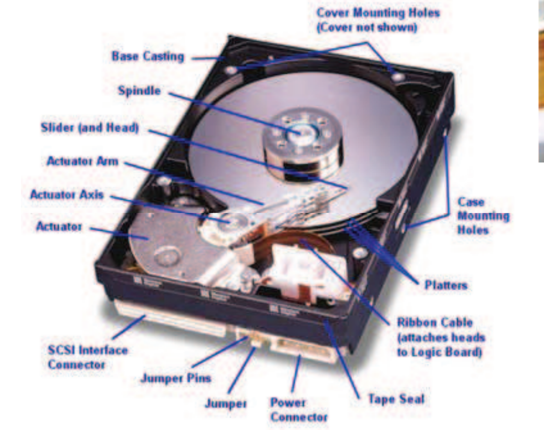
---

## Motivation: Who Cares About I/O?

- CPU Performance: 60% per year
- I/O system performance limited by mechanical delays (disk I/O) or time to access remote services
  - Improvement of < 10% per year (IO per sec or MB per sec)
- Amdahl's Law: system speed-up limited by the slowest part!
  - 10% IO & 10x CPU => 5x Performance (lose 50%)
  - 10% IO & 100x CPU => 10x Performance (lose 90%)
- I/O bottleneck:
  - Diminishing fraction of time in CPU
  - Diminishing value of faster CPUs

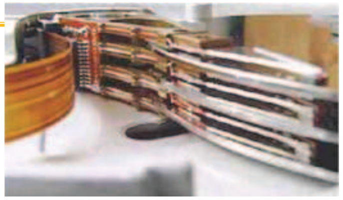
---

## Hard Disk Drives


Cover Mounting Holes (Cover not shown)
Base Casting
Spindle
Slider (and Head)
Actuator Arm
Actuator Axis
Actuator
Case Mounting Holes
Platters
Ribbon Cable (attaches heads to Logic Board)
SCSI Interface Connector
Jumper Pins
Jumper
Power Connector
Tape Seal

Western Digital Drive

http://www.storagereview.com/guide/



Read/Write Head
Side View

1GB Microdrive
512MB Microdrive
4GB Microdrive

IBM/Hitachi Microdrive

---

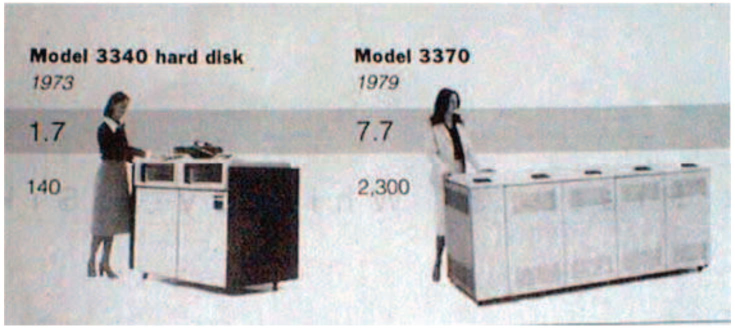
## Historical Perspective

- 1956 IBM Ramac — early 1970s Winchester
  - Developed for mainframe computers, proprietary interfaces
  - Steady shrink in form factor: 27 in. to 14 in.
- Form factor and capacity drives market more than performance
- 1970s developments
  - 5.25 inch floppy disk formfactor (microcode into mainframe)
  - Emergence of industry standard disk interfaces
- Early 1980s: PCs and first generation workstations
- Mid 1980s: Client/server computing
  - Centralized storage on file server
    - » accelerates disk downsizing: 8 inch to 5.25
  - Mass market disk drives become a reality
    - » industry standards: SCSI, IPI, IDE
    - » 5.25 inch to 3.5 inch drives for PCs, End of proprietary interfaces
- 1900s: Laptops => 2.5 inch drives
- 2000s: Shift to perpendicular recording
  - 2007: Seagate introduces 1TB drive
  - 2009: Seagate/WD introduces 2TB drive
  - 2010: Seagate/Hitachi/WD introduce 3TB drives

# Disk History

Data density Mbit/sq. in.

Capacity of Unit Shown Megabytes

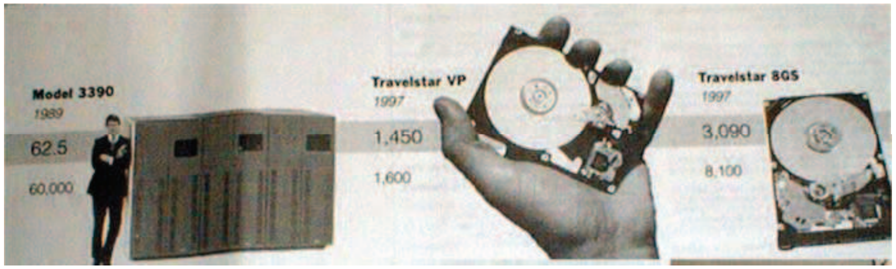


| 1973: | 1979: |
|---|---|
| 1. 7 Mbit/sq. in | 7. 7 Mbit/sq. in |
| 140 MBytes | 2,300 MBytes |

*source: New York Times, 2/23/98, page C3,*
*"Makers of disk drives crowd even mroe data into even smaller spaces"*

---

# Disk History



| 1989: | 1997: | 1997: |
|---|---|---|
| 63 Mbit/sq. in | 1450 Mbit/sq. in | 3090 Mbit/sq. in |
| 60,000 MBytes | 2300 MBytes | 8100 MBytes |

*source: New York Times, 2/23/98, page C3,*
*"Makers of disk drives crowd even mroe data into even smaller spaces"*

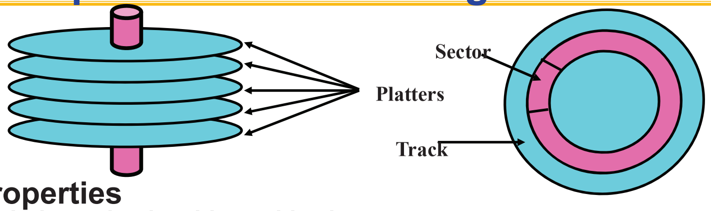
---

# Example: Seagate Barracuda (2010)

- 3TB! 488 Gb/in$^2$
- 5 (3.5") platters, 2 heads each
- Perpendicular recording
- 7200 RPM, 4.16ms latency
- 600MB/sec burst, 149MB/sec sustained transfer speed
- 64MB cache
- Error Characteristics:
  - MBTF: 750,000 hours
  - Bit error rate: $10^{-14}$
- Special considerations:
  - Normally need special "bios" (EFI): Bigger than easily handled by 32-bit OSes.
  - Seagate provides special "Disk Wizard" software that virtualizes drive into multiple chunks that makes it bootable on these OSes.

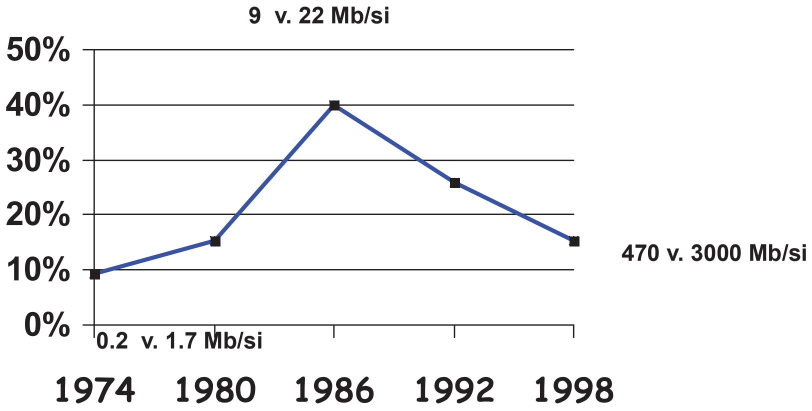
---

# Properties of a Hard Magnetic Disk



- Properties
  - Independently addressable element: sector
    - » OS always transfers groups of sectors together—"blocks"
  - A disk can access directly any given block of information it contains (random access). Can access any file either sequentially or randomly.
  - A disk can be rewritten in place: it is possible to read/modify/write a block from the disk
- Typical numbers (depending on the disk size):
  - 500 to more than 20,000 tracks per surface
  - 32 to 800 sectors per track
    - » A sector is the smallest unit that can be read or written
- Zoned bit recording
  - Constant bit density: more sectors on outer tracks
  - Speed varies with track location

# MBits per square inch: DRAM as % of Disk over time

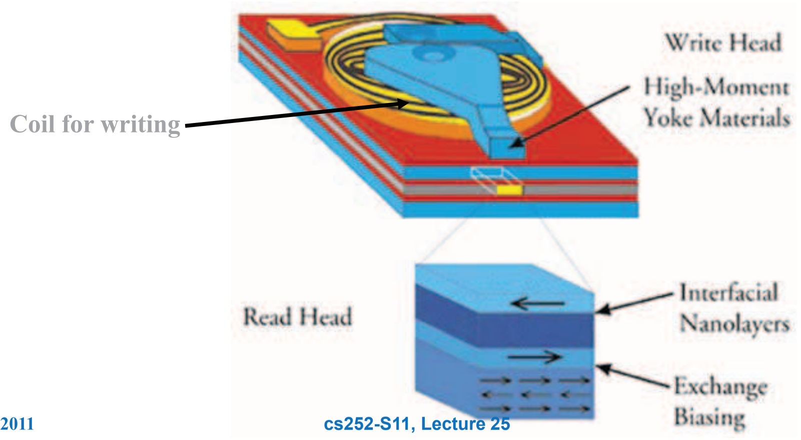


*source: New York Times, 2/23/98, page C3,*
*“Makers of disk drives crowd even mroe data into even smaller spaces”*

---

# Nano-layered Disk Heads

- Special sensitivity of Disk head comes from “Giant Magneto-Resistive effect” or (GMR)
- IBM is (was) leader in this technology
  - Same technology as TMJ-RAM breakthrough

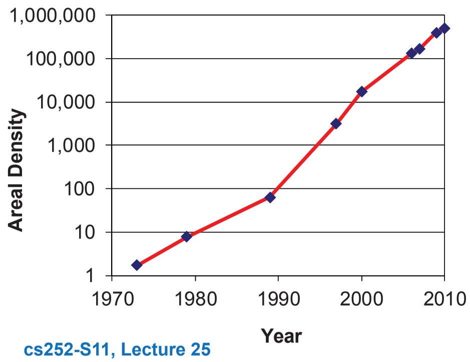

Coil for writing
Write Head
High-Moment Yoke Materials
Read Head
Interfacial Nanolayers
Exchange Biasing

---

# Disk Figure of Merit: Areal Density

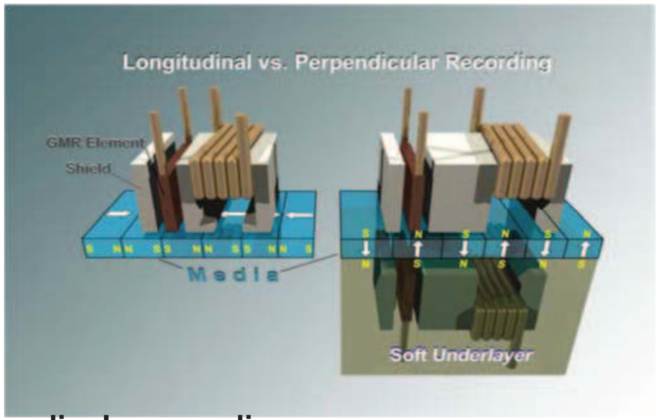
- Bits recorded along a track
  - Metric is Bits Per Inch (BPI)
- Number of tracks per surface
  - Metric is Tracks Per Inch (TPI)
- Disk Designs Brag about bit density per unit area
  - Metric is Bits Per Square Inch: Areal Density = BPI x TPI

| Year | Areal Density |
|---|---|
| 1973 | 2 |
| 1979 | 8 |
| 1989 | 63 |
| 1997 | 3,090 |
| 2000 | 17,100 |
| 2006 | 130,000 |
| 2007 | 164,000 |
| 2009 | 400,000 |
| 2010 | 488,000 |



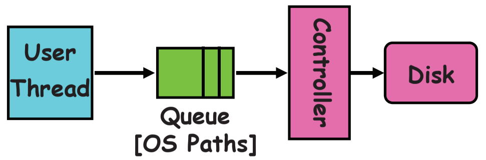
---

# Newest technology: Perpendicular Recording

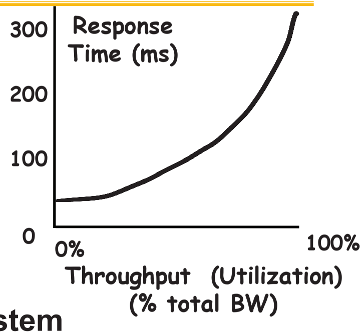

Longitudinal vs. Perpendicular Recording
GMR Element
Shield
Media
Soft Underlayer

- In Perpendicular recording:
  - Bit densities much higher
  - Magnetic material placed on top of magnetic underlayer that reflects recording head and effectively doubles recording field

## Disk I/O Performance



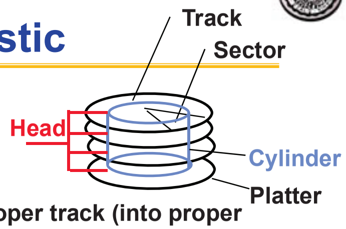
Response Time = Queue+Disk Service Time



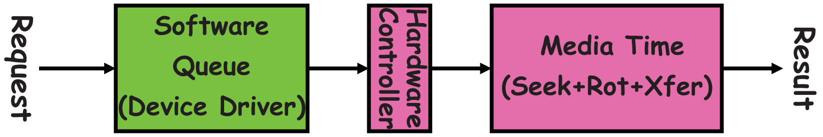
- Performance of disk drive/file system
  - Metrics: Response Time, Throughput
  - Contributing factors to latency:
    - » Software paths (can be loosely modeled by a queue)
    - » Hardware controller
    - » Physical disk media
- Queuing behavior:
  - Can lead to big increase of latency as utilization approaches 100%

---

## Magnetic Disk Characteristic



- Cylinder: all the tracks under the head at a given point on all surface
- Read/write data is a three-stage process:
  - Seek time: position the head/arm over the proper track (into proper cylinder)
  - Rotational latency: wait for the desired sector to rotate under the read/write head
  - Transfer time: transfer a block of bits (sector) under the read-write head
- Disk Latency = Queueing Time + Controller time +
  Seek Time + Rotation Time + Xfer Time



- Highest Bandwidth:
  - transfer large group of blocks sequentially from one track

---

## Disk Time Example

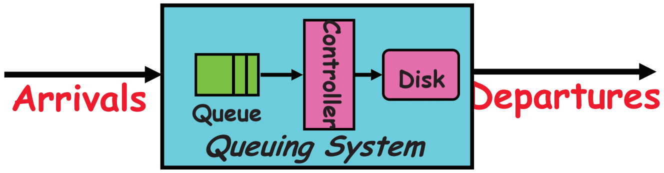
- Disk Parameters:
  - Transfer size is 8K bytes
  - Advertised average seek is 12 ms
  - Disk spins at 7200 RPM
  - Transfer rate is 4 MB/sec
- Controller overhead is 2 ms
- Assume that disk is idle so no queuing delay
- Disk Latency =
  Queuing Time +
  Seek Time + Rotation Time + Xfer Time + Ctrl Time
- What is Average Disk Access Time for a Sector?
  - Ave seek + ave rot delay + transfer time + controller overhead
  - 12 ms + [0.5/(7200 RPM/60s/M)] ×1000 ms/s +
    [8192 bytes/($4 \times 10^6$ bytes/s)] ×1000 ms/s + 2 ms
  - 12 + 4.17 + 2.05 + 2 = 20.22 ms
- Advertised seek time assumes no locality: typically 1/4 to 1/3 advertised seek time: 12 ms => 4 ms

---

## Typical Numbers of a Magnetic Disk

- Average seek time as reported by the industry:
  - Typically in the range of 4 ms to 12 ms
  - Due to locality of disk reference may only be 25% to 33% of the advertised number
- Rotational Latency:
  - *Most* disks rotate at 3,600 to 7200 RPM (Up to 15,000RPM or more)
  - Approximately 16 ms to 8 ms per revolution, respectively
  - An average latency to the desired information is halfway around the disk: 8 ms at 3600 RPM, 4 ms at 7200 RPM
- Transfer Time is a function of:
  - Transfer size (usually a sector): 1 KB / sector
  - Rotation speed: 3600 RPM to 15000 RPM
  - Recording density: bits per inch on a track
  - Diameter: ranges from 1 in to 5.25 in
  - Typical values: 2 to 600 MB per second
- Controller time?
  - Depends on controller hardware—need to examine each case individually

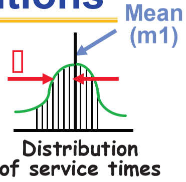
# Introduction to Queuing Theory



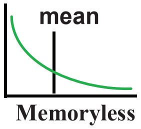
- What about queuing time??
  - Let's apply some queuing theory
  - Queuing Theory applies to long term, steady state behavior ⇒ Arrival rate = Departure rate
- Little's Law:
  **Mean # tasks in system = arrival rate x mean response time**
  - Observed by many, Little was first to prove
  - Simple interpretation: you should see the same number of tasks in queue when entering as when leaving.
- Applies to any system in equilibrium, as long as nothing in black box is creating or destroying tasks
  - Typical queuing theory doesn't deal with transient behavior, only steady-state behavior

---

# Background: Use of random distributions



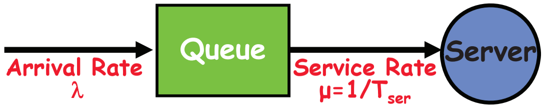
- Server spends variable time with customers
  - Mean (Average) m1 = $\Sigma p(T)\times T$
  - Variance $\sigma^2 = \Sigma p(T)\times(T\text{-}m1)^2 = \Sigma p(T)\times T^2\text{-}m1=E(T^2)\text{-}m1$
  - Squared coefficient of variance: $C = \sigma^2/m1^2$
    Aggregate description of the distribution.
- Important values of C:
  - No variance or deterministic ⇒ C=0
  - "memoryless" or exponential ⇒ C=1
    - » Past tells nothing about future
    - » Many complex systems (or aggregates) well described as memoryless
  - Disk response times C ≈ 1.5 (majority seeks < avg)



- Mean Residual Wait Time, m1(z):
  - Mean time must wait for server to complete current task
  - Can derive m1(z) = ½m1×(1 + C)
    - » Not just ½m1 because doesn't capture variance
  - C = 0 ⇒ m1(z) = ½m1; C = 1 ⇒ m1(z) = m1

---

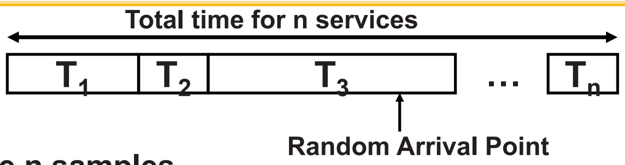
# A Little Queuing Theory: Mean Wait Time



- Parameters that describe our system:
  - $\lambda$: mean number of arriving customers/second
  - $T_{ser}$: mean time to service a customer ("m1")
  - C: squared coefficient of variance = $\sigma^2/m1^2$
  - μ: service rate = $1/T_{ser}$
  - u: server utilization ($0\le u\le 1$): $u = \lambda/\mu = \lambda \times T_{ser}$
- Parameters we wish to compute:
  - $T_q$: Time spent in queue
  - $L_q$: Length of queue = $\lambda \times T_q$ (by Little's law)
- Basic Approach:
  - Customers before us must finish; mean time = $L_q \times T_{ser}$
  - If something at server, takes m1(z) to complete on avg
    - » Chance server busy = u ⇒ mean time is u × m1(z)
- Computation of wait time in queue ($T_q$):
  **$T_q = L_q \times T_{ser} + u \times m1(z)$**

---

# Mean Residual Wait Time: m1(z)



- Imagine n samples
  - There are $n \times P(T_x)$ samples of size $T_x$
  - Total space of samples of size $T_x$: $T_x \times n \times P(T_x) = n \times T_x P(T_x)$
  - Total time for n services: $\sum_x n \times T_x P(T_x) = n \times T_{ser}$
  - Chance arrive in service of length $T_x$: $\dfrac{n \times T_x P(T_x)}{n \times T_{ser}} = \dfrac{T_x P(T_x)}{T_{ser}}$
  - Avg remaining time if land in $T_x$: $\frac{1}{2}T_x$
  - Finally: Average Residual Time m1(z):

$$\sum_x \left(\frac{1}{2}T_x\right)\left(\frac{T_x P(T_x)}{T_{ser}}\right) = \frac{1}{2}\frac{E(T^2)}{T_{ser}} = \frac{1}{2}T_{ser}\left(\frac{\sigma^2 + T_{ser}^{\ 2}}{T_{ser}^{\ 2}}\right) = \frac{1}{2}T_{ser}(1+C)$$

# A Little Queuing Theory: M/G/1 and M/M/1

- Computation of wait time in queue ($T_q$):

  $T_q = L_q \times T_{ser} + u \times m1(z)$ Little's Law

  $T_q = \lambda \times T_q \times T_{ser} + u \times m1(z)$ Defn of utilization (u)

  $T_q = u \times T_q + u \times m1(z)$

  $T_q \times (1 - u) = m1(z) \times u \Rightarrow T_q = m1(z) \times u/(1-u) \Rightarrow$

  $T_q = T_{ser} \times ½(1+C) \times u/(1 - u)$
- Notice that as $u \to 1$, $T_q \to \infty$ !
- Assumptions so far:
  - System in equilibrium; No limit to the queue: works First-In-First-Out
  - Time between two successive arrivals in line are random and memoryless: (M for C=1 exponentially random)
  - Server can start on next customer immediately after prior finishes
- General service distribution (no restrictions), 1 server:
  - Called M/G/1 queue: $T_q = T_{ser} \times ½(1+C) \times u/(1 - u))$
- Memoryless service distribution (C = 1):
  - Called M/M/1 queue: $T_q = T_{ser} \times u/(1 - u)$

---

# A Little Queuing Theory: An Example

- Example Usage Statistics:
  - User requests 10 x 8KB disk I/Os per second
  - Requests & service exponentially distributed (C=1.0)
  - Avg. service = 20 ms (From controller+seek+rot+trans)
- Questions:
  - How utilized is the disk?
    - » Ans: server utilization, $u = \lambda T_{ser}$
  - What is the average time spent in the queue?
    - » Ans: $T_q$
  - What is the number of requests in the queue?
    - » Ans: $L_q$
  - What is the avg response time for disk request?
    - » Ans: $T_{sys} = T_q + T_{ser}$
- Computation:

$\lambda$ (avg # arriving customers/s) = 10/s

$T_{ser}$ (avg time to service customer) = 20 ms (0.02s)

$u$ (server utilization) = $\lambda \times T_{ser}$= 10/s x .02s = 0.2

$T_q$ (avg time/customer in queue) = $T_{ser} \times u/(1 - u)$
= 20 x 0.2/(1-0.2) = 20 x 0.25 = 5 ms (0 .005s)

$L_q$ (avg length of queue) = $\lambda \times T_q$=10/s x .005s = 0.05

$T_{sys}$ (avg time/customer in system) =$T_q + T_{ser}$= 25 ms

---

# Administrivia

- Exam: Finally finished grading!
  - AVG: 64.4, Std: 21.4
  - Sorry for the delay!
  - Please look at my grading to make sure that I didn't mess up
  - Solutions are up – please go through them
- Final dates:
  - Wednesday 5/4: Quantum Computing (and DNA computing?)
  - Thursday 5/5: Oral Presentations: 10-11:30
  - Monday 5/9: Final papers due
    - » 10-pages, double-column, conference format

---

# Types of Parallelism

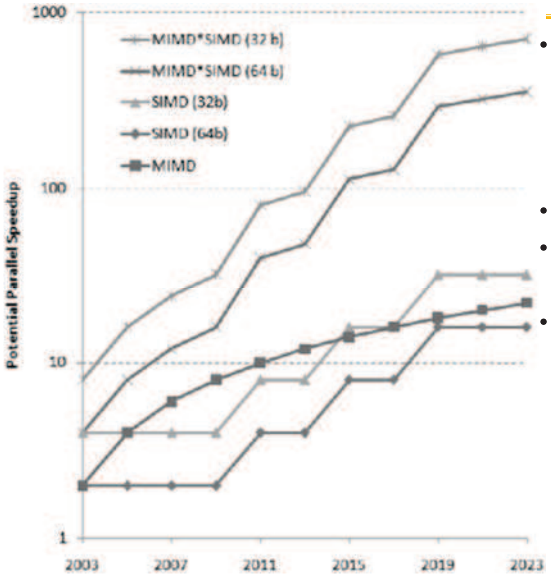
- Instruction-Level Parallelism (ILP)
  - Execute independent instructions from one instruction stream in parallel (pipelining, superscalar, VLIW)
- Thread-Level Parallelism (TLP)
  - Execute independent instruction streams in parallel (multithreading, multiple cores)
- Data-Level Parallelism (DLP)
  - Execute multiple operations of the same type in parallel (vector/SIMD execution)

- Which is easiest to program?
- Which is most flexible form of parallelism?
  - i.e., can be used in more situations
- Which is most efficient?
  - i.e., greatest tasks/second/area, lowest energy/task

# Resurgence of DLP

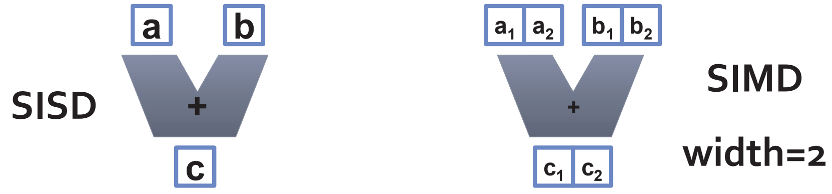
- Convergence of application demands and technology constraints drives architecture choice
- New applications, such as graphics, machine vision, speech recognition, machine learning, etc. all require large numerical computations that are often trivially data parallel
- SIMD-based architectures (vector-SIMD, subword-SIMD, SIMT/GPUs) are most efficient way to execute these algorithms

# DLP important for conventional CPUs too



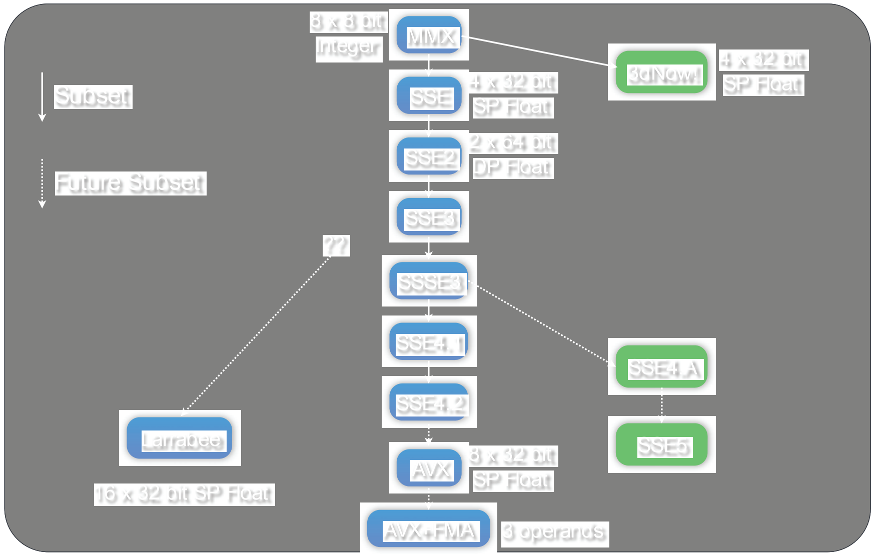
- Prediction for x86 processors, from Hennessy & Patterson, upcoming 5th edition
  - Note: Educated guess, not Intel product plans!
- TLP: 2+ cores / 2 years
- DLP: 2x width / 4 years
- DLP will account for more mainstream parallelism growth than TLP in next decade.
  - SIMD –single-instruction multiple-data (DLP)
  - MIMD- multiple-instruction multiple-data (TLP)

# SIMD


SISD
a
b
+
c
$a_1$ $a_2$
$b_1$ $b_2$
+
$c_1$ $c_2$
SIMD
width=2

- Single Instruction Multiple Data architectures make use of data parallelism
- We care about SIMD because of area and power efficiency concerns
  - Amortize control overhead over SIMD width
- Parallelism exposed to programmer & compiler

# A Brief History of x86 SIMD

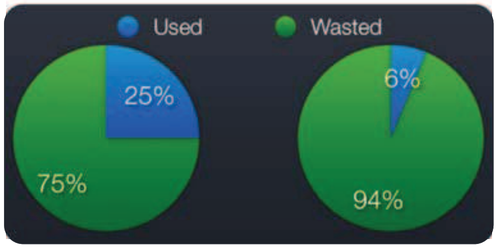

Subset
Future Subset
8 x 8 bit Integer
MMX
SSE
4 x 32 bit SP Float
SSE2
2 x 64 bit DP Float
SSE3
SSSE3
SSE4.1
SSE4.2
AVX
8 x 32 bit SP Float
AVX+FMA
3 operands
3dNow!
4 x 32 bit SP Float
SSE4.A
SSE5
??
Larrabee
16 x 32 bit SP Float

# SIMD: Neglected Parallelism

- It is difficult for a compiler to exploit SIMD
- How do you deal with sparse data & branches?
  - Many languages (like C) are difficult to vectorize
  - Fortran is somewhat better
- Most common solution:
  - Either forget about SIMD
    - » Pray the autovectorizer likes you
  - Or instantiate intrinsics (assembly language)
  - Requires a new code version for every SIMD extension

---

# What to do with SIMD?



4 way SIMD (SSE) 16 way SIMD (LRB)

- Neglecting SIMD is becoming more expensive
  - AVX: 8 way SIMD, Larrabee: 16 way SIMD, Nvidia: 32 way SIMD, ATI: 64 way SIMD
- This problem composes with thread level parallelism
- We need a programming model which addresses both problems

---

# Graphics Processing Units (GPUs)

- Original GPUs were dedicated fixed-function devices for generating 3D graphics (mid-late 1990s) including high-performance floating-point units
  - Provide workstation-like graphics for PCs
  - User could configure graphics pipeline, but not really program it
- Over time, more programmability added (2001-2005)
  - E.g., New language Cg for writing small programs run on each vertex or each pixel, also Windows DirectX variants
  - Massively parallel (millions of vertices or pixels per frame) but very constrained programming model
- Some users noticed they could do general-purpose computation by mapping input and output data to images, and computation to vertex and pixel shading computations
  - Incredibly difficult programming model as had to use graphics pipeline model for general computation

---

# General-Purpose GPUs (GP-GPUs)

- In 2006, Nvidia introduced GeForce 8800 GPU supporting a new programming language: CUDA
  - "Compute Unified Device Architecture"
  - Subsequently, broader industry pushing for OpenCL, a vendor-neutral version of same ideas.
- Idea: Take advantage of GPU computational performance and memory bandwidth to accelerate some kernels for general-purpose computing
- Attached processor model: Host CPU issues data-parallel kernels to GP-GPU for execution
- This lecture has a simplified version of Nvidia CUDA-style model and only considers GPU execution for computational kernels, not graphics
  - Would probably need another course to describe graphics processing

# Simplified CUDA Programming Model

- Computation performed by a very large number of independent small scalar threads (*CUDA threads* or *microthreads*) grouped into *thread blocks.*

```
// C version of DAXPY loop.
void daxpy(int n, double a, double*x, double*y)
{ for (int i=0; i<n; i++)
      y[i] = a*x[i] + y[i]; }

// CUDA version.
__host__   // Piece run on host processor.
int nblocks = (n+255)/256; // 256 CUDA threads/block
daxpy<<<nblocks,256>>>(n,2.0,x,y);

__device__   // Piece run on GP-GPU.
void daxpy(int n, double a, double*x, double*y)
{ int i = blockIdx.x*blockDim.x + threadId.x;
  if (i<n) y[i]=a*x[i]+y[i]; }
```

---

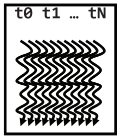
# Hierarchy of Concurrent Threads

- Parallel kernels composed of many threads
  - all threads execute the same sequential program

Thread t

- Threads are grouped into thread blocks
  - threads in the same block can cooperate

Block b



- Threads/blocks have unique IDs

---

# What is a CUDA Thread?

- Independent thread of execution
  - has its own PC, variables (registers), processor state, etc.
  - no implication about how threads are scheduled
- CUDA threads might be physical threads
  - as on NVIDIA GPUs
- CUDA threads might be virtual threads
  - might pick 1 block = 1 physical thread on multicore CPU

---

# What is a CUDA Thread Block?

- Thread block = virtualized multiprocessor
  - freely choose processors to fit data
  - freely customize for each kernel launch
- Thread block = a (data) parallel task
  - all blocks in kernel have the same entry point
  - but may execute any code they want
- Thread blocks of kernel must be independent tasks
  - program valid for *any interleaving* of block executions

# Mapping back

- Thread parallelism
  - each thread is an independent thread of execution
- Data parallelism
  - across threads in a block
  - across blocks in a kernel
- Task parallelism
  - different blocks are independent
  - independent kernels

---

# Synchronization

- Threads within a block may synchronize with barriers

```
… Step 1 …
__syncthreads();
… Step 2 …
```

- Blocks coordinate via atomic memory operations
  - e.g., increment shared queue pointer with `atomicInc()`
- Implicit barrier between dependent kernels

```
vec_minus<<<nblocks, blksize>>>(a, b, c);

vec_dot<<<nblocks, blksize>>>(c, c);
```

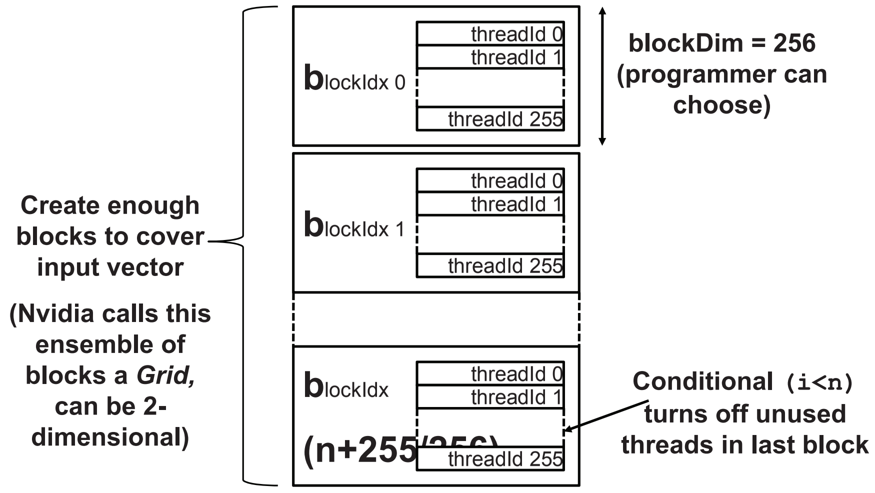
---

# Blocks must be independent

- Any possible interleaving of blocks should be valid
  - presumed to run to completion without pre-emption
  - can run in any order
  - can run concurrently OR sequentially
- Blocks may coordinate but not synchronize
  - shared queue pointer: OK
  - shared lock: BAD … can easily deadlock
- Independence requirement gives scalability

---

# Programmer's View of Execution


blockIdx 0
threadId 0
threadId 1
threadId 255
blockIdx 1
threadId 0
threadId 1
threadId 255
blockIdx (n+255/256)
threadId 0
threadId 1
threadId 255

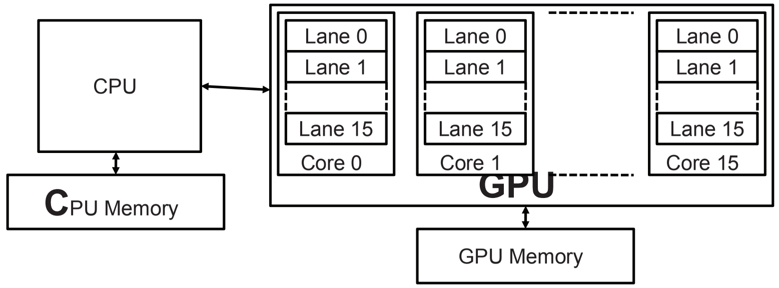
**blockDim = 256 (programmer can choose)**

**Create enough blocks to cover input vector**

**(Nvidia calls this ensemble of blocks a *Grid*, can be 2-dimensional)**

**Conditional `(i<n)` turns off unused threads in last block**

# Hardware Execution Model

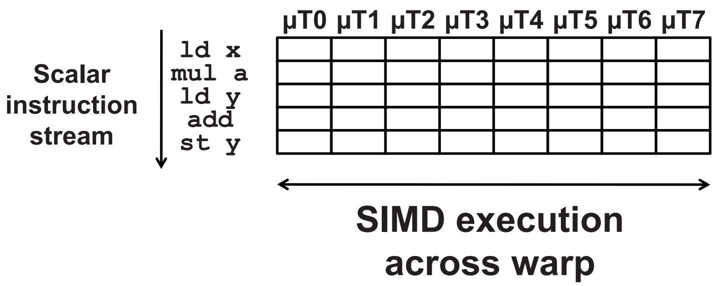

CPU
CPU Memory
Lane 0
Lane 1
Lane 15
Core 0
Lane 0
Lane 1
Lane 15
Core 1
Lane 0
Lane 1
Lane 15
Core 15
GPU
GPU Memory

- GPU is built from multiple parallel cores, each core contains a multithreaded SIMD processor with multiple lanes but with no scalar processor
- CPU sends whole “grid” over to GPU, which distributes thread blocks among cores (each thread block executes on one core)
  - Programmer unaware of number of cores

---

# “Single Instruction, Multiple Thread”

- GPUs use a SIMT model, where individual scalar instruction streams for each CUDA thread are grouped together for SIMD execution on hardware (Nvidia groups 32 CUDA threads into a *warp*)


Scalar instruction stream
µT0 µT1 µT2 µT3 µT4 µT5 µT6 µT7

```
ld x
mul a
ld y
add
st y
```

SIMD execution across warp

---

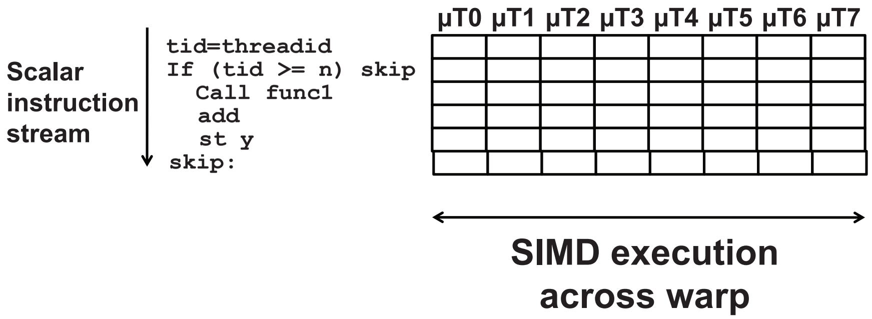
# Implications of SIMT Model

- All “vector” loads and stores are scatter-gather, as individual µthreads perform scalar loads and stores
  - GPU adds hardware to dynamically coalesce individual µthread loads and stores to mimic vector loads and stores
- Every µthread has to perform stripmining calculations redundantly (“am I active?”) as there is no scalar processor equivalent

---

# Conditionals in SIMT model

- Simple if-then-else are compiled into predicated execution, equivalent to vector masking
- More complex control flow compiled into branches
- How to execute a vector of branches?


Scalar instruction stream
µT0 µT1 µT2 µT3 µT4 µT5 µT6 µT7

```
tid=threadid
If (tid >= n) skip
  Call func1
  add
  st y
skip:
```

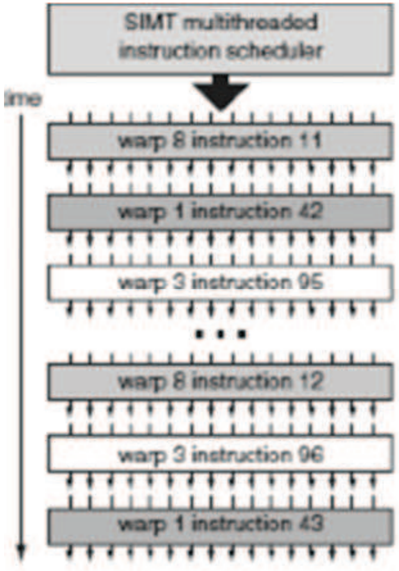
SIMD execution across warp

## Branch divergence

- Hardware tracks which µthreads take or don’t take branch
- If all go the same way, then keep going in SIMD fashion
- If not, create mask vector indicating taken/not-taken
- Keep executing not-taken path under mask, push taken branch PC+mask onto a hardware stack and execute later
- When can execution of µthreads in warp reconverge?

---

## Warps are multithreaded on core

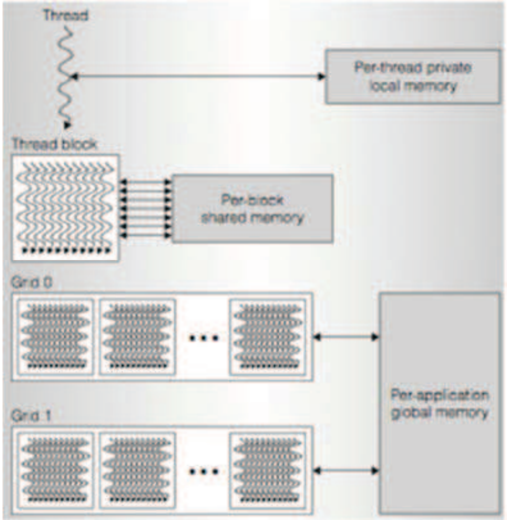

SIMT multithreaded instruction scheduler
time
warp 8 instruction 11
warp 1 instruction 42
warp 3 instruction 95
warp 8 instruction 12
warp 3 instruction 96
warp 1 instruction 43

- One warp of 32 µthreads is a single thread in the hardware
- Multiple warp threads are interleaved in execution on a single core to hide latencies (memory and functional unit)
- A single thread block can contain multiple warps (up to 512 µT max in CUDA), all mapped to single core
- Can have multiple blocks executing on one core

[Nvidia, 2010]

---

## GPU Memory Hierarchy


Thread
Per-thread private local memory
Thread block
Per-block shared memory
Grid 0
Grid 1
Per-application global memory

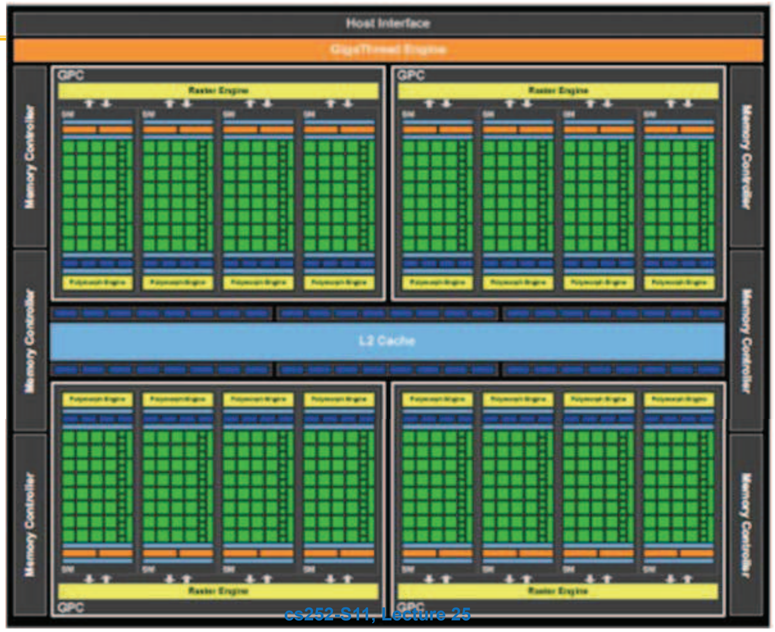
[ Nvidia, 2010]

---

## SIMT

- Illusion of many independent threads
- But for efficiency, programmer must try and keep µthreads aligned in a SIMD fashion
  - Try and do unit-stride loads and store so memory coalescing kicks in
  - Avoid branch divergence so most instruction slots execute useful work and are not masked off

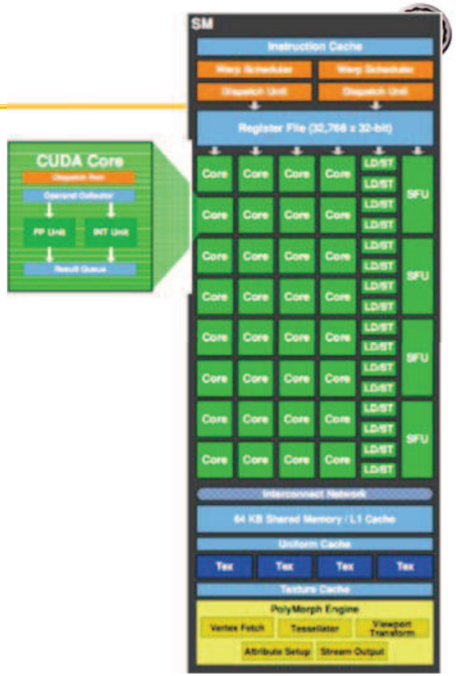
# Nvidia Fermi GF100 GPU


Host Interface
GigaThread Engine
GPC
Raster Engine
Memory Controller
L2 Cache
Polymorph Engine
SM

[Nvidia, 2010]

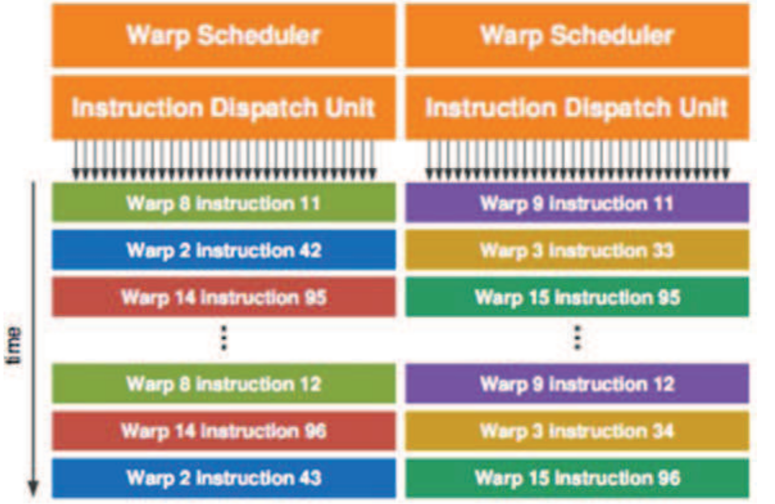
---

# Fermi "Streaming Multiprocessor" Core


SM
Instruction Cache
Warp Scheduler
Warp Scheduler
Dispatch Unit
Dispatch Unit
Register File (32,768 x 32-bit)
CUDA Core
Dispatch Port
Operand Collector
FP Unit
INT Unit
Result Queue
Core
LD/ST
SFU
Interconnect Network
64 KB Shared Memory / L1 Cache
Uniform Cache
Tex
Tex
Tex
Tex
Texture Cache
PolyMorph Engine
Vertex Fetch
Tessellator
Viewport Transform
Attribute Setup
Stream Output

---

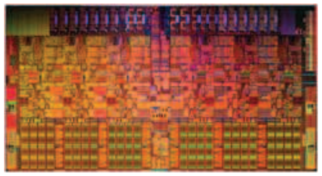
# Fermi Dual-Issue Warp Scheduler


Warp Scheduler
Warp Scheduler
Instruction Dispatch Unit
Instruction Dispatch Unit
Warp 8 instruction 11
Warp 9 instruction 11
Warp 2 instruction 42
Warp 3 instruction 33
Warp 14 instruction 95
Warp 15 instruction 95
Warp 8 instruction 12
Warp 9 instruction 12
Warp 14 instruction 96
Warp 3 instruction 34
Warp 2 instruction 43
Warp 15 instruction 96
time

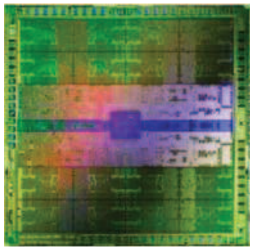
---

# Multicore & Manycore, *Comparison*

| Specifications | Westmere-EP | Fermi (GF110) |
|---|---|---|
| Processing Elements | 6 cores, 2 issue, 4 way SIMD @3.46 GHz | 16 cores, 2 issue, 16 way SIMD @1.54 GHz |
| Resident Strands/Threads (max) | 6 cores, 2 threads, 4 way SIMD:<br>48 strands | 16 cores, 48 SIMD vectors, 32 way SIMD:<br>24576 threads |
| SP GFLOP/s | 166 | 1577 |
| Memory Bandwidth | 32 GB/s | 192 GB/s |
| Register File | 6 kB (?) | 2 MB |
| Local Store/L1 Cache | 192 kB | 1024 kB |
| L2 Cache | 1536 kB | 0.75 MB |
| L3 Cache | 12 MB | - |



Westmere-EP (32nm)



Fermi (40nm)

# Conclusion: GPU Future

- GPU: A type of Vector Processor originally optimized for graphics processing
  - Has become general purpose with introduction of CUDA
  - Memory system extracts unit stride behavior dynamically
  - "CUDA Threads" grouped into "WARPS" automatically (32 threads)
  - "Thread-Blocks" (with up to 512 CUDA Threads) dynamically assigned to processors (since number of processors/system varies)
- High-end desktops have separate GPU chip, but trend towards integrating GPU on same die as CPU
  - Advantage is shared memory with CPU, no need to transfer data
  - Disadvantage is reduced memory bandwidth compared to dedicated smaller-capacity specialized memory system
    - » Graphics DRAM (GDDR) versus regular DRAM (DDR3)
- Will GP-GPU survive? Or will improvements in CPU DLP make GP-GPU redundant?
  - On same die, CPU and GPU should have same memory bandwidth
  - GPU might have more FLOPS as needed for graphics anyway