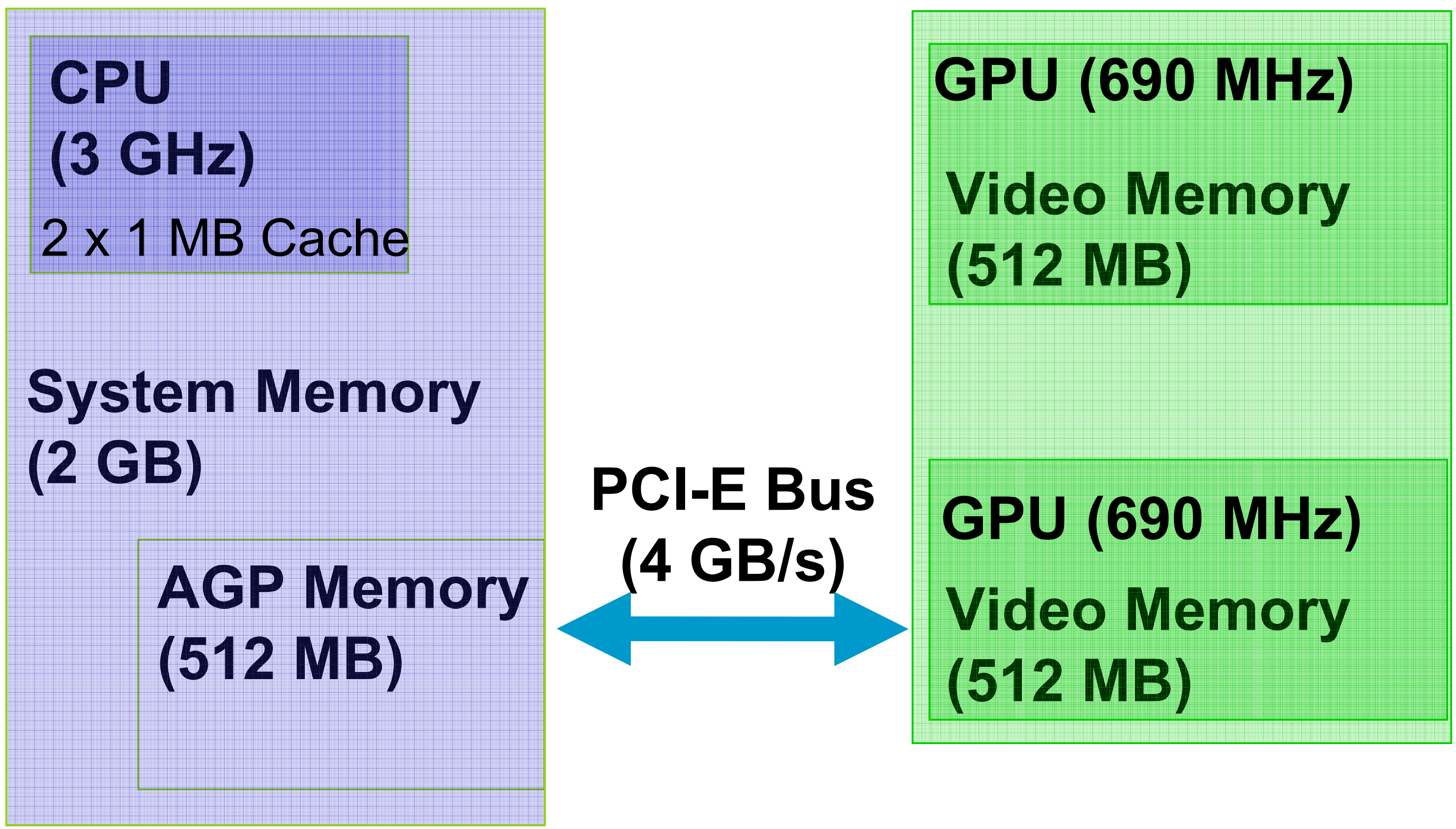
Query Processing on GPUs

- Graphics Processor Overview
- Mapping Computation to GPUs
- Database and data mining applications
  - Database queries
  - Quantile and frequency queries
  - External memory sorting
  - Scientific computations
- Summary

CPU vs. GPU


CPU
(3 GHz)
2 x 1 MB Cache
System Memory
(2 GB)
AGP Memory
(512 MB)
PCI-E Bus
(4 GB/s)
GPU (690 MHz)
Video Memory
(512 MB)
GPU (690 MHz)
Video Memory
(512 MB)

Query Processing on CPUs

- Slow random memory accesses
  - Small CPU caches (< 2MB)
  - Random memory accesses slower than even sequential disk accesses
- High memory latency
  - Huge memory to compute gap!
- CPUs are deeply pipelined
  - Pentium 4 has 30 pipeline stages
  - Do not hide latency - high cycles per instruction (CPI)
- CPU is under-utilized for data intensive applications

Graphics Processing Units (GPUs)

- *Commodity processor for graphics applications*
- Massively parallel vector processors
- High memory bandwidth
  - Low memory latency pipeline
  - Programmable
- High growth rate
- Power-efficient

GPU: Commodity Processor

Cell phones

Laptops

Consoles

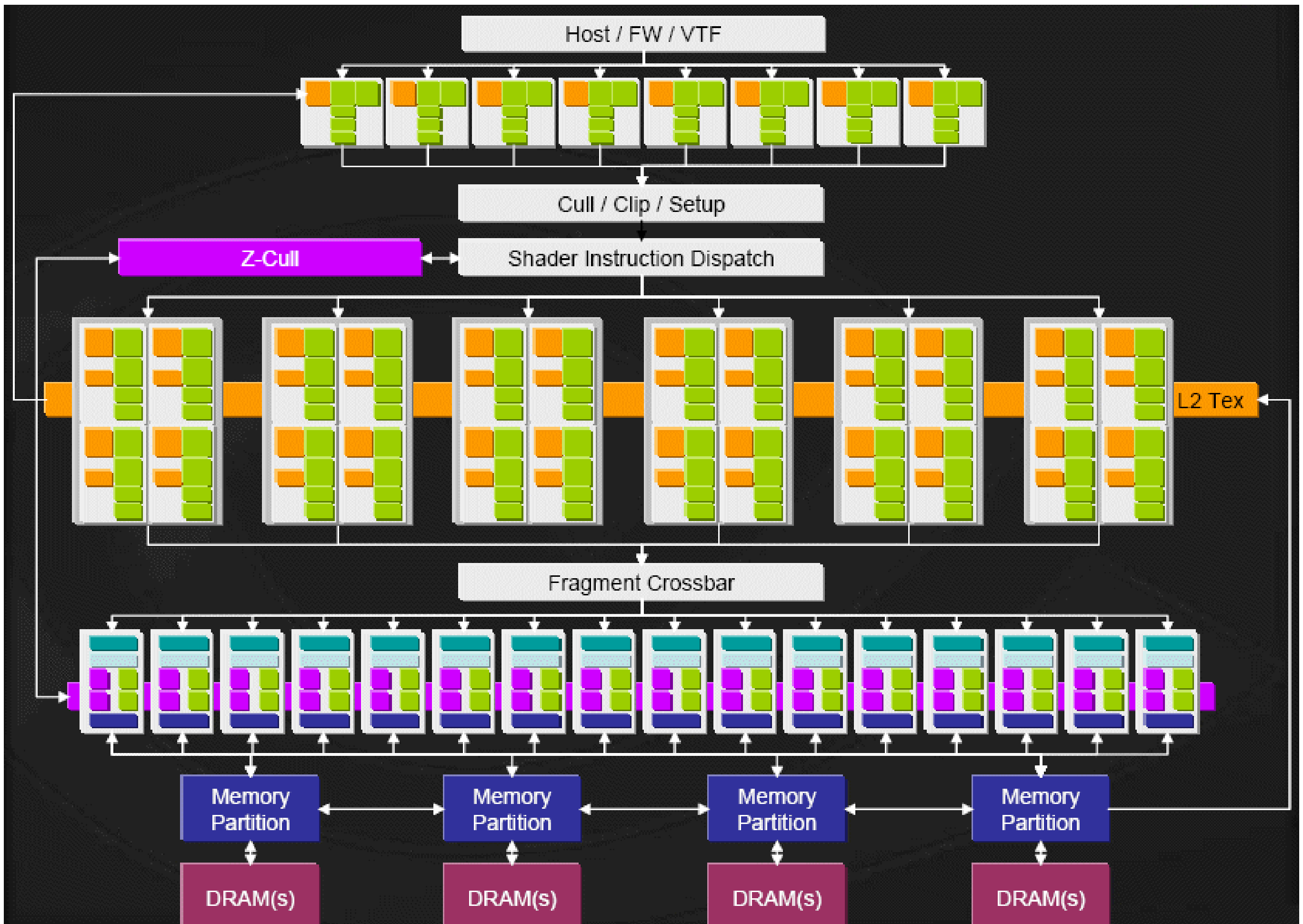
Desktops

PSP

Graphics Processing Units (GPUs)

- Commodity processor for graphics applications
- *Massively parallel vector processors*
  - ***10x*** *more operations per sec than CPUs*
- High memory bandwidth
  - Low memory latency pipeline
  - Programmable
- High growth rate
- Power-efficient

Parallelism on GPUs



Graphics FLOPS

GPU – 1.3 TFLOPS

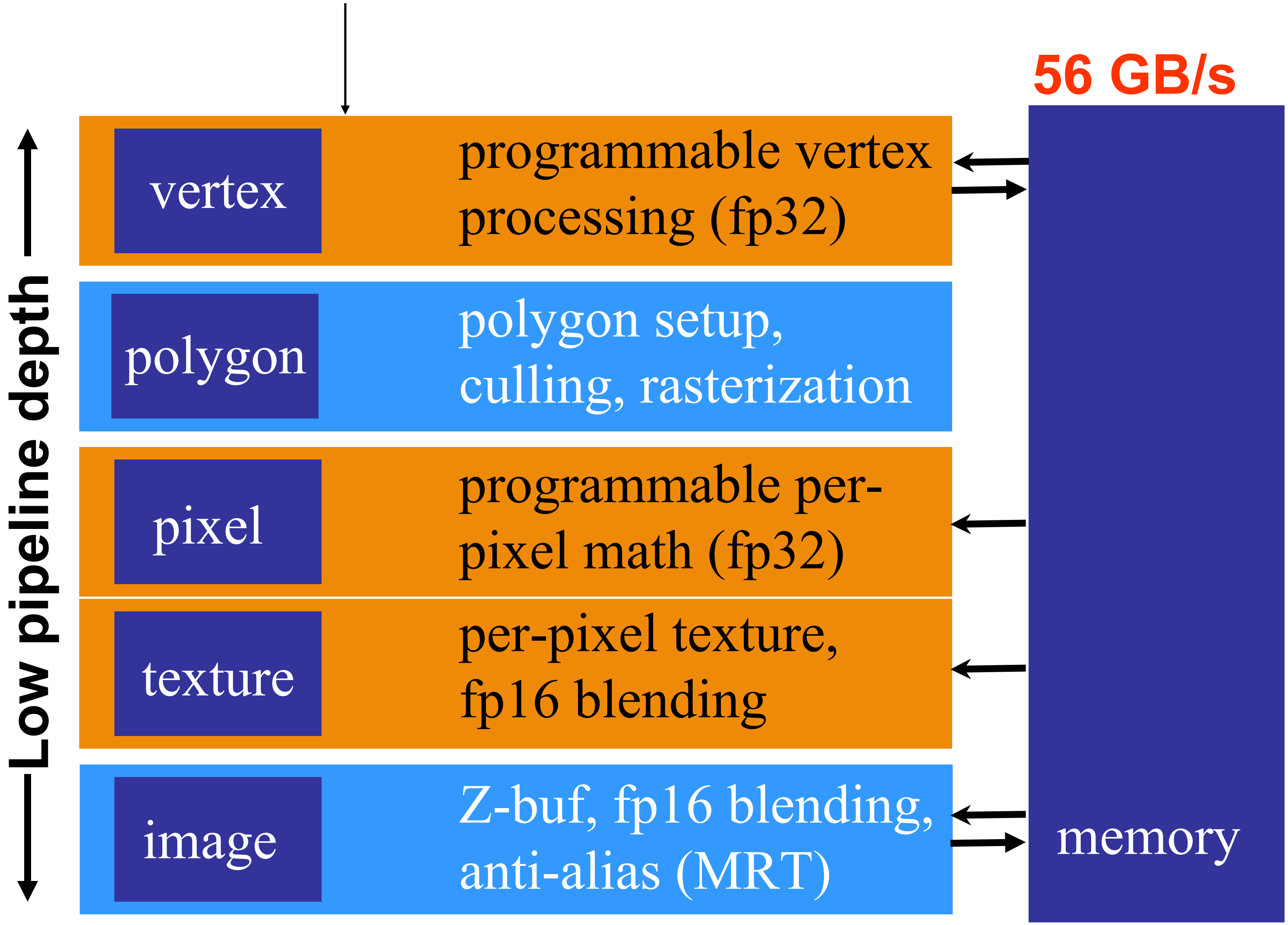
CPU – 25.6 GFLOPS

Graphics Processing Units (GPUs)

- Commodity processor for graphics applications
- Massively parallel vector processors
- *High memory bandwidth*
  - *Low memory latency pipeline*
  - *Programmable*
  - ***10x** more memory bandwidth than CPUs*
- High growth rate
- Power-efficient

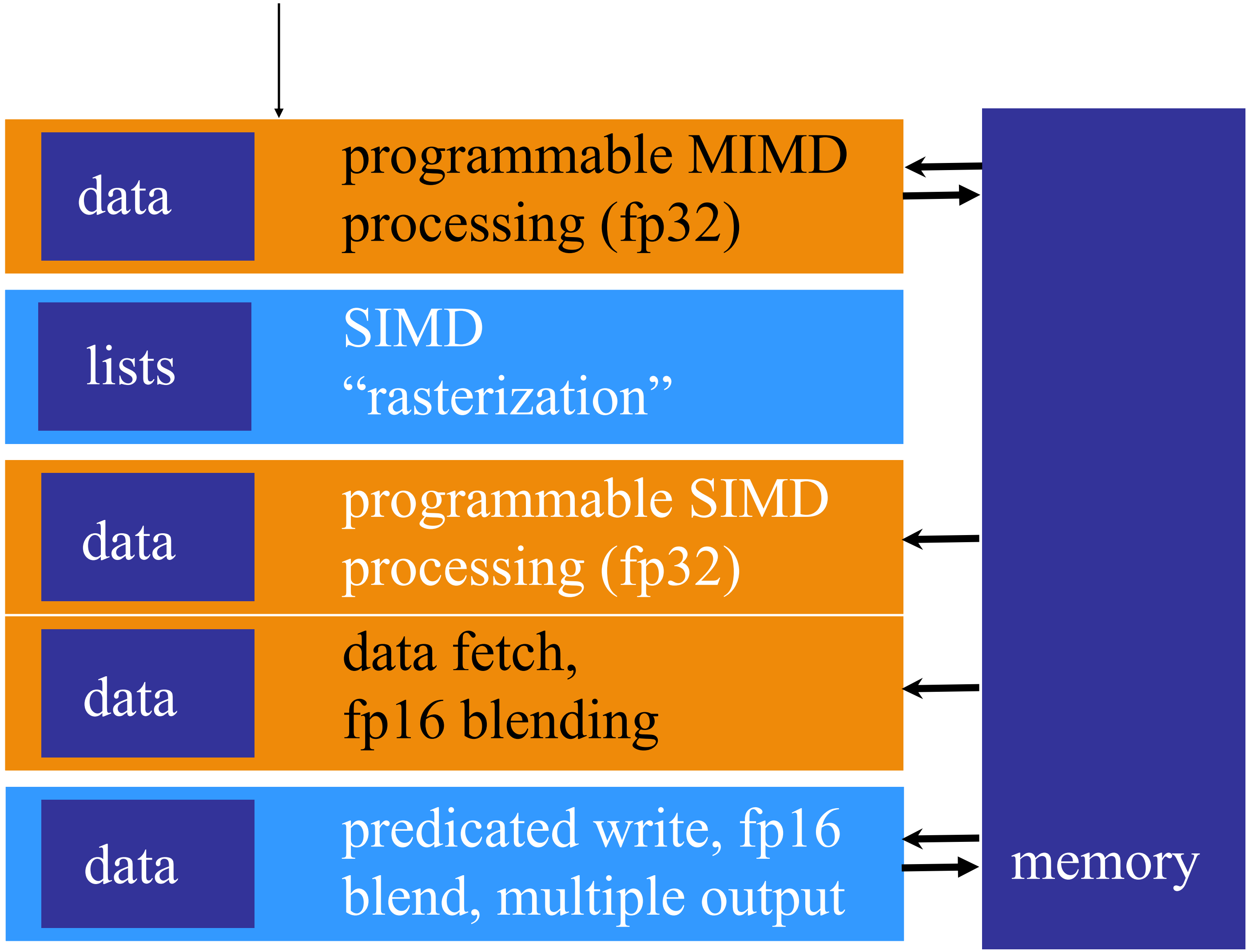
Graphics Pipeline

Hides memory latency!!



NON-Graphics Pipeline Abstraction

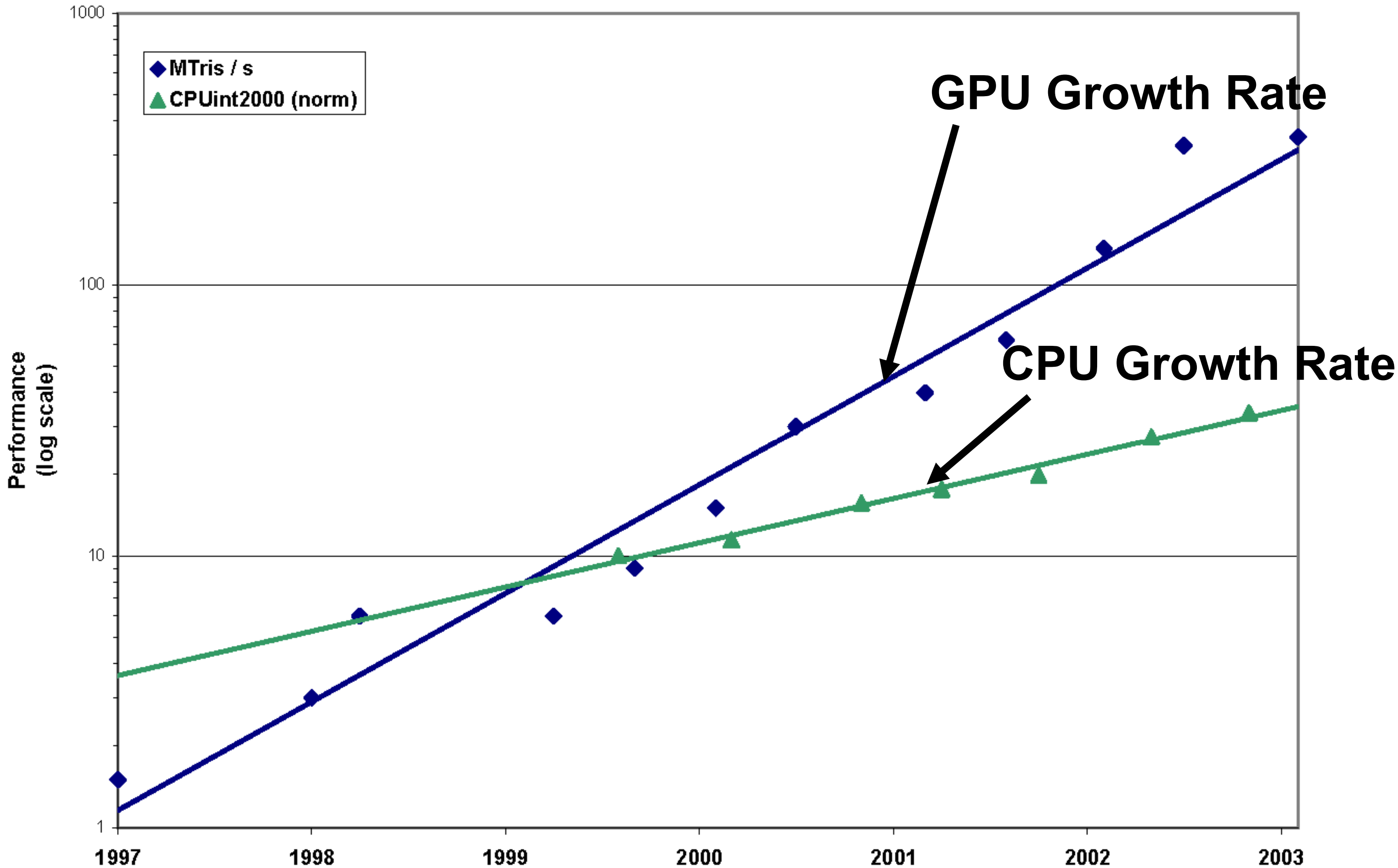
Courtesy: David Kirk, Chief Scientist, NVIDIA



Graphics Processing Units (GPUs)

- Commodity processor for graphics applications
- Massively parallel vector processors
- High memory bandwidth
  - Low memory latency pipeline
  - Programmable
- *High growth rate*
- Power-efficient

Exploiting Technology Moving Faster than Moore's Law

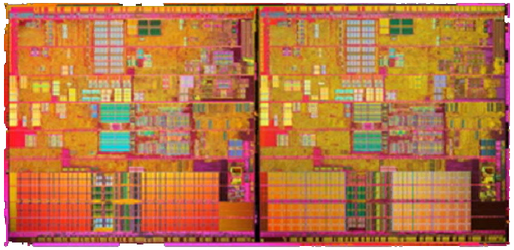


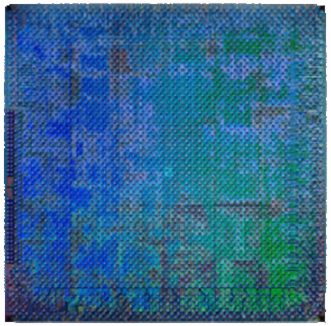
Graphics Processing Units (GPUs)

- Commodity processor for graphics applications
- Massively parallel vector processors
- High memory bandwidth
  - Low memory latency pipeline
  - Programmable
- High growth rate
- *Power-efficient*

CPU vs. GPU

(Henry Moreton: NVIDIA, Aug. 2005)





	PEE 840	7800GTX	GPU/CPU
Graphics GFLOPs	25.6	1300	50.8
Power (W)	130	65	0.5
GFLOPS/W	0.2	20.0	**101.6**

Outline

- Graphics Processor Overview
- Mapping Computation to GPUs
- Database and data mining applications
  - Database queries
  - Quantile and frequency queries
  - External memory sorting
  - Scientific computations
- Summary

The Importance of Data Parallelism

- GPUs are designed for graphics
  - Highly parallel tasks
- GPUs process *independent* vertices & fragments
  - Temporary registers are zeroed
  - No shared or static data
  - No read-modify-write buffers
- Data-parallel processing
  - GPUs architecture is ALU-heavy
    - **Multiple vertex & pixel pipelines, multiple ALUs per pipe**
  - Hide memory latency (with more computation)

Arithmetic Intensity

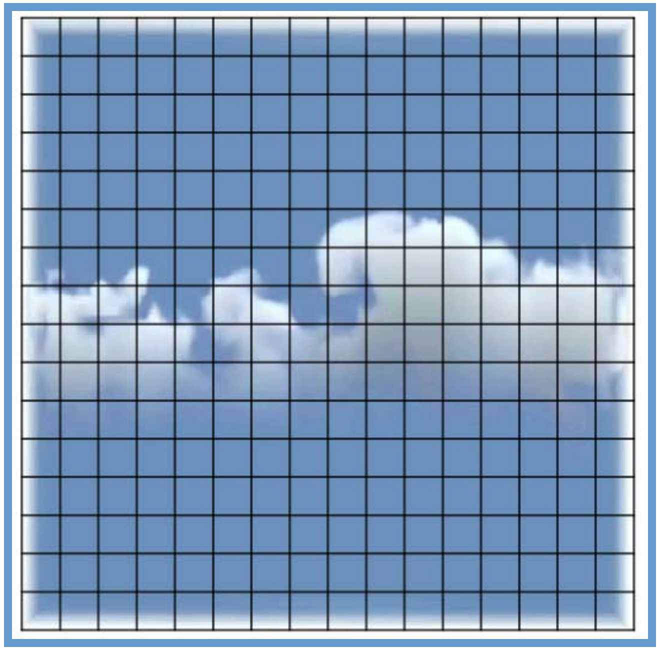
- Arithmetic intensity
  - ops per word transferred
  - Computation / bandwidth
- Best to have *high* arithmetic intensity
- Ideal GPGPU apps have
  - Large data sets
  - High parallelism
  - Minimal dependencies between data elements

Data Streams & Kernels

- Streams
  - Collection of records requiring similar computation
    - **Vertex positions, Voxels, FEM cells, etc.**
  - Provide data parallelism
- Kernels
  - Functions applied to each element in stream
    - **transforms, PDE, …**
  - Few dependencies between stream elements
    - **Encourage high Arithmetic Intensity**

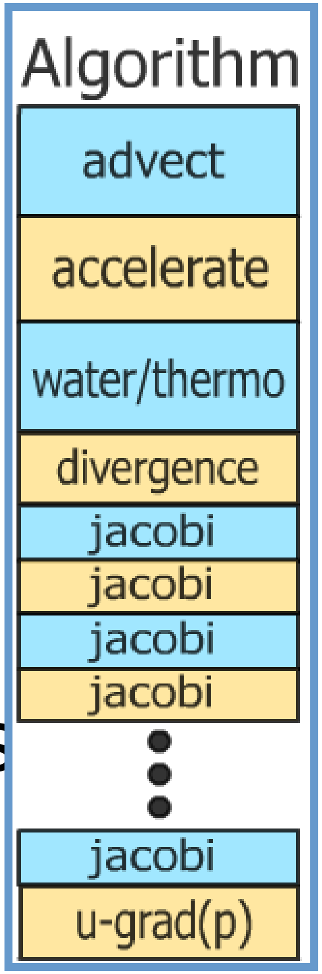
Example: Simulation Grid

- Common GPGPU computation style
  - Textures represent computational grids = streams
- Many computations map to grids
  - Matrix algebra
  - Image & Volume processing
  - Physically-based simulation
- Non-grid streams can be mapped to grids



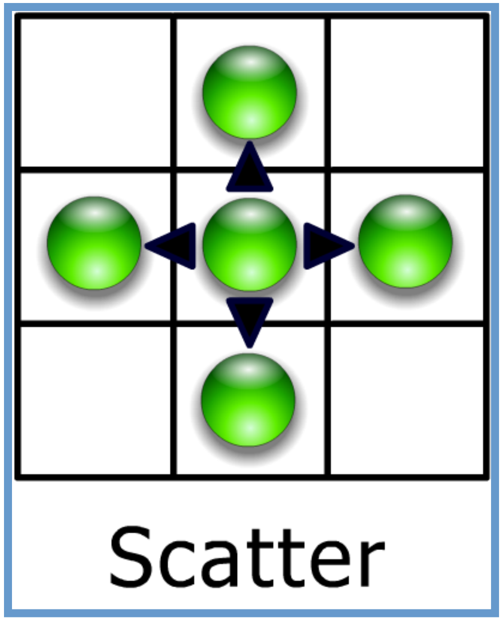
Stream Computation

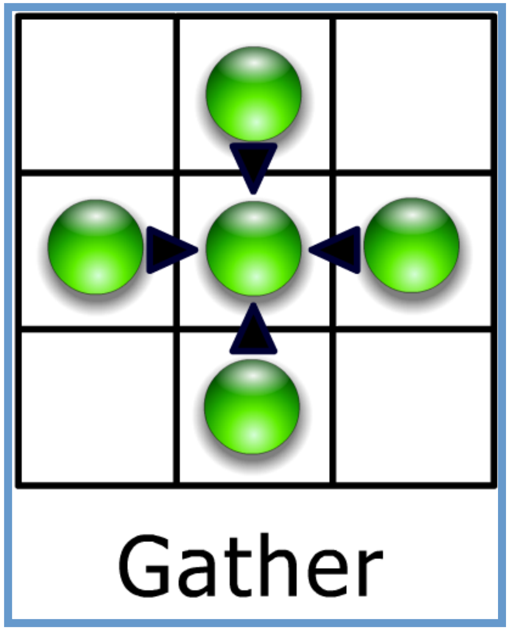
- Grid Simulation algorithm
  - Made up of steps
  - Each step updates entire grid
  - Must complete before next step can begin
- Grid is a stream, steps are kernels
  - Kernel applied to each stream element



Scatter vs. Gather

- Grid communication
  - Grid cells share information





Computational Resources Inventory

- Programmable parallel processors
  - Vertex & Fragment pipelines
- Rasterizer
  - Mostly useful for interpolating addresses (texture coordinates) and per-vertex constants
- Texture unit
  - Read-only memory interface
- Render to texture
  - Write-only memory interface

Vertex Processor

- Fully programmable (SIMD / MIMD)
- Processes 4-vectors (RGBA / XYZW)
- Capable of scatter but not gather
  - Can change the location of current vertex
  - Cannot read info from other vertices
  - Can only read a small constant memory
- Latest GPUs: Vertex Texture Fetch
  - Random access memory for vertices
  - ≈Gather (But not from the vertex stream itself)

Fragment Processor

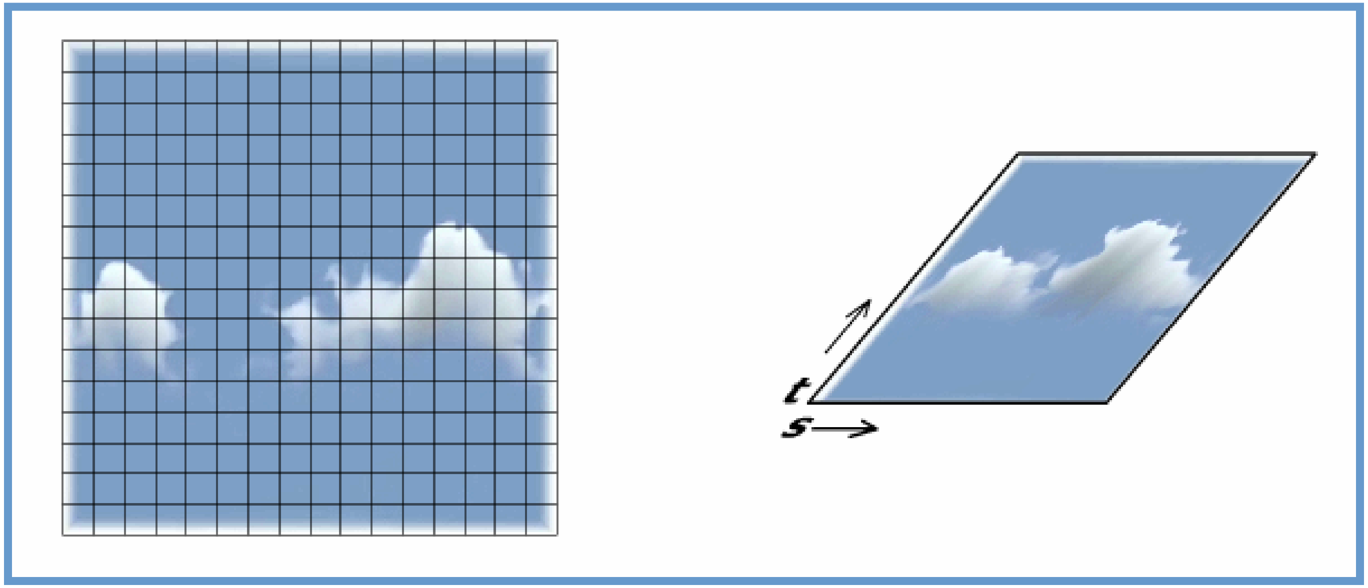
- Fully programmable (SIMD)
- Processes 4-component vectors (RGBA / XYZW)
- Random access memory read (textures)
- Capable of gather but not scatter
  - RAM read (texture fetch), but no RAM write
  - Output address fixed to a specific pixel
- Typically more useful than vertex processor
  - More fragment pipelines than vertex pipelines
  - Direct output (fragment processor is at end of pipeline)

CPU-GPU Analogies

- CPU programming is familiar
  - GPU programming is graphics-centric
- Analogies can aid understanding

CPU-GPU Analogies

CPU GPU



Stream / Data Array = Texture

Memory Read = Texture Sample

Kernels

advect

```cpp
for (int j = 1; j < height - 1; ++j)
{
  for (int i = 1; i < width - 1; ++i)
  {
    // get velocity at this cell
    Vec2f v = grid(x, y);

    // trace backwards along velocity field
    float x = (i - (v.x * timestep / dx));
    float y = (j - (v.y * timestep / dy));

    grid(x,y) = grid.bilerp(x, y);
  }
}
```

C++

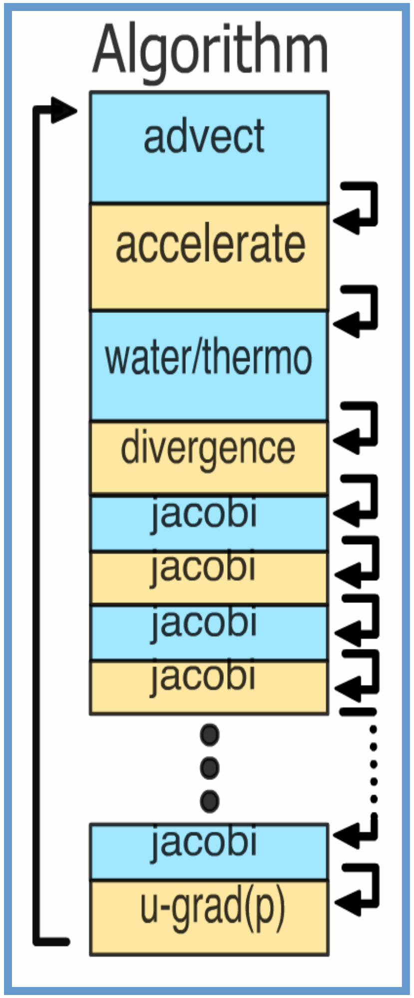
```
void advect(float2       uv   : WPOS,
        out float4       xNew : COLOR,

    uniform float        dt,  // timestep
    uniform float        dx,  // grid scale
    uniform samplerRECT  u,   // velocity
    uniform samplerRECT  x)   // state
{
  // trace backwards along velocity field
  float2 pos = ub - dt * f2texRECT(u, uv) / dx;

  xNew = f4texRECTbilerp(x, pos);
}
```

Cg

Kernel / loop body / algorithm step = Fragment Program

Feedback

- Each algorithm step depends on the results of previous steps
- Each time step depends on the results of the previous time step



Feedback

CPU

.

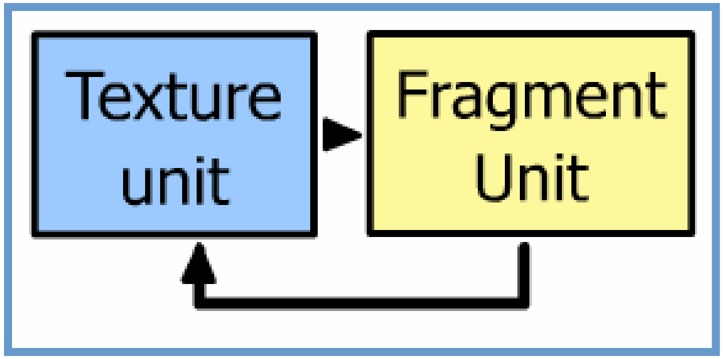
`Grid[i][j]= x;`

.

.

.

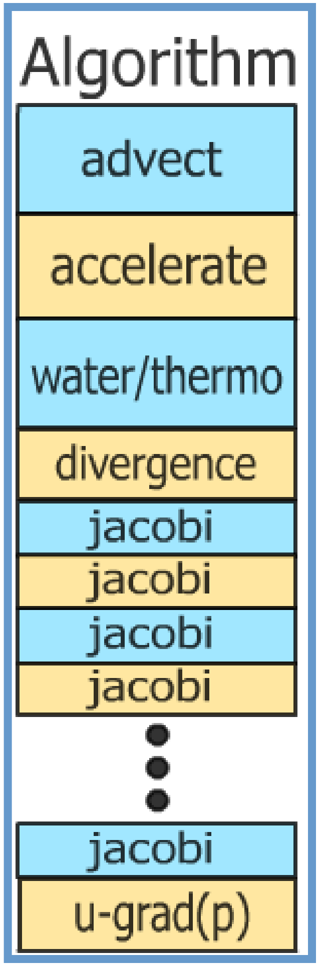
GPU



Array Write = Render to Texture

GPU Simulation Overview

- Analogies lead to implementation
  - Algorithm steps are fragment programs
    - **Computational *kernels***
  - Current state is stored in textures
  - Feedback via render to texture
- One question: how do we invoke computation?



Invoking Computation

- Must invoke computation at each pixel
  - Just draw geometry!
  - Most common GPGPU invocation is a full-screen quad
- Other Useful Analogies
  - Rasterization = Kernel Invocation
  - Texture Coordinates = Computational Domain
  - Vertex Coordinates = Computational Range

Typical "Grid" Computation

- Initialize "view" (so that pixels:texels::1:1)

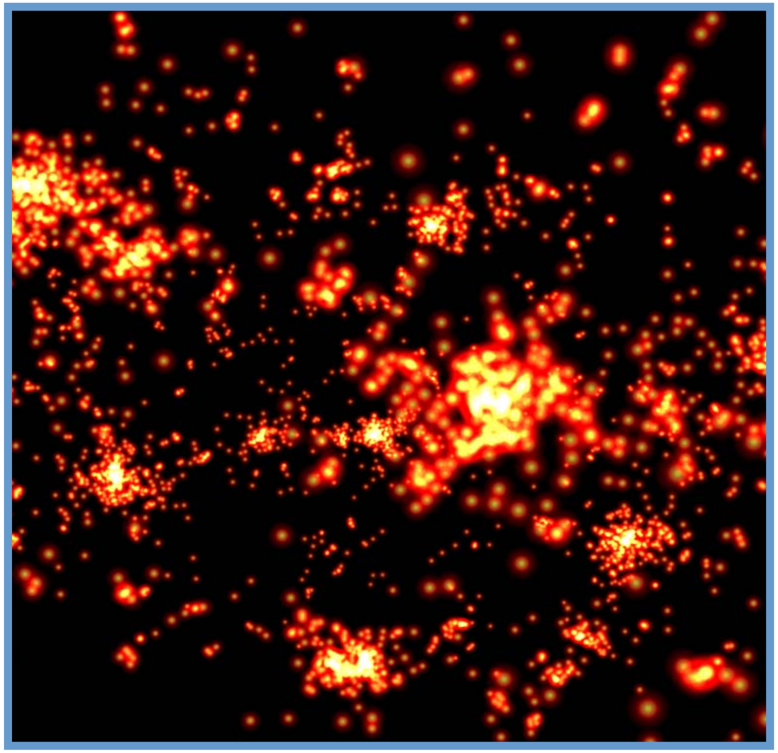
```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(0, 1, 0, 1, 0, 1);
glViewport(0, 0, outTexResX, outTexResY);
```

- For each algorithm step:
  - Activate render-to-texture
  - Setup input textures, fragment program
  - Draw a full-screen quad (1x1)

Example: N-Body Simulation

- Brute force ☹
- N = 8192 bodies
- N^2 gravity computations

- 64M force comps. / frame
- ~25 flops per force
- 10.5 fps
- 17+ GFLOPs sustained
  - GeForce 7800 GTX

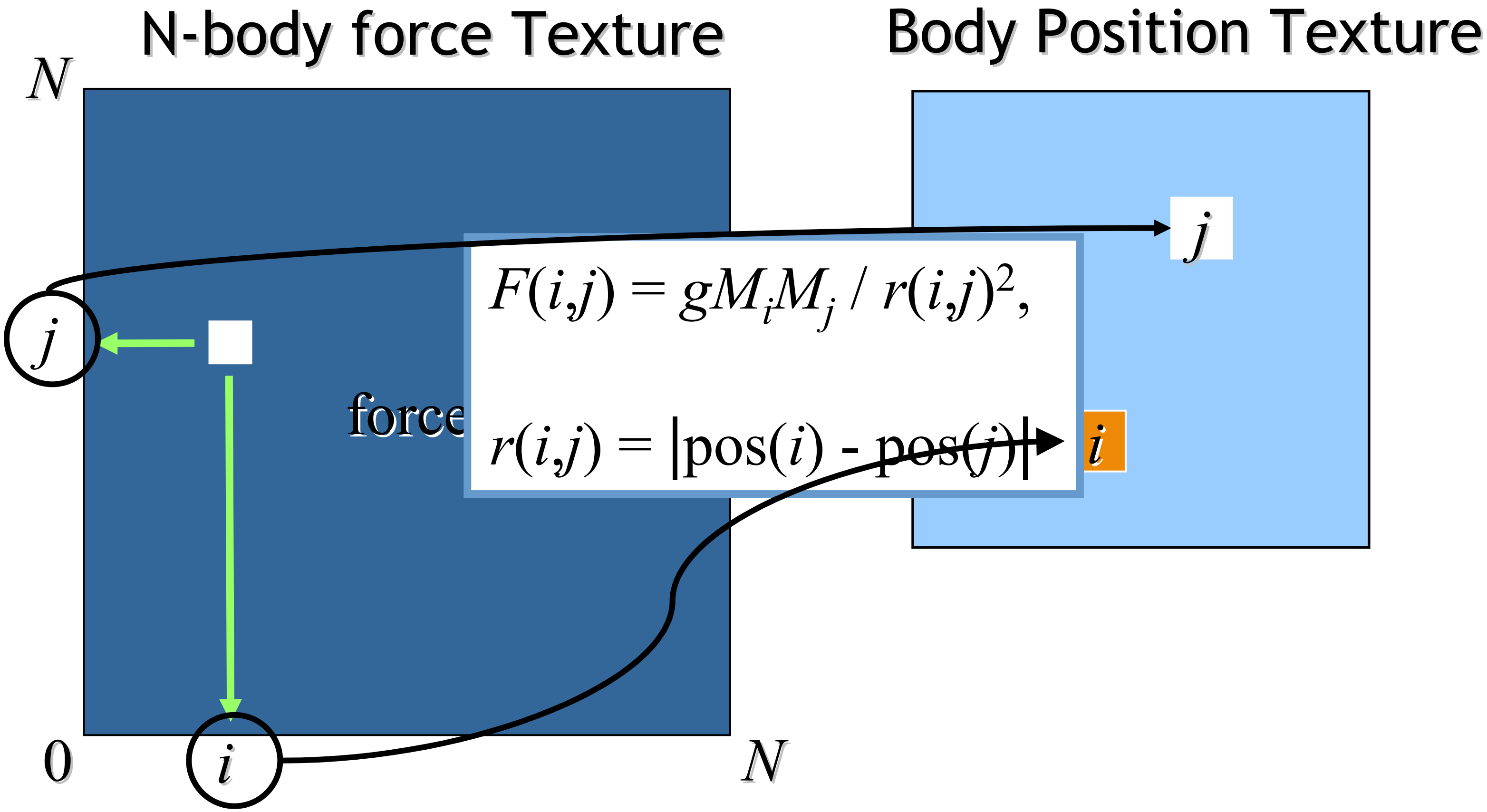


Nyland, Harris, Prins, GP² 2004 poster

Computing Gravitational Forces

- Each body attracts all other bodies
  - N bodies, so N^2 forces
- Draw into an NxN buffer
  - Pixel (i,j) computes force between bodies i and j
  - Very simple fragment program

Computing Gravitational Forces



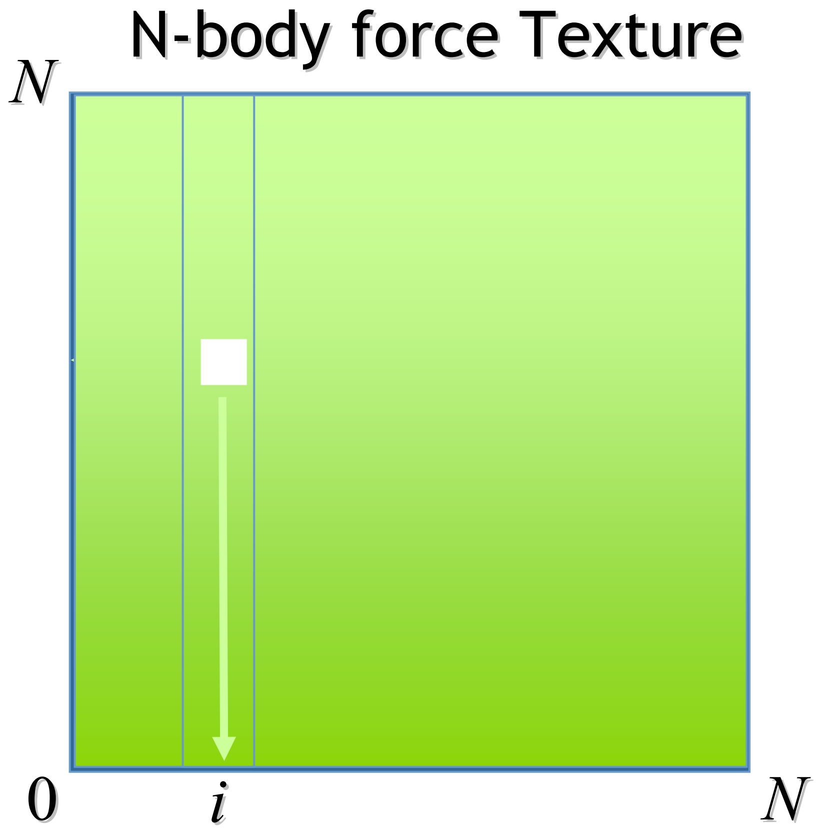
Force is proportional to the inverse square of the distance between bodies

Computing Gravitational Forces

```
float4 force(float2 ij       : WPOS,
     uniform sampler2D pos) : COLOR0
{
   // Pos texture is 2D, not 1D, so we need to
   // convert body index into 2D coords for pos tex
   float4 iCoords = getBodyCoords(ij);
   float4 iPosMass = texture2D(pos, iCoords.xy);
   float4 jPosMass = texture2D(pos, iCoords.zw);
   float3 dir = iPos.xyz - jPos.xyz;
   float r2 = dot(dir, dir);
  dir = normalize(dir);
   return dir * g * iPosMass.w * jPosMass.w / r2;
}
```

Computing Total Force

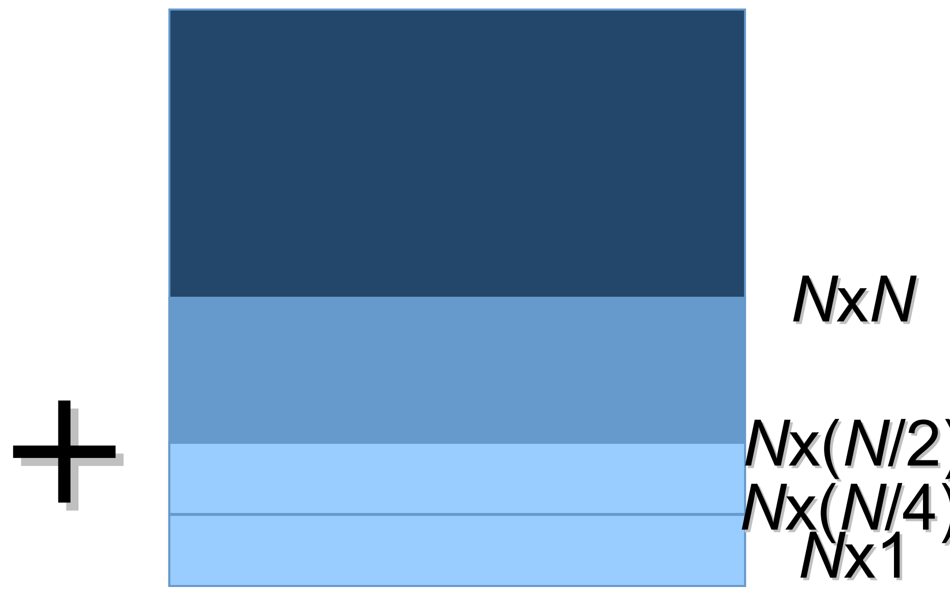
- Have: array of (i,j) forces
- Need: total force on each particle i
  - Sum of each column of the force array
- Can do all N columns in parallel



This is called a *Parallel Reduction*

Parallel Reductions

- 1D parallel reduction:
  - sum N columns or rows in parallel
  - add two halves of texture together
  - repeatedly...
  - Until we're left with a single row of texels



Requires $\log_2 N$ steps

Update Positions and Velocities

- Now we have a 1-D array of total forces
  - One per body
- Update Velocity
  - $\boldsymbol{u}(i,t+dt) = \boldsymbol{u}(i,t) + \boldsymbol{F}_{total}(i) * dt$
  - Simple pixel shader reads previous velocity and force textures, creates new velocity texture
- Update Position
  - $x(i, t+dt) = \boldsymbol{x}(i,t) + \boldsymbol{u}(i,t) * dt$
  - Simple pixel shader reads previous position and velocity textures, creates new position texture

Summary

- Presented mappings of basic computational concepts to GPUs
  - Basic concepts and terminology
  - For introductory "Hello GPGPU" sample code, see http://www.gpgpu.org/developer
- Only the beginning:
  - Rest of course presents advanced techniques, strategies, and specific algorithms.

Outline

- Graphics Processor Overview
- Mapping Computation to GPUs
- Applications
  - Database queries
  - Quantile and frequency queries
  - External memory sorting
  - Scientific computations
- Summary

N. Govindaraju, B. Lloyd, W. Wang, M. Lin and D. Manocha ,
Proc. of ACM SIGMOD 04

Basic DB Operations

Basic SQL query

Select A

From T

Where C

A= attributes or aggregations (SUM, COUNT, MAX etc)

T=relational table

C= Boolean Combination of Predicates (using operators AND, OR, NOT)

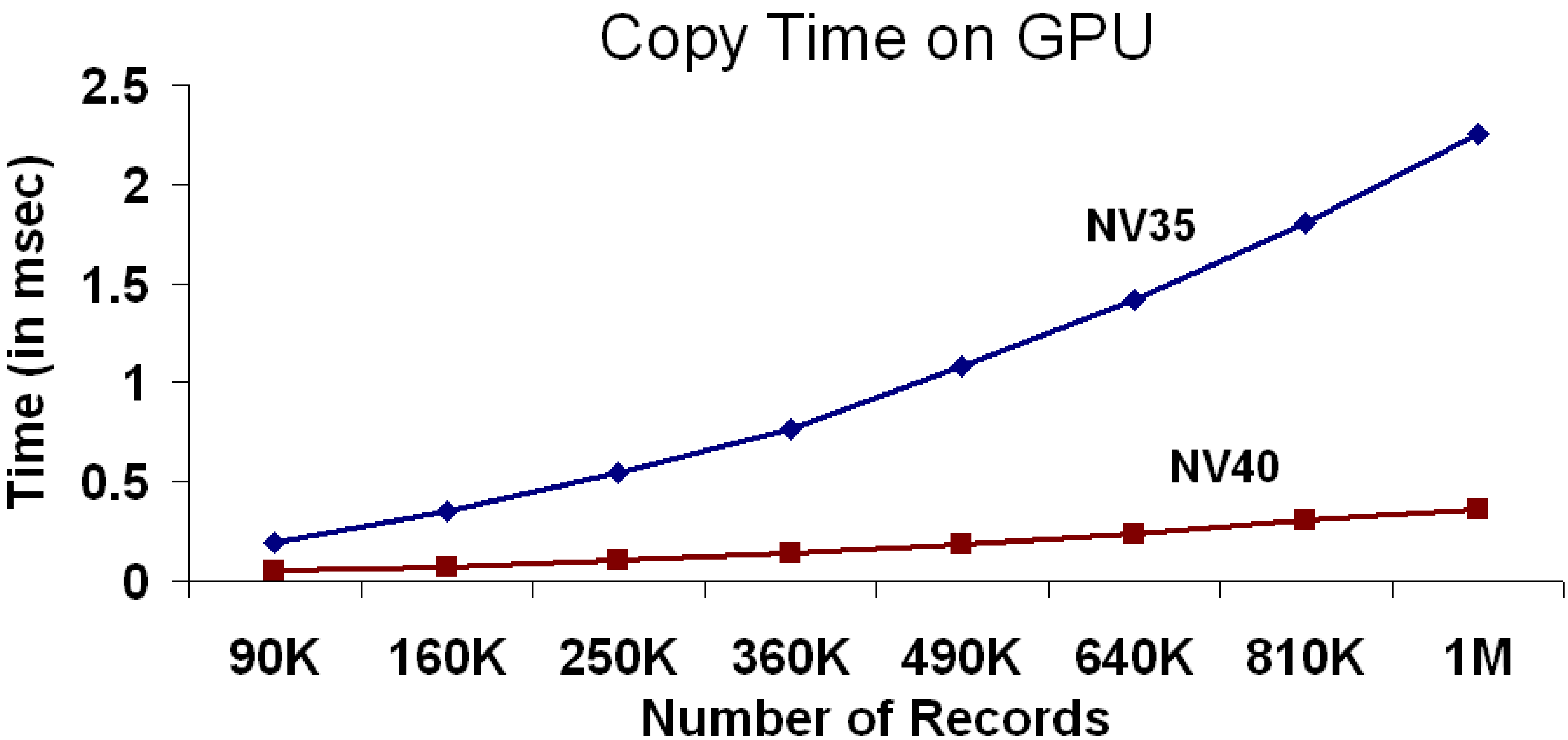
Database Operations

- Predicates
  - a_i op constant or a_i op a_j
  - op: <,>,<=,>=,!=, =, TRUE, FALSE
- Boolean combinations
  - Conjunctive Normal Form (CNF)
- Aggregations
  - COUNT, SUM, MAX, MEDIAN, AVG

Data Representation

- Attribute values a_i are stored in 2D textures on the GPU
- A fragment program is used to copy attributes to the depth buffer

Copy Time to the Depth Buffer



Predicate Evaluation

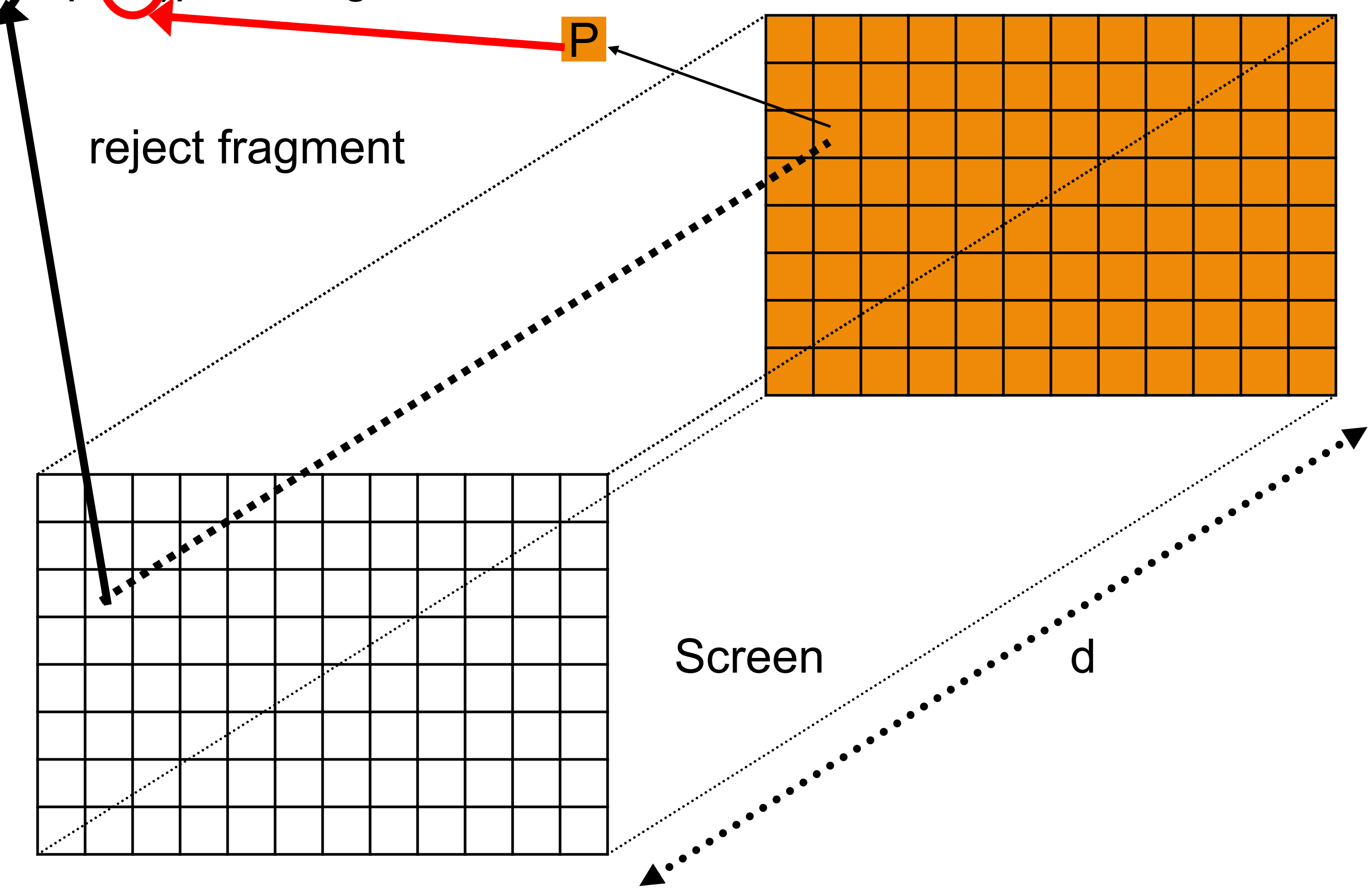
- a_i op constant (d)
  - Copy the attribute values a_i into depth buffer
  - Specify the comparison operation used in the depth test
  - Draw a screen filling quad at depth d and perform the depth test

a_i op d

If (a_i op d) pass fragment

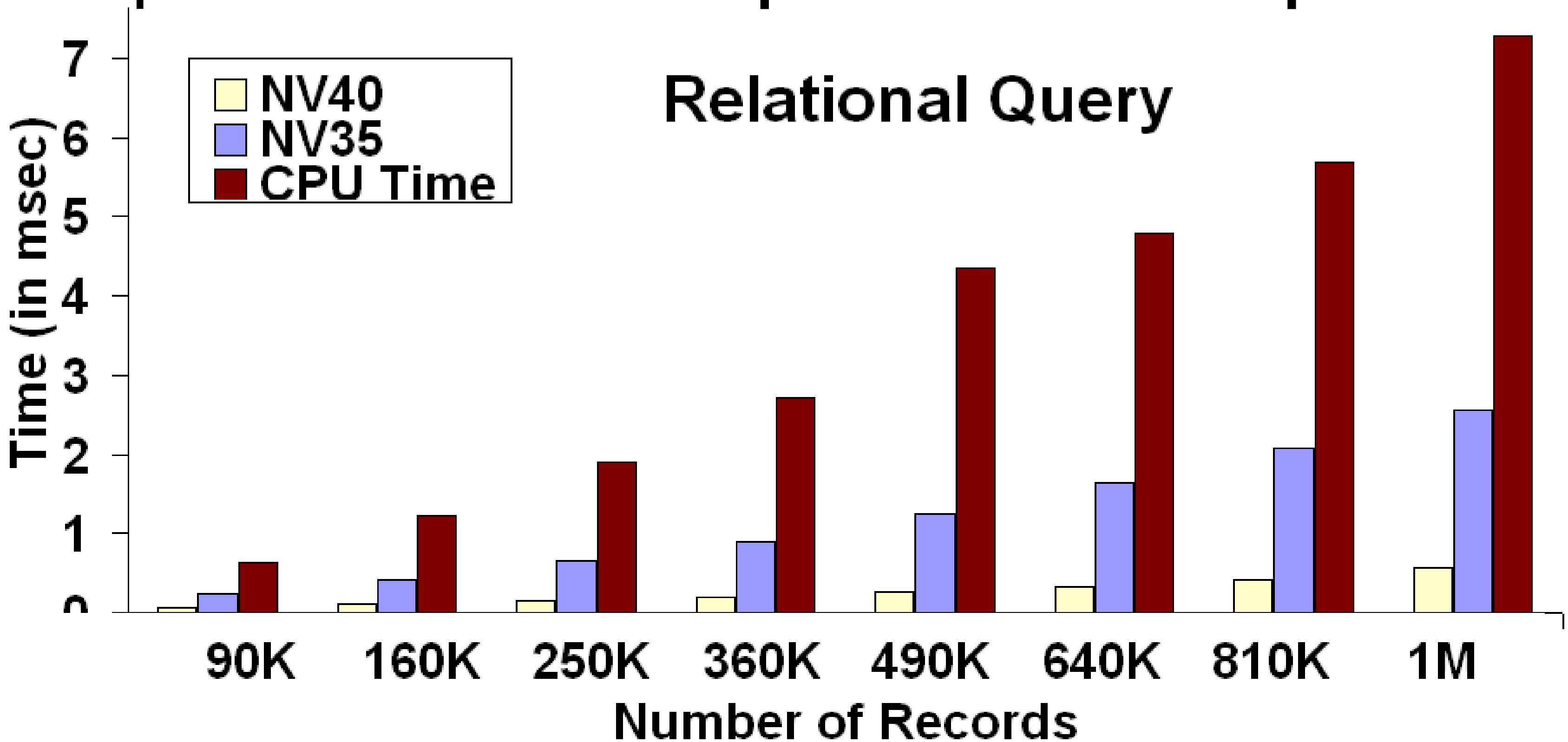
Else

reject fragment



Predicate Evaluation

CPU implementation — Intel compiler 7.1 with SIMD optimizations



GPU is nearly 20 times faster than 2.8 GHz Xeon

Predicate Evaluation

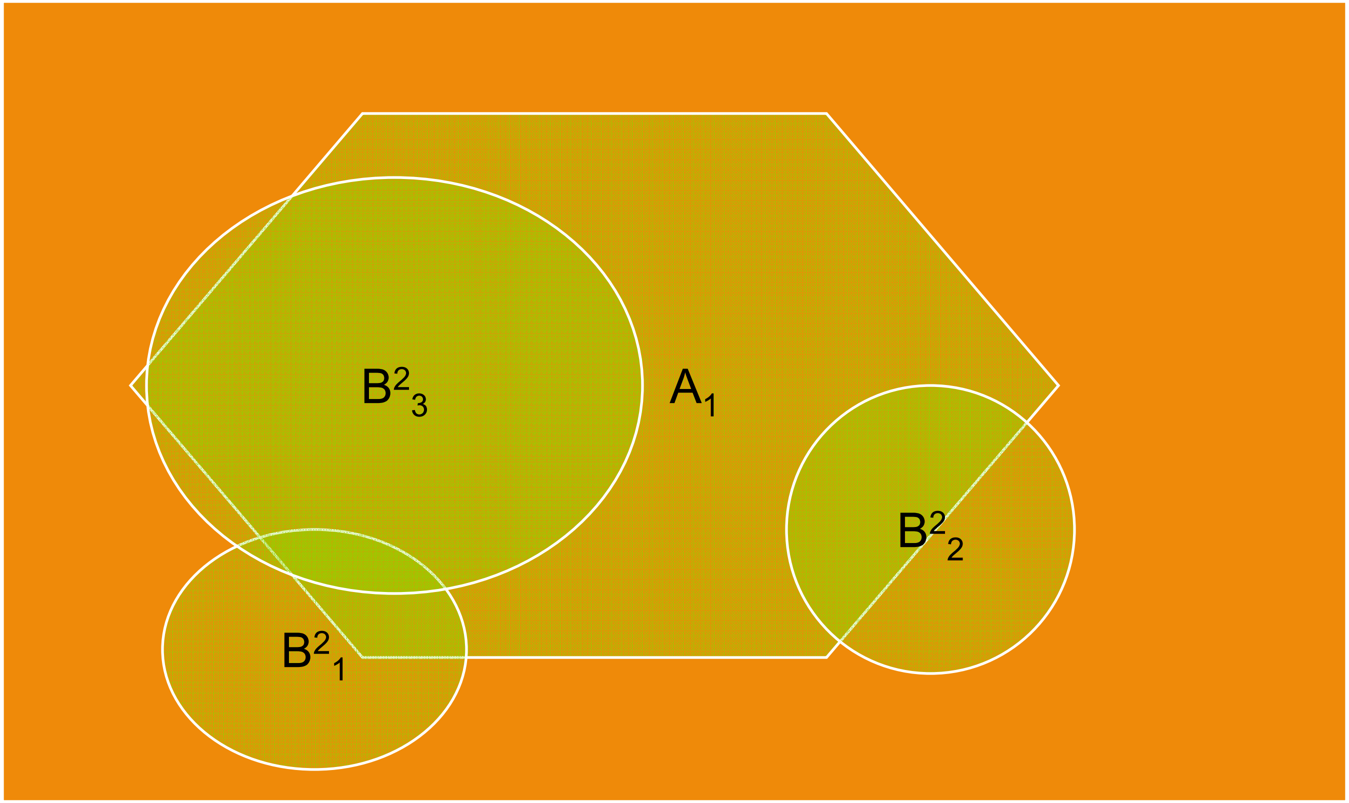
- a_i op a_j
  - Equivalent to $(a_i - a_j)$ op 0

Boolean Combination

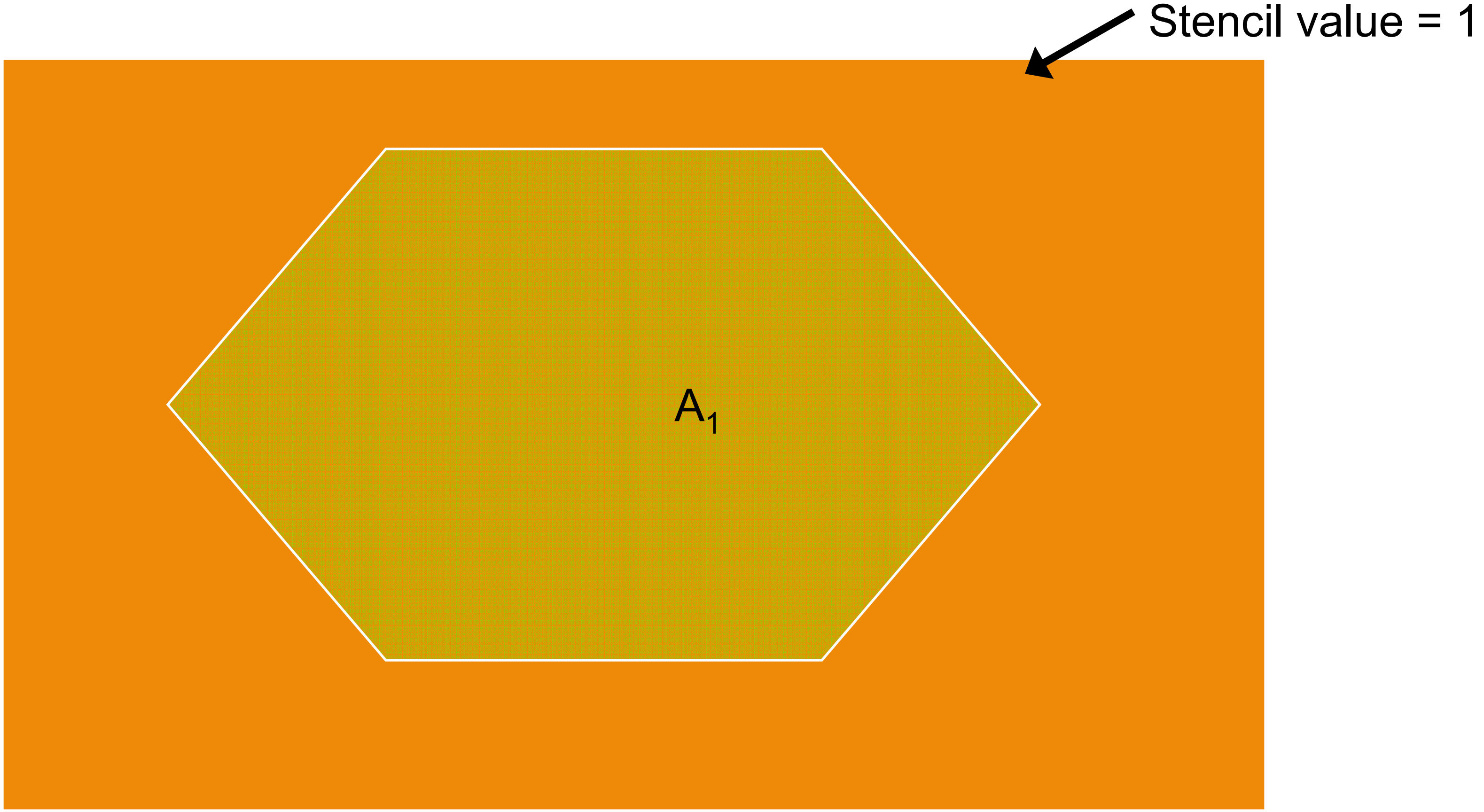
- CNF:
  - (A_1 AND A_2 AND … AND A_k) where
    A_i = ($B^i{}_1$ OR $B^i{}_2$ OR … OR $B^i{}_{mi}$)
- Performed using stencil test recursively
  - C_1 = (TRUE AND A_1) = A_1
  - C_i = (A_1 AND A_2 AND … AND A_i) = (C_{i-1} AND A_i)
- Different stencil values are used to code the outcome of C_i
  - Positive stencil values — pass predicate evaluation
  - Zero — fail predicate evaluation

A_1 AND A_2

$$A_2 = (B^2_1 \text{ OR } B^2_2 \text{ OR } B^2_3)$$

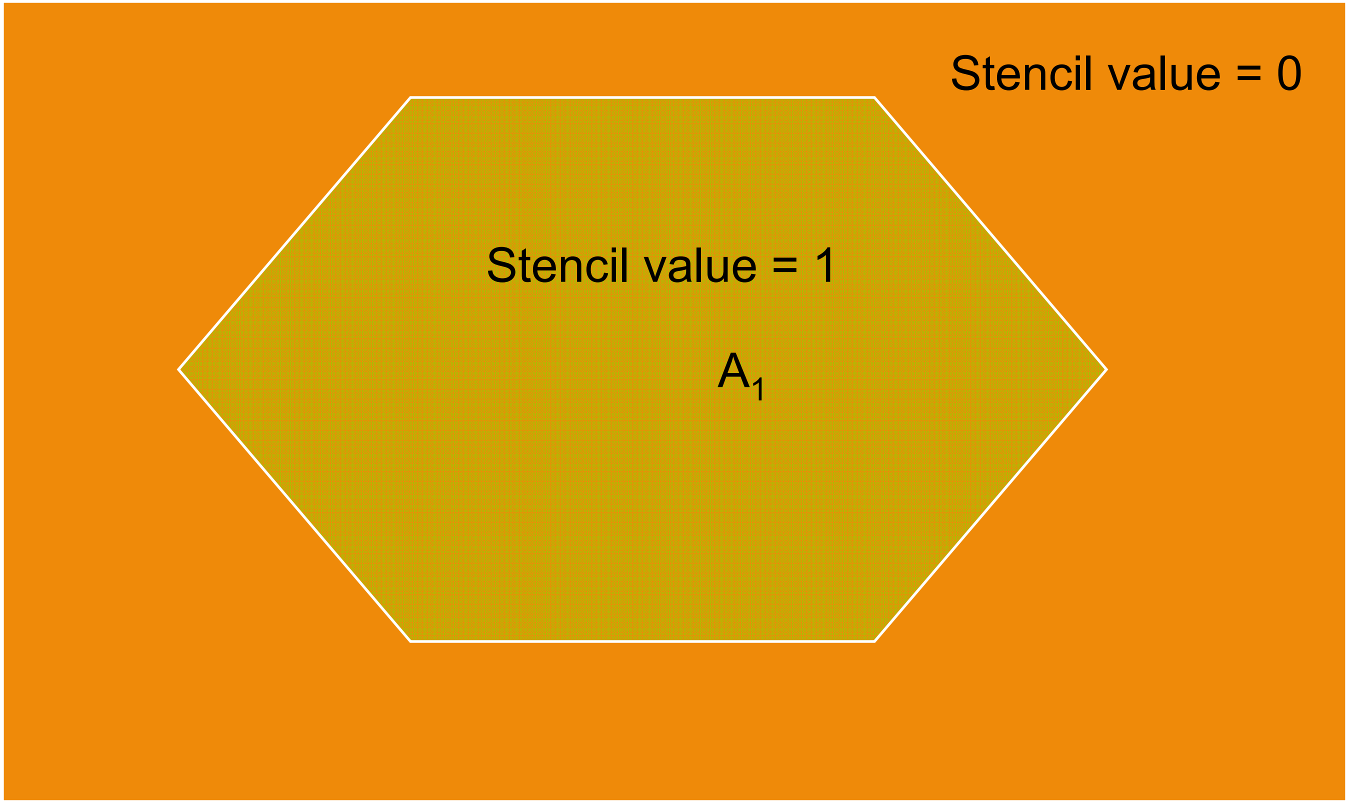


A_1 AND A_2

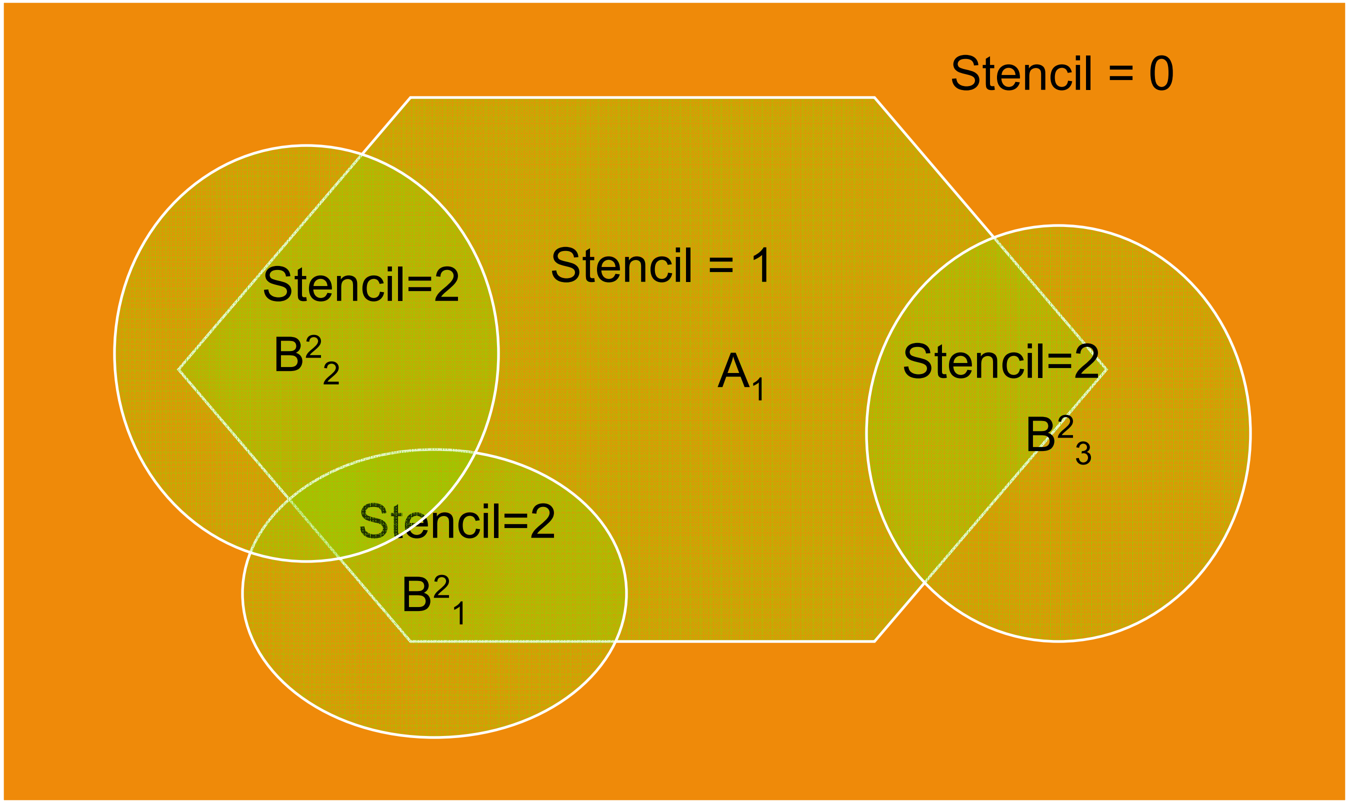


A_1 AND A_2

TRUE AND A_1

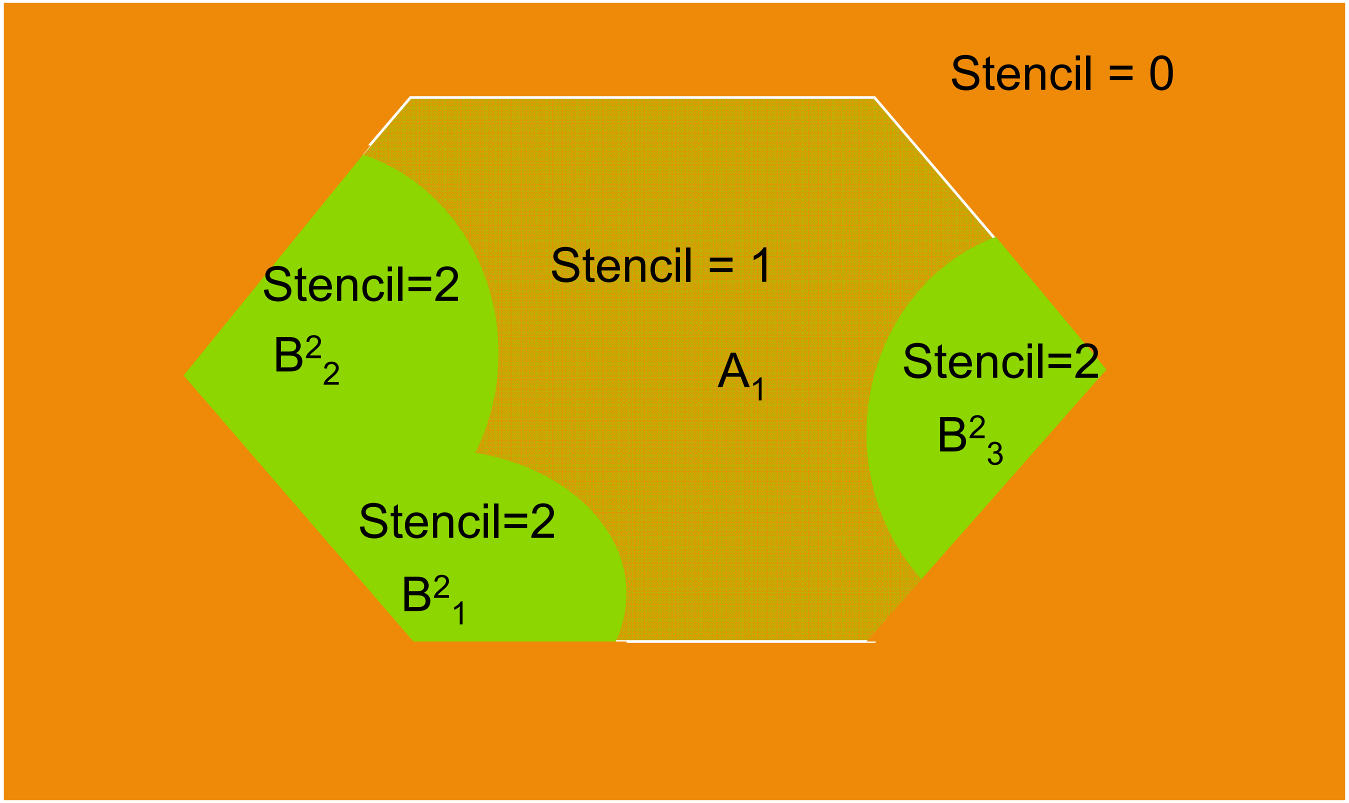


A_1 AND A_2



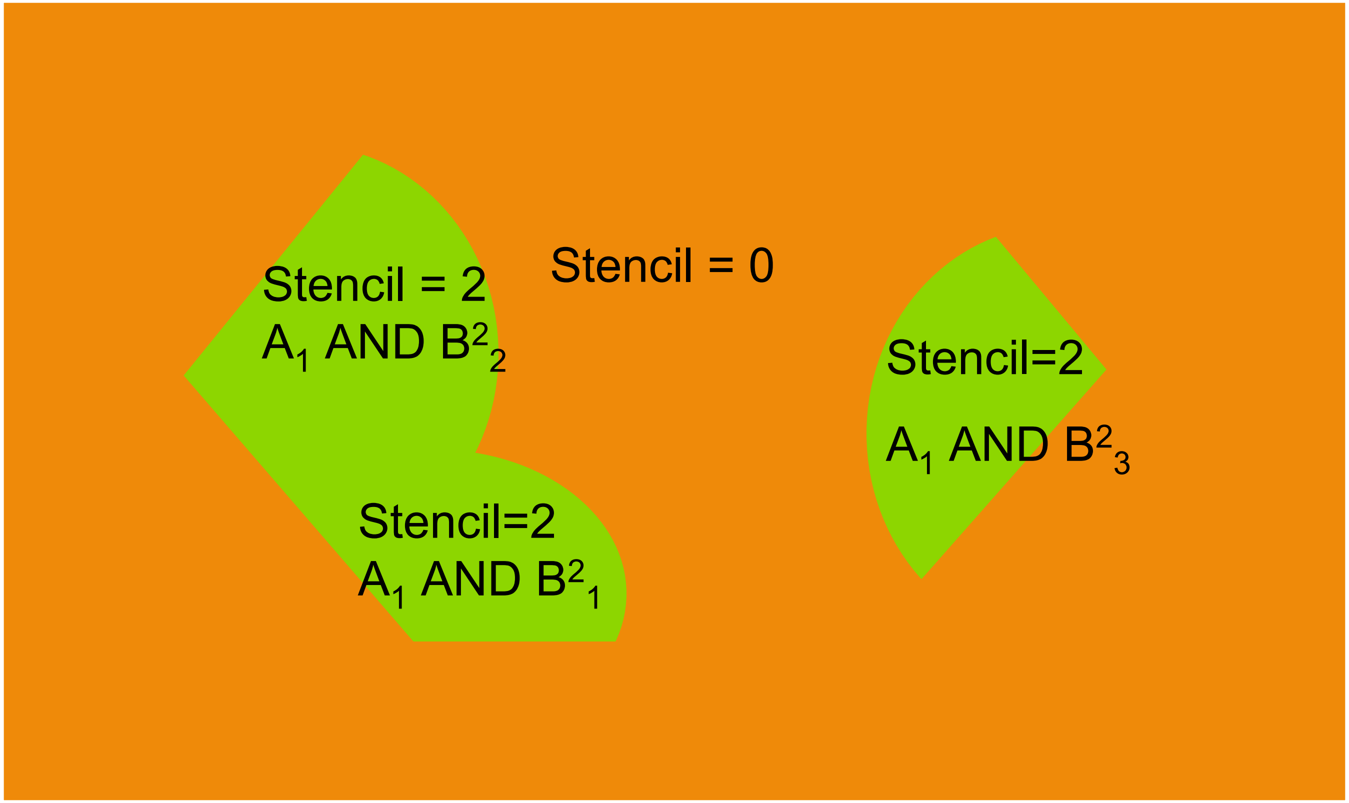
Stencil = 0
Stencil = 1
A_1
Stencil=2
B^2_2
Stencil=2
B^2_1
Stencil=2
B^2_3

A_1 AND A_2



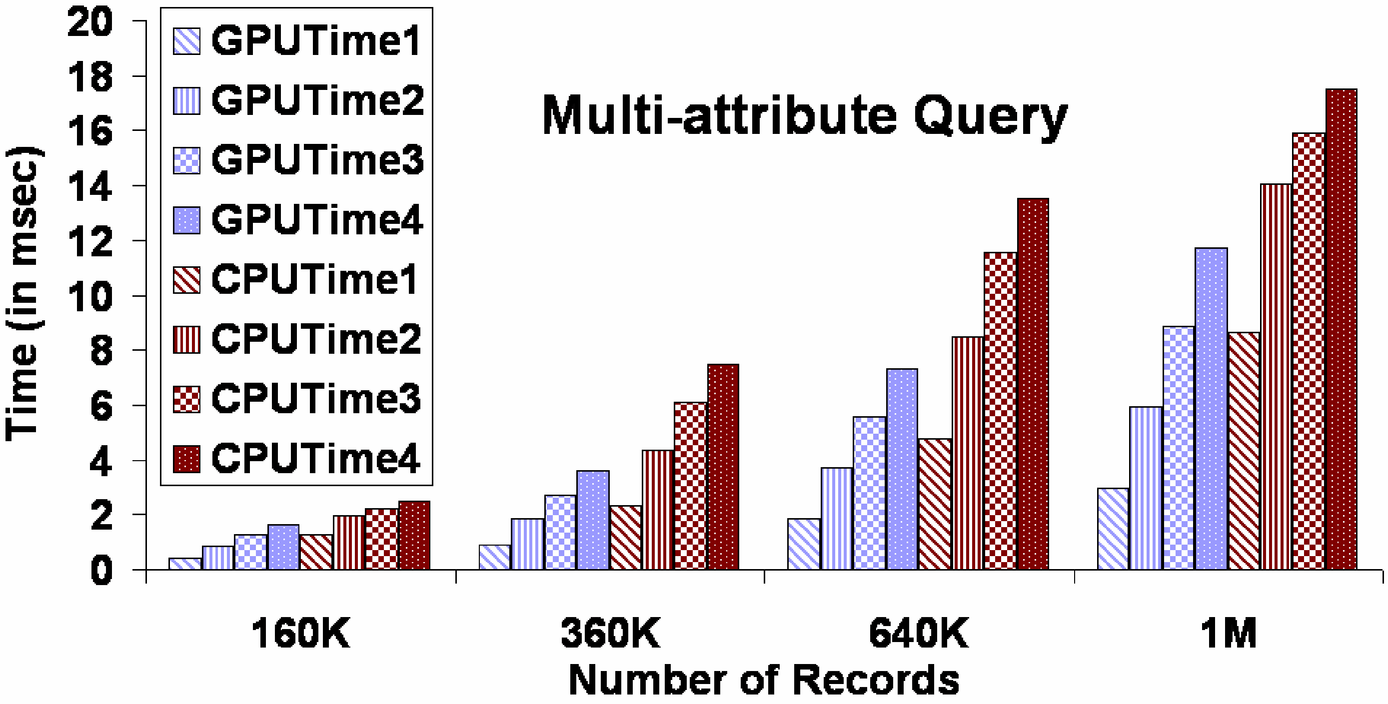
Stencil = 0
Stencil = 1
A_1
Stencil=2
$B^2{}_2$
Stencil=2
$B^2{}_3$
Stencil=2
$B^2{}_1$

A_1 AND A_2



Stencil = 0
Stencil = 2
A_1 AND $B^2{}_2$
Stencil=2
A_1 AND $B^2{}_1$
Stencil=2
A_1 AND $B^2{}_3$

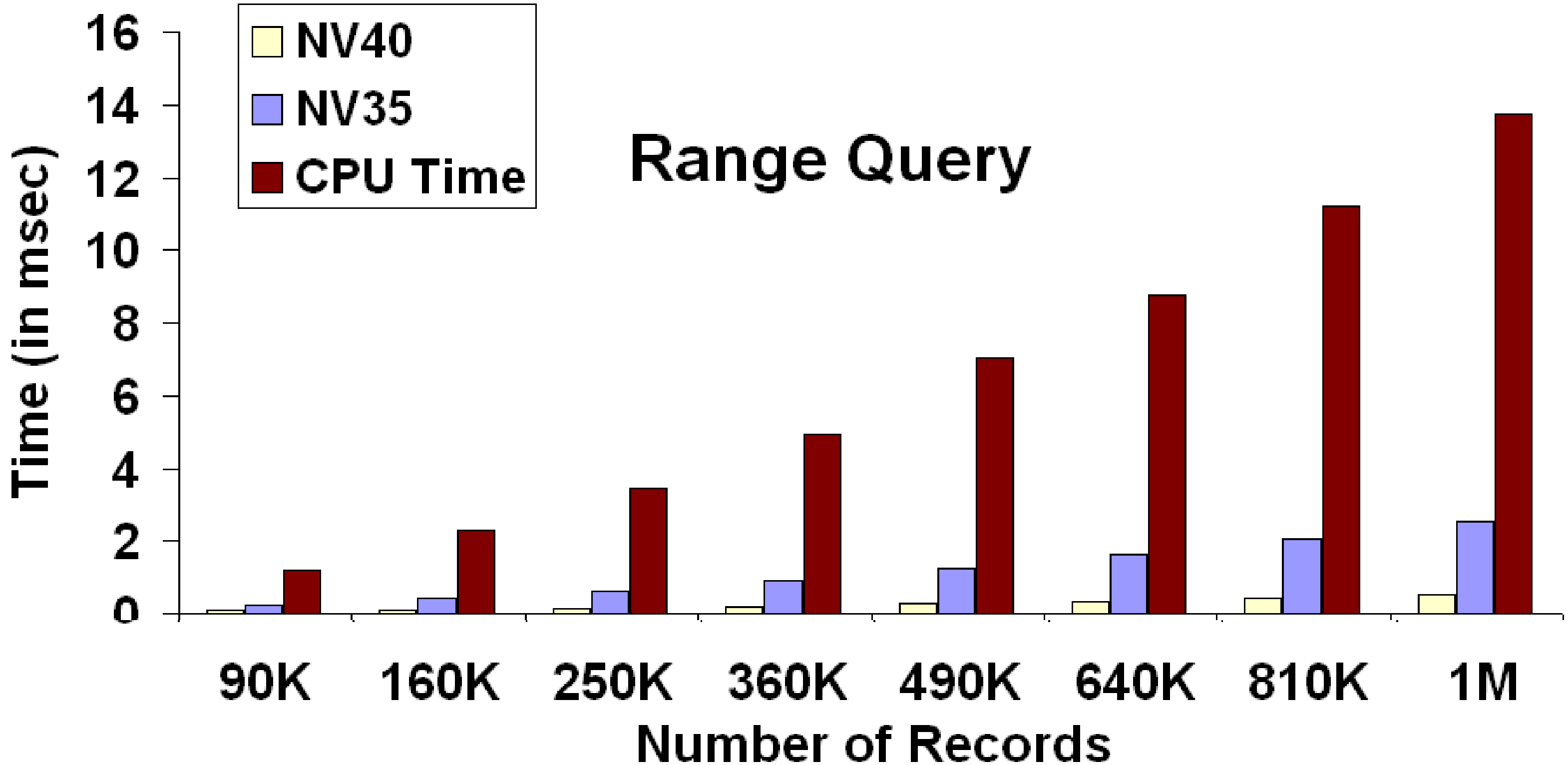
Multi-Attribute Query



Range Query

- Compute a_i within [low, high]
  - Evaluated as (a_i >= low) AND (a_i <= high)
- Use NVIDIA depth bounds test to evaluate both conditionals in a single clock cycle

Range Query



GPU is nearly 20 times faster than 2.8 GHz Xeon

Aggregations

- COUNT, MAX, MIN, SUM, AVG

COUNT

- Use occlusion queries to get the number of pixels passing the tests
- Syntax:
  - Begin occlusion query
  - Perform database operation
  - End occlusion query
  - Get count of number of attributes that passed database operation
- Involves no additional overhead!
- Efficient selectivity computation

MAX, MIN, MEDIAN

- Kth-largest number
- Traditional algorithms require data rearrangements
- We perform
  - no data rearrangements
  - no frame buffer readbacks

K-th Largest Number

- Given a set S of values
  - c(m) —number of values $\geq$ m
  - v_k — the k-th largest number
- We have
  - If $c(m) > k-1$, then $m \leq v_k$
  - If $c(m) \leq k-1$, then $m > v_k$
- Evaluate one bit at a time

2^{nd} Largest in 9 Values

0011	1011	1101
0111	0101	0001
0111	1010	0010

$m = 0000$
$v_2 = 1011$

Draw a Quad at Depth 8
Compute c(1000)

0011	1011	1101
0111	0101	0001
0111	1010	0010

m = 1000
v_2 = 1011

1^{st} bit = 1

0011	1011	1101
0111	0101	0001
0111	1010	0010

$m = 1000$
$v_2 = 1011$

$c(m) = 3$

Draw a Quad at Depth 12
Compute c(1100)

0011	1011	1101
0111	0101	0001
0111	1010	0010

m = 1100
v_2 = 1011

2^{nd} bit = 0

0011	1011	1101
0111	0101	0001
0111	1010	0010

m = 1100
v_2 = 1011

c(m) = 1

Draw a Quad at Depth 10
Compute c(1010)

0011	1011	1101
0111	0101	0001
0111	1010	0010

m = 1010
v_2 = 1011

3^{rd} bit = 1

0011	1011	1101
0111	0101	0001
0111	1010	0010

m = 1010
v_2 = 1011

c(m) = 3

Draw a Quad at Depth 11
Compute c(1011)

0011	1011	1101
0111	0101	0001
0111	1010	0010

$m = 1011$
$v_2 = 1011$

4^{th} bit = 1

0011	1011	1101
0111	0101	0001
0111	1010	0010

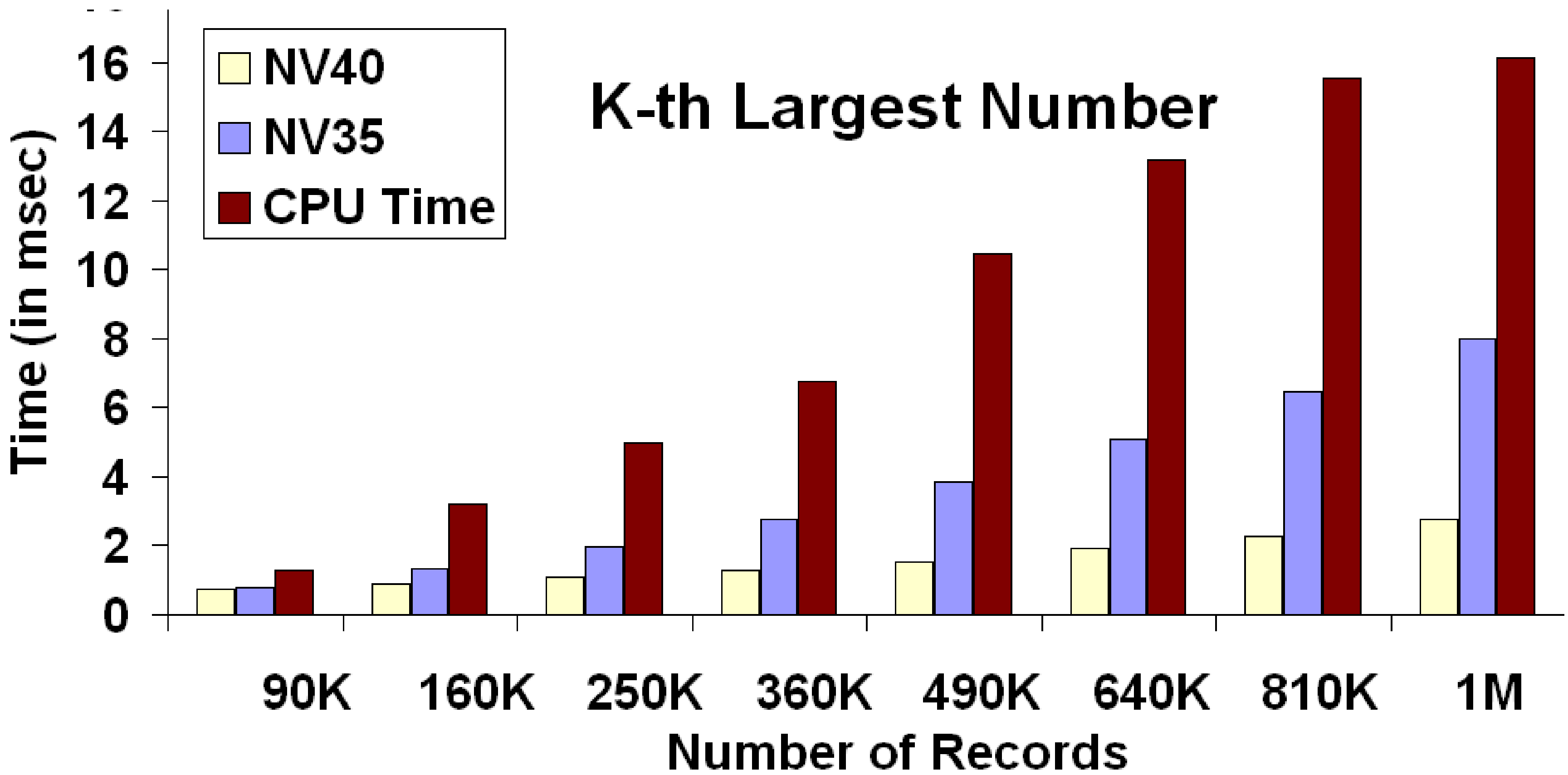
m = 1011**1**
v_2 = 1011

c(m) = 2

Our algorithm

- Initialize m to 0
- Start with the MSB and scan all bits till LSB
- At each bit, put 1 in the corresponding bit-position of m
- If c(m) < k, make that bit 0
- Proceed to the next bit

Median



GPU is nearly 6 times faster than 2.8 GHz Xeon!

Outline

- Graphics Processor Overview
- Mapping Computation to GPUs
- Database and data mining applications
  - Database queries
  - Quantile and frequency queries
  - External memory sorting
  - Scientific computations
- Summary

Streaming

N. Govindaraju, N. Raghuvanshi, D. Manocha
Proc. Of ACM SIGMOD 05

- Stream is a continuous sequence of data values arriving at a port
- Many real world applications process data streams
  - Networking data,
  - Stock marketing and financial data,
  - Data collected from sensors
  - Data logs from web trackers

Stream Queries

- Applications perform continuous queries and usually collect statistics on streams
  - *Frequencies* of elements
  - *Quantiles* in a sequence
  - And many more (SUM, MEAN, VARIANCE, etc.)

- Widely studied in databases, networking, computational geometry, theory of algorithms, etc.

Stream Queries

- Massive amounts of data is processed in real-time!
- Memory limitations – estimate the query results instead of exact results

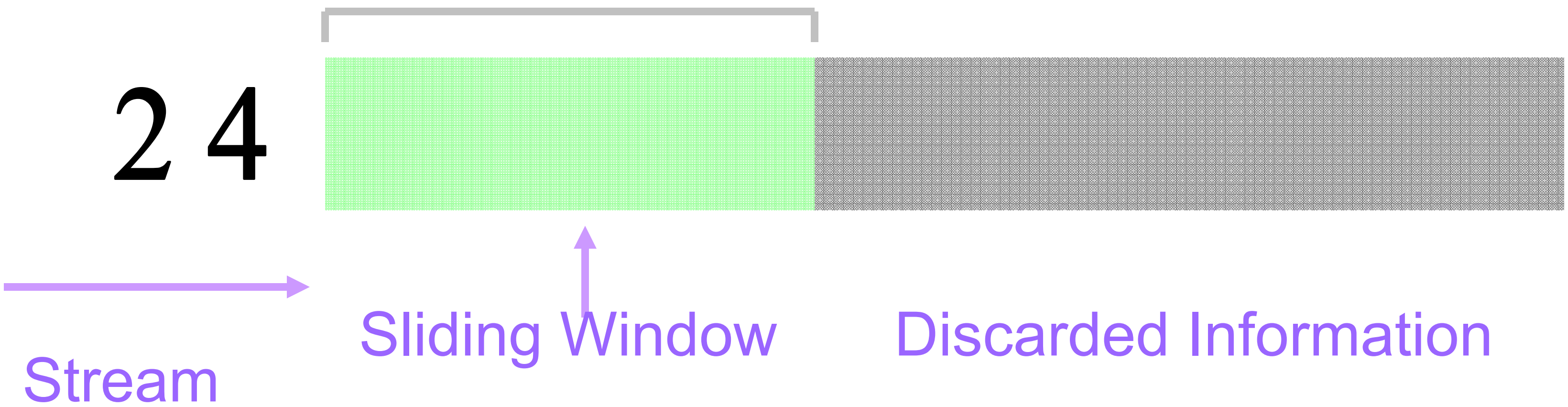
Approximate Queries

2 4

Stream history

Stream

Approximate Queries



ε-Approximate Queries

φ-*quantile* : element with rank $\lceil \varphi N \rceil$, $0 < \varphi < 1$

ε-approximate φ-*quantile* : Any element with rank $\lceil (\varphi \pm \varepsilon) N \rceil$ $0 < \varepsilon < 1$

Frequency : Number of occurrences of an element **f**

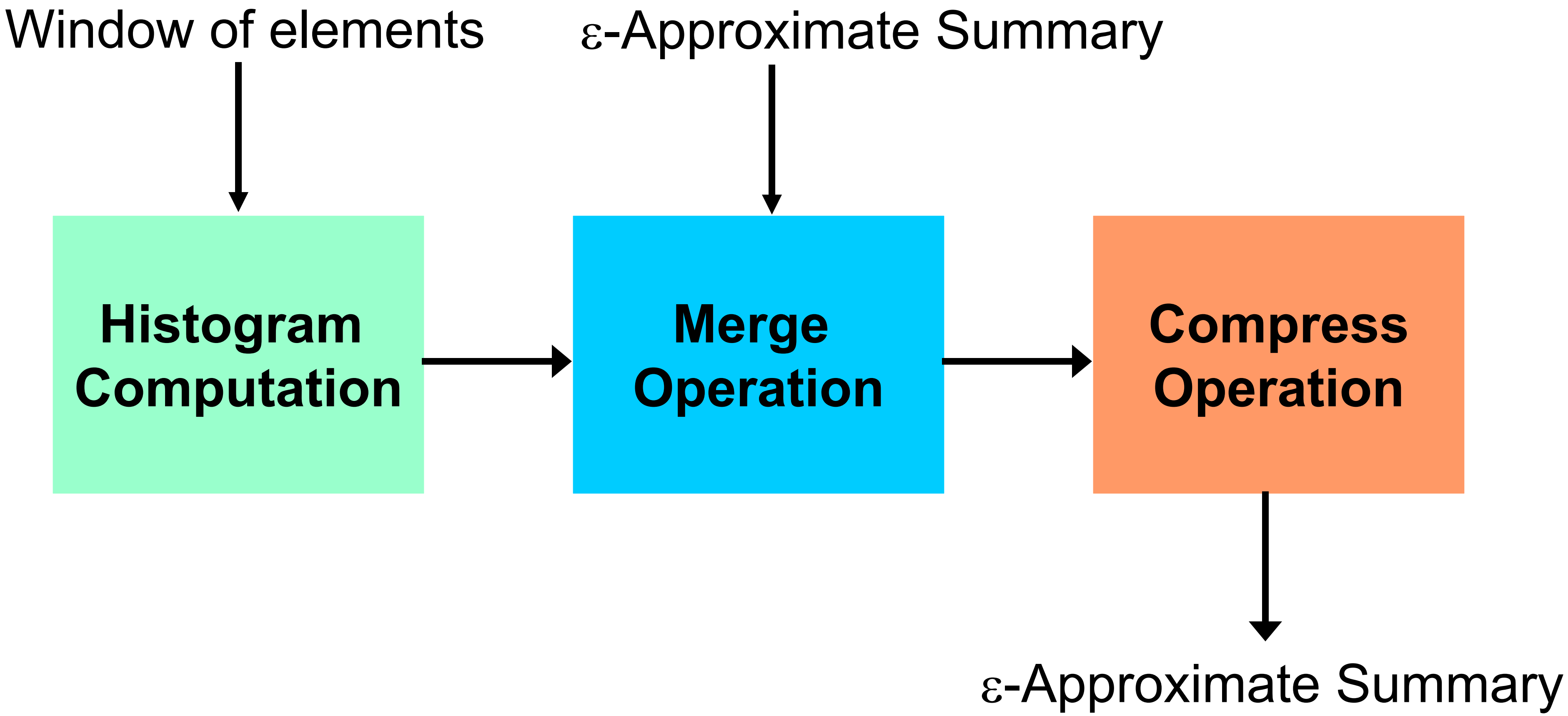
ε-approximate *frequency* : Any element with frequency $f \geq f' \geq (f - \varepsilon N)$, $0 < \varepsilon < 1$

ε-Approximate Queries

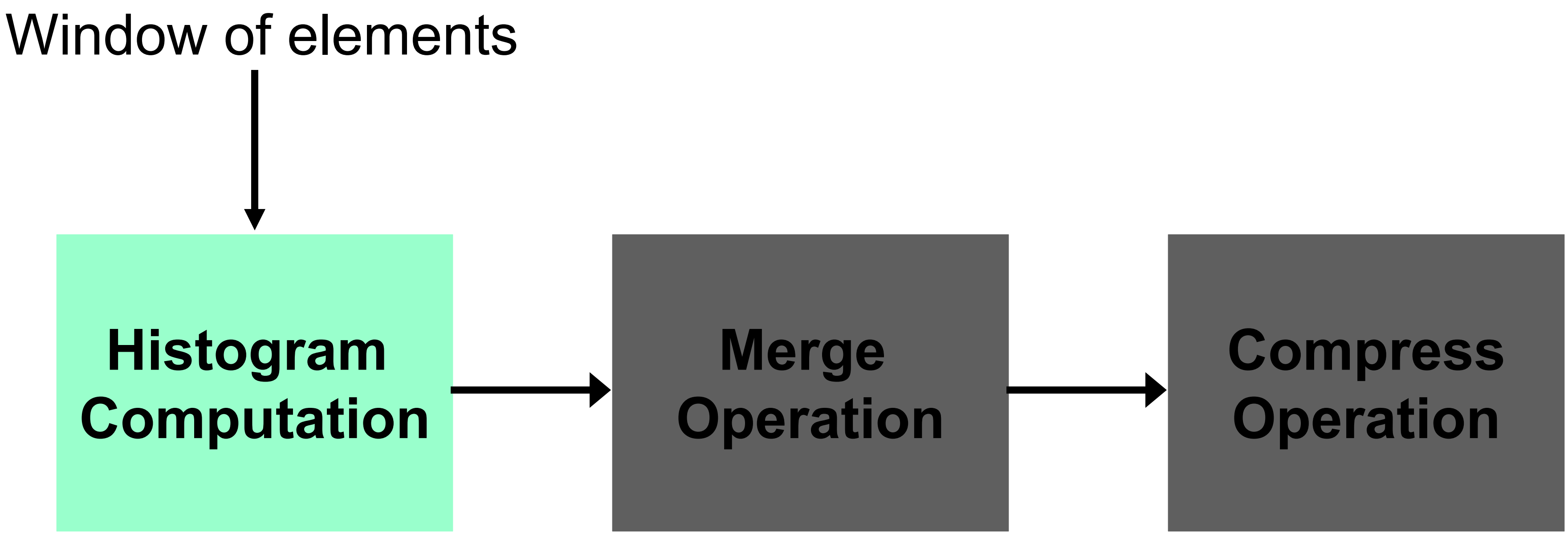
Queries computed using a ε-approximate summary data structure

- Performed by batch insertion a subset of window elements
- **Ref**: [**Manku and Motwani 2002, Greenwald and Khanna 2004, Arasu and Manku 2004**]

ε-Approximate Summary Construction

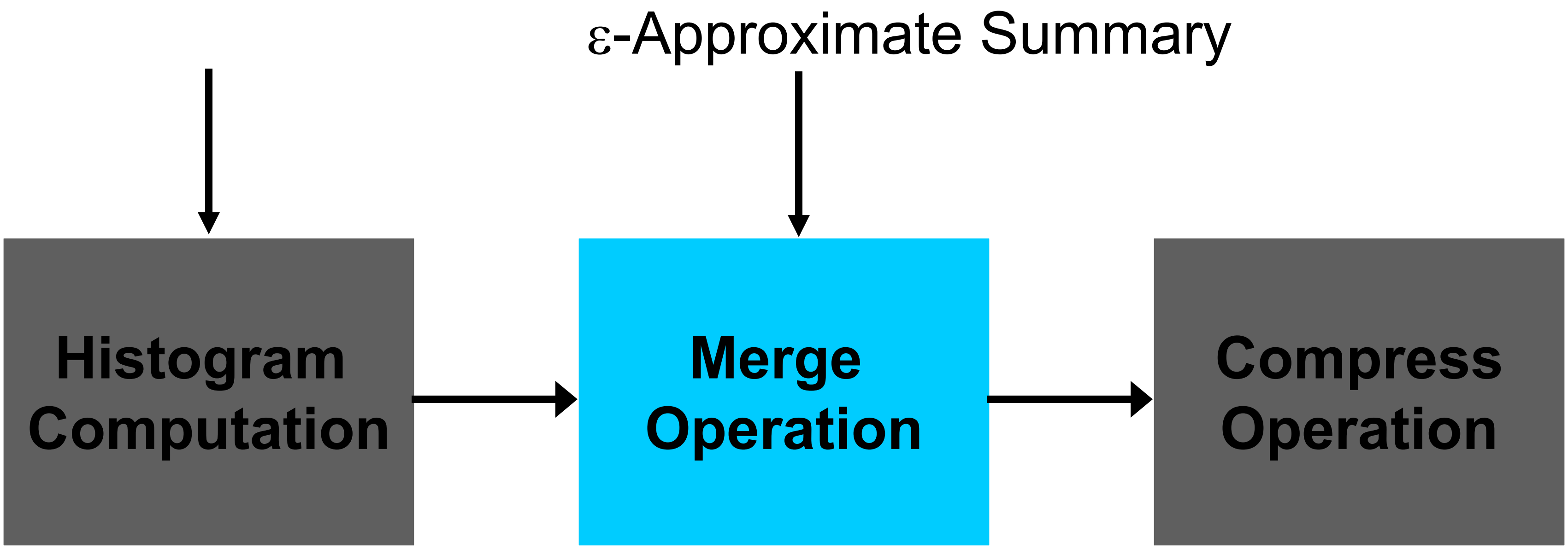


ε-Approximate Summary Construction



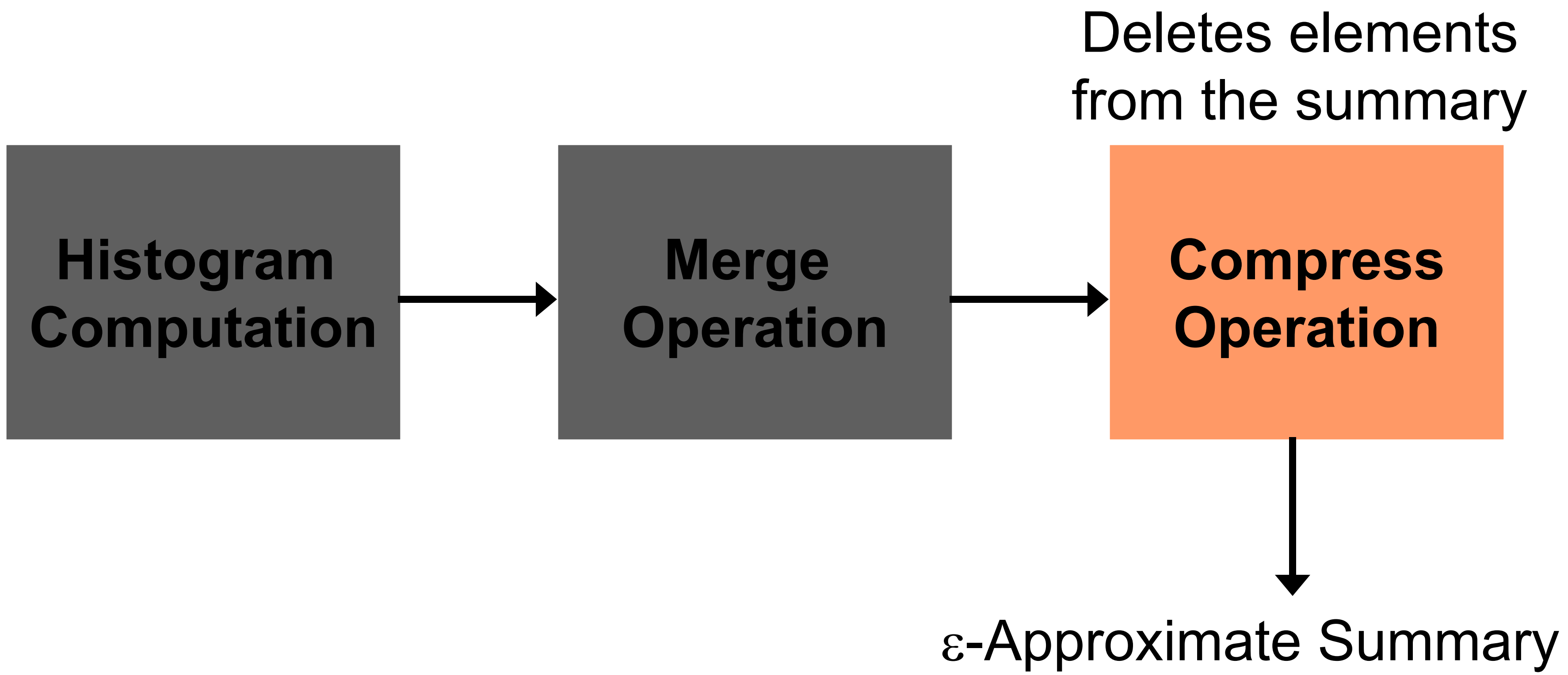
Computes histogram by sorting the elements

ε-Approximate Summary Construction

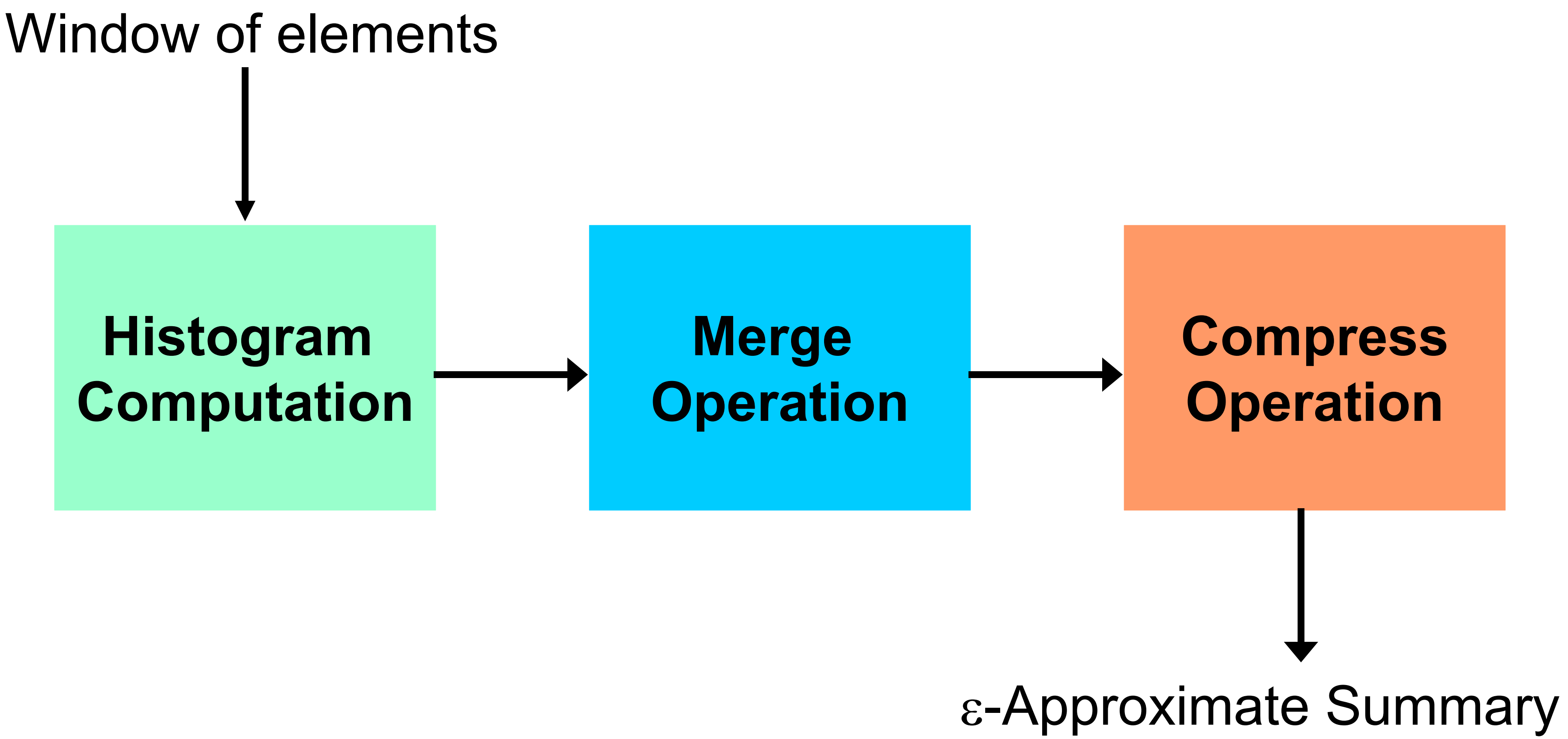


Inserts a subset of the histogram into the summary

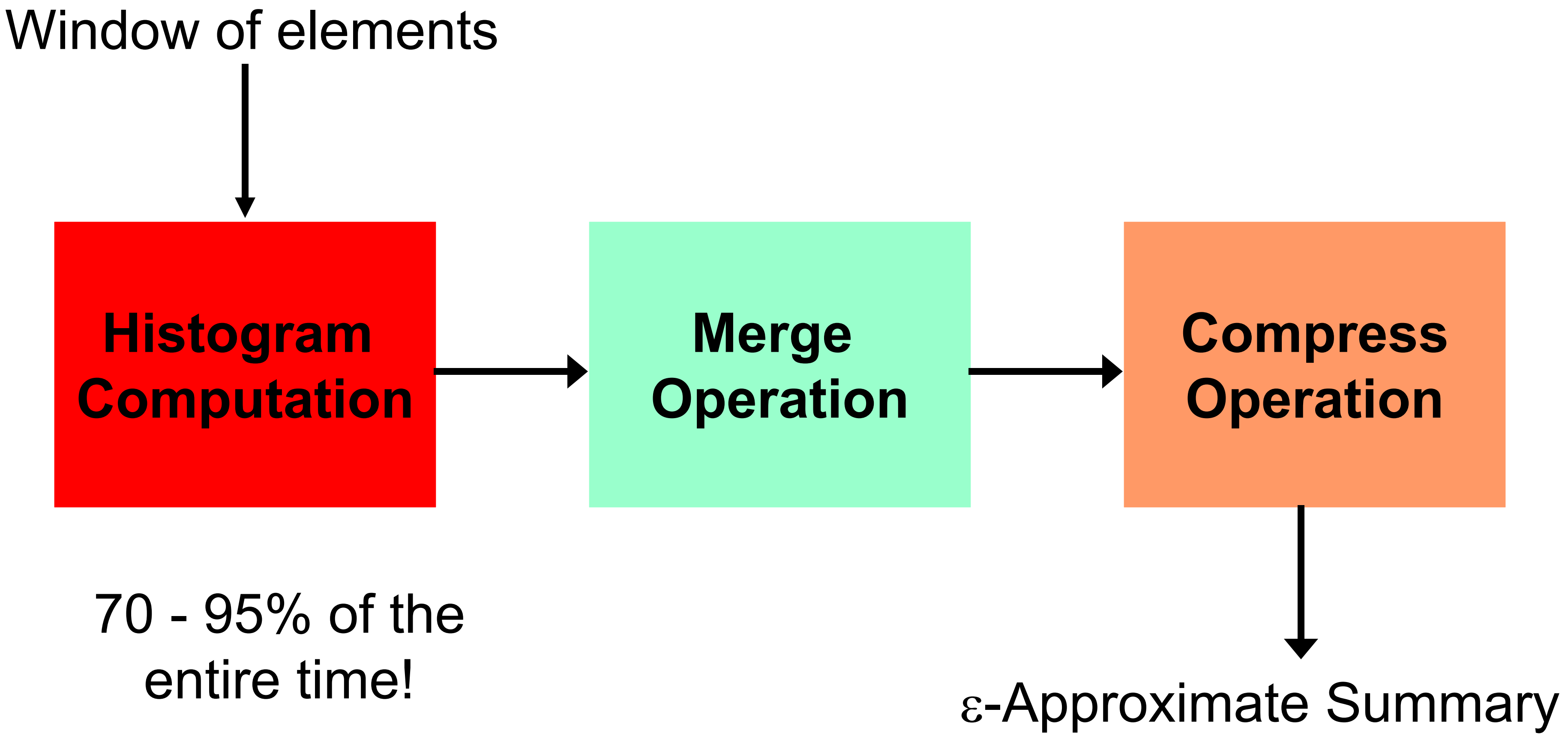
ε-Approximate Summary Construction



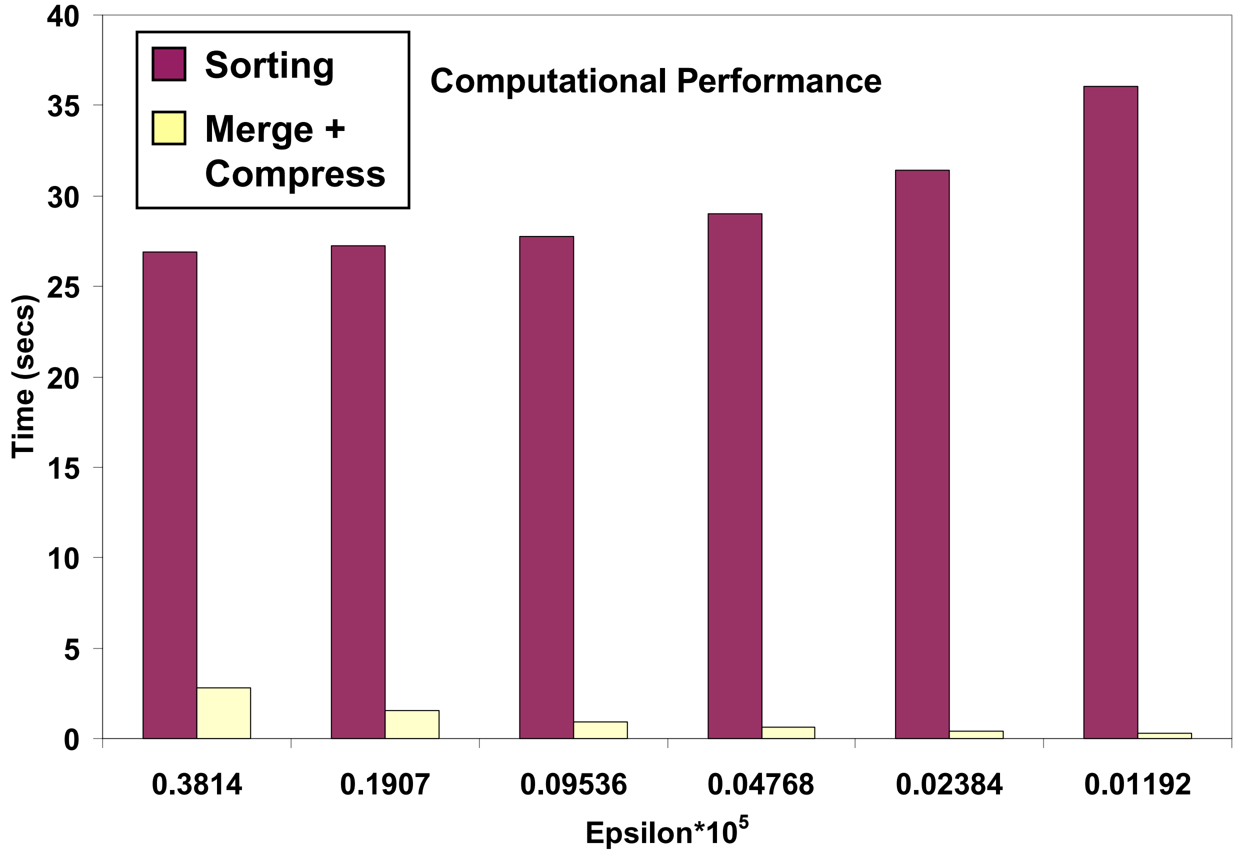
ε-Approximate Summary Construction



ε-Approximate Summary Construction



Timing Breakup: Frequency Estimation



Sorting takes nearly 90% time on CPUs

Sorting on CPUs

- Well studied
  - Optimized Quicksort performs better [LaMarca and Ladner 1997]
- Performance mainly governed by cache sizes
  - Large overhead per cache miss – nearly 100 clock cycles

Sorting on CPUs

- Sorting incurs cache misses
  - Irregular data access patterns in sorting
  - Small cache sizes (few KB)
- Additional stalls - branch mispredictions
- Degrading performance in new CPUs!
  [LaMarca and Ladner 97]

Sorting on GPUs

- Use the high data parallelism, and memory bandwidth on GPUs for fast sorting
- Many sorting algorithms require writes to arbitrary locations
  - Not supported on GPUs
  - Map algorithms with deterministic access pattern to GPUs (e.g., periodic balanced sorting network [Dowd 89])
  - Represent data in 2D images

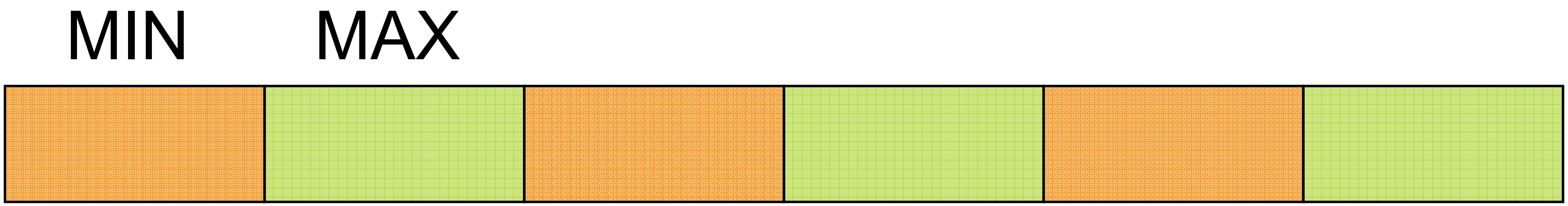
Sorting Networks

- Multi-stage algorithm
  - Each stage involves multiple steps
- In each step
  1. Compare one pixel against exactly one other pixel
  2. Perform a conditional assignment (MIN or MAX) at each pixel

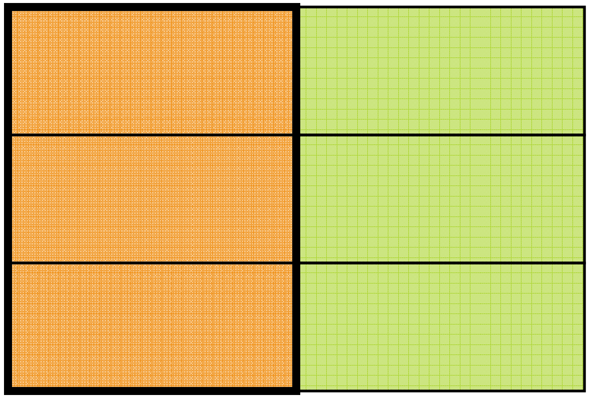
2D Memory Addressing

- GPUs optimized for 2D representations
  - Map 1D arrays to 2D arrays
  - Minimum and maximum regions mapped to row-aligned or column-aligned quads

1D – 2D Mapping



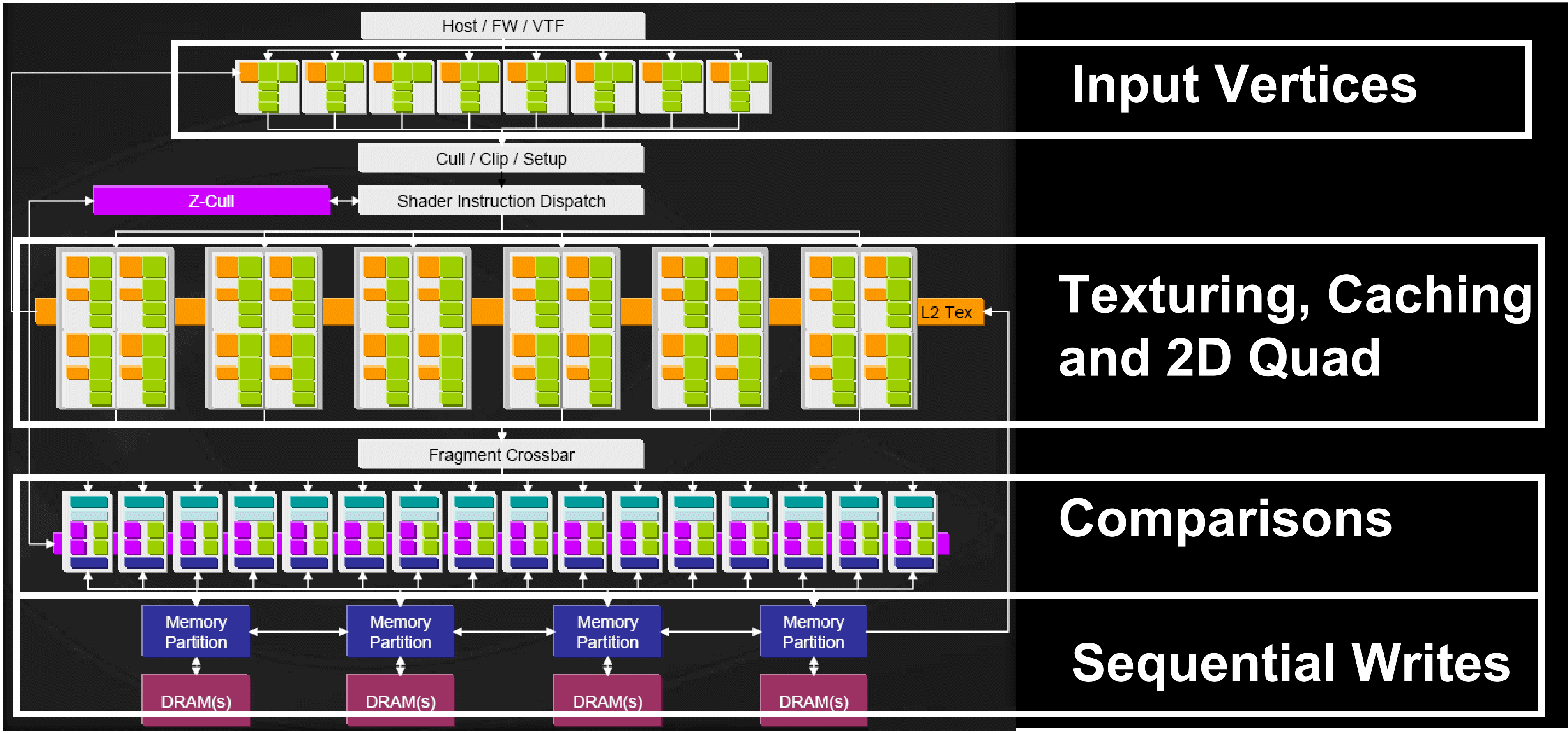
1D – 2D Mapping



MIN

Effectively reduce instructions per element

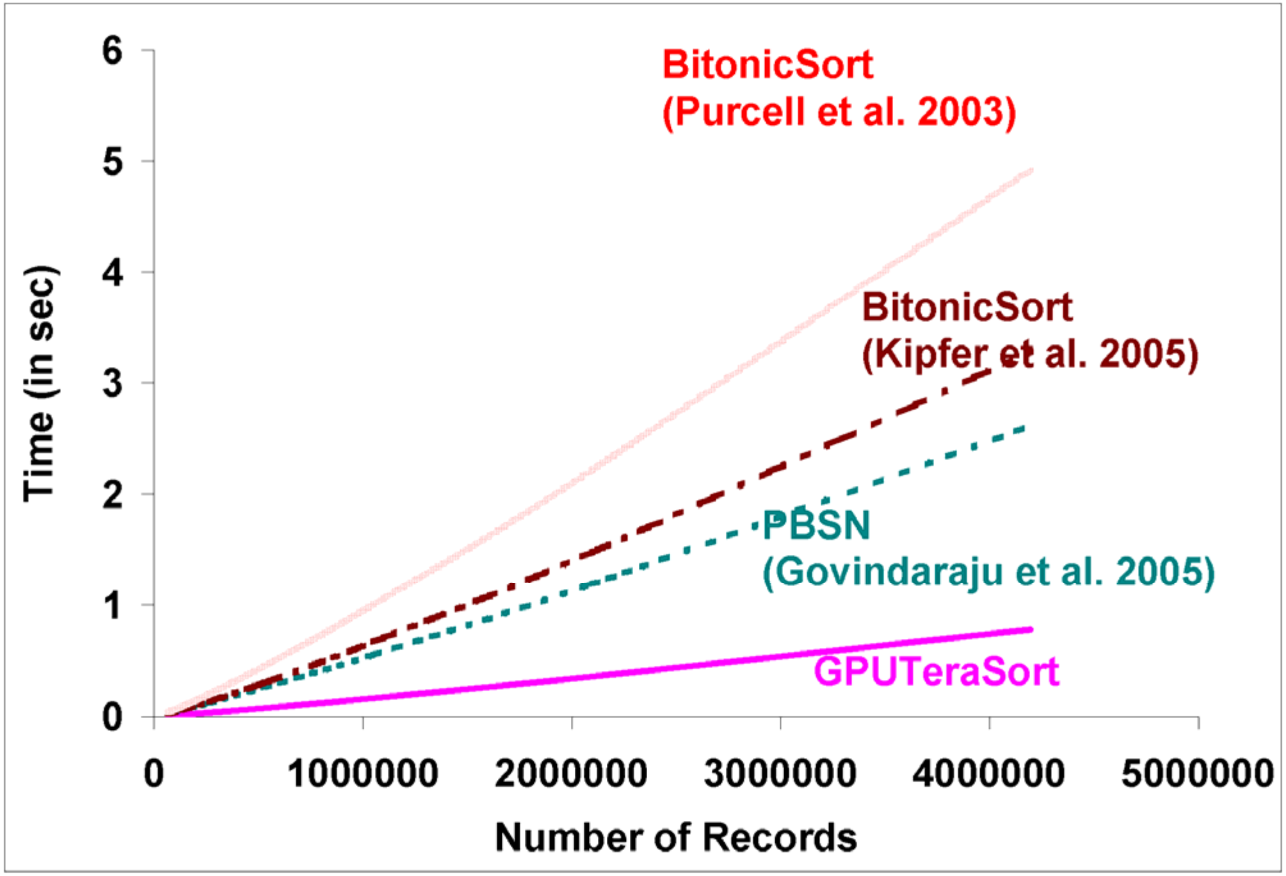
Sorting on GPU: Pipelining and Parallelism



Sorting Analysis

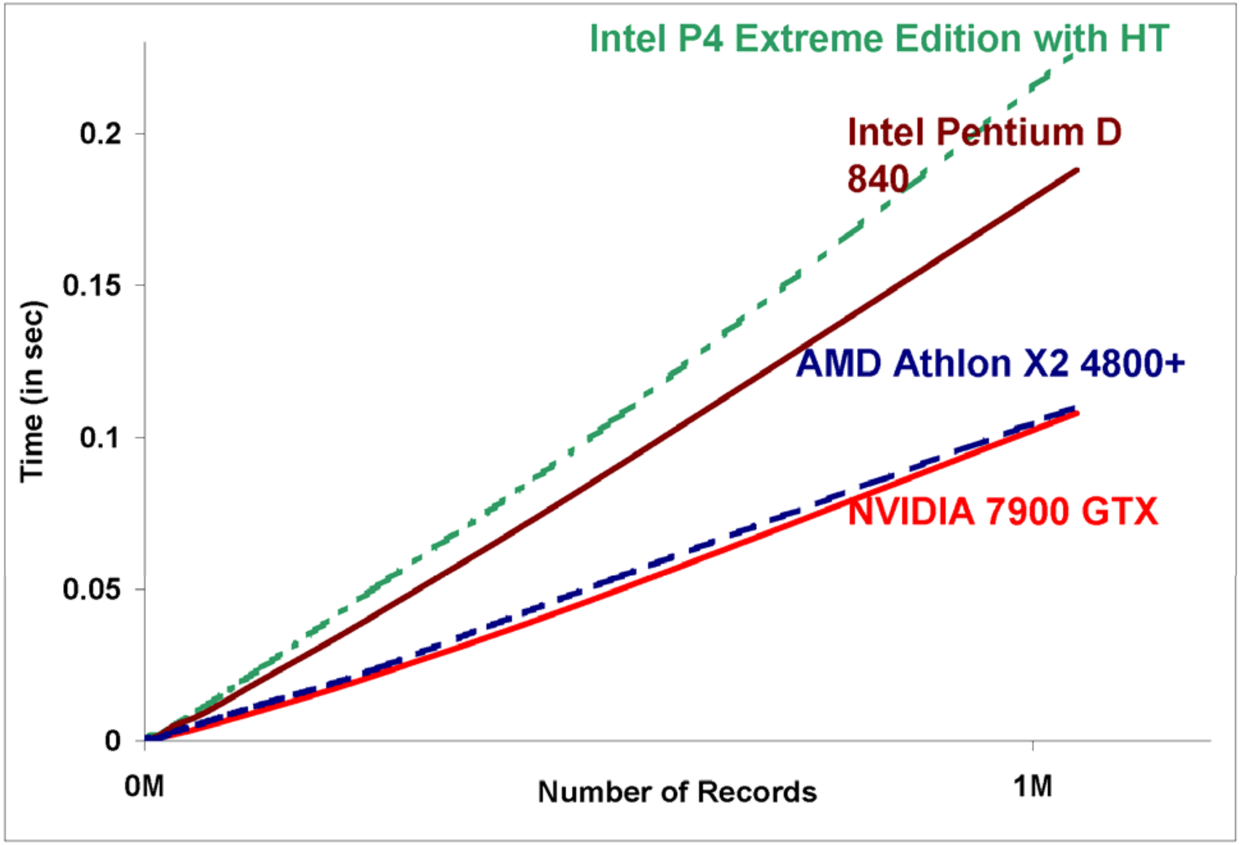
- Performed entirely on GPU
  - $O(\log^2 n)$ steps
  - Each step performs n comparisons
  - Total comparisons: $O(n \log^2 n)$
- Data sent and readback from GPU
  - Bandwidth: $O(n)$ — low bandwidth requirement from CPU to GPU

Comparison with GPU-Based Algorithms



3-6x faster than prior GPU-based algorithms!

GPU vs. High-End Multi-Core CPUs



2-2.5x faster than Intel high-end processors

Single **GPU performance comparable to *high-end dual core* Athlon**

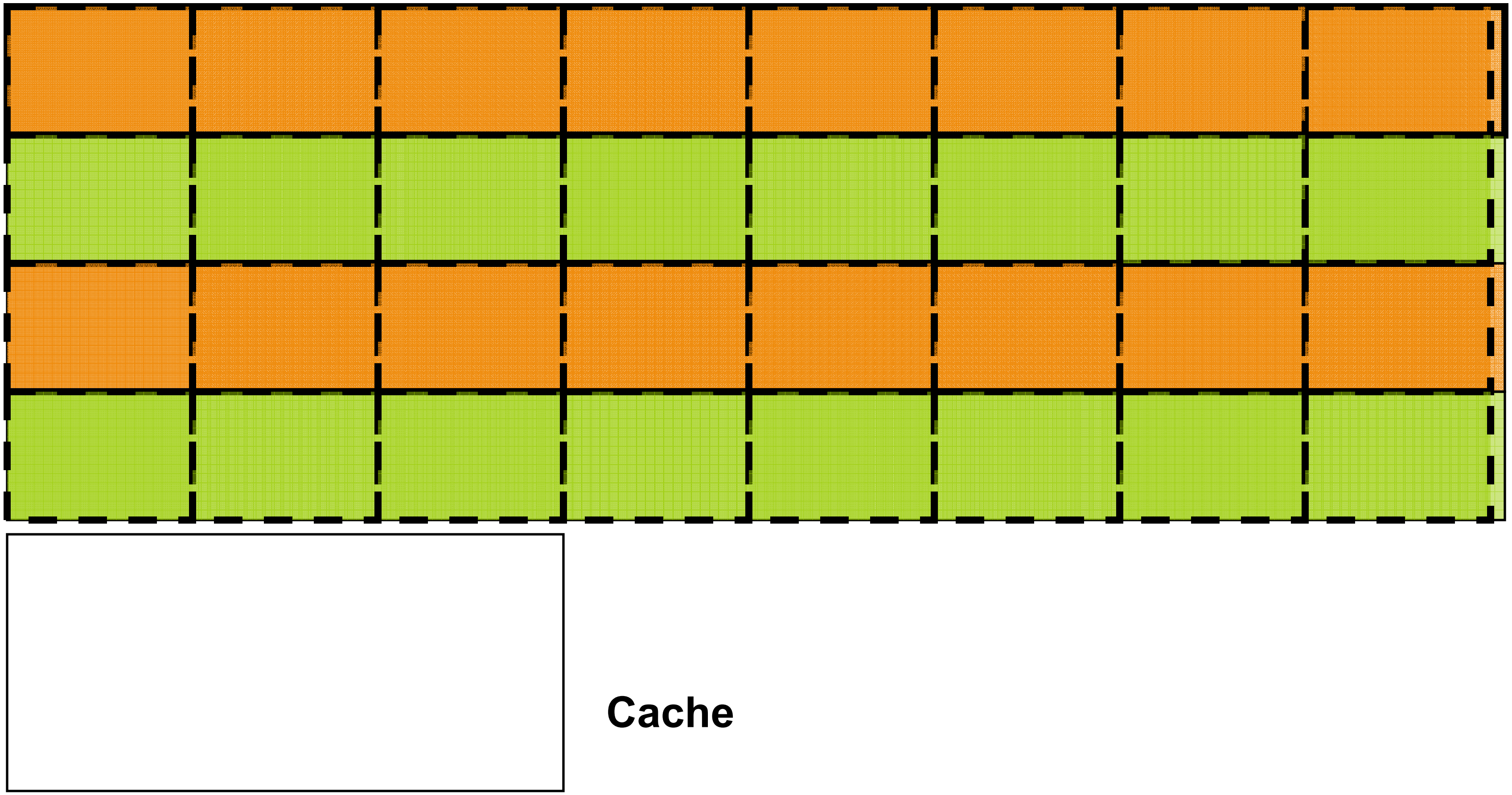
Hand-optimized CPU code from Intel Corporation!

N. Govindaraju, J. Gray, D. Manocha
Microsoft Research TR 2005

GPU Cache Model

- Small data caches
  - Low memory latency
  - Vendors do not disclose cache information – critical for scientific computing on GPUs
- We design simple model
  - Determine cache parameters (block and cache sizes)
  - Improve sorting performance

Cache Evictions



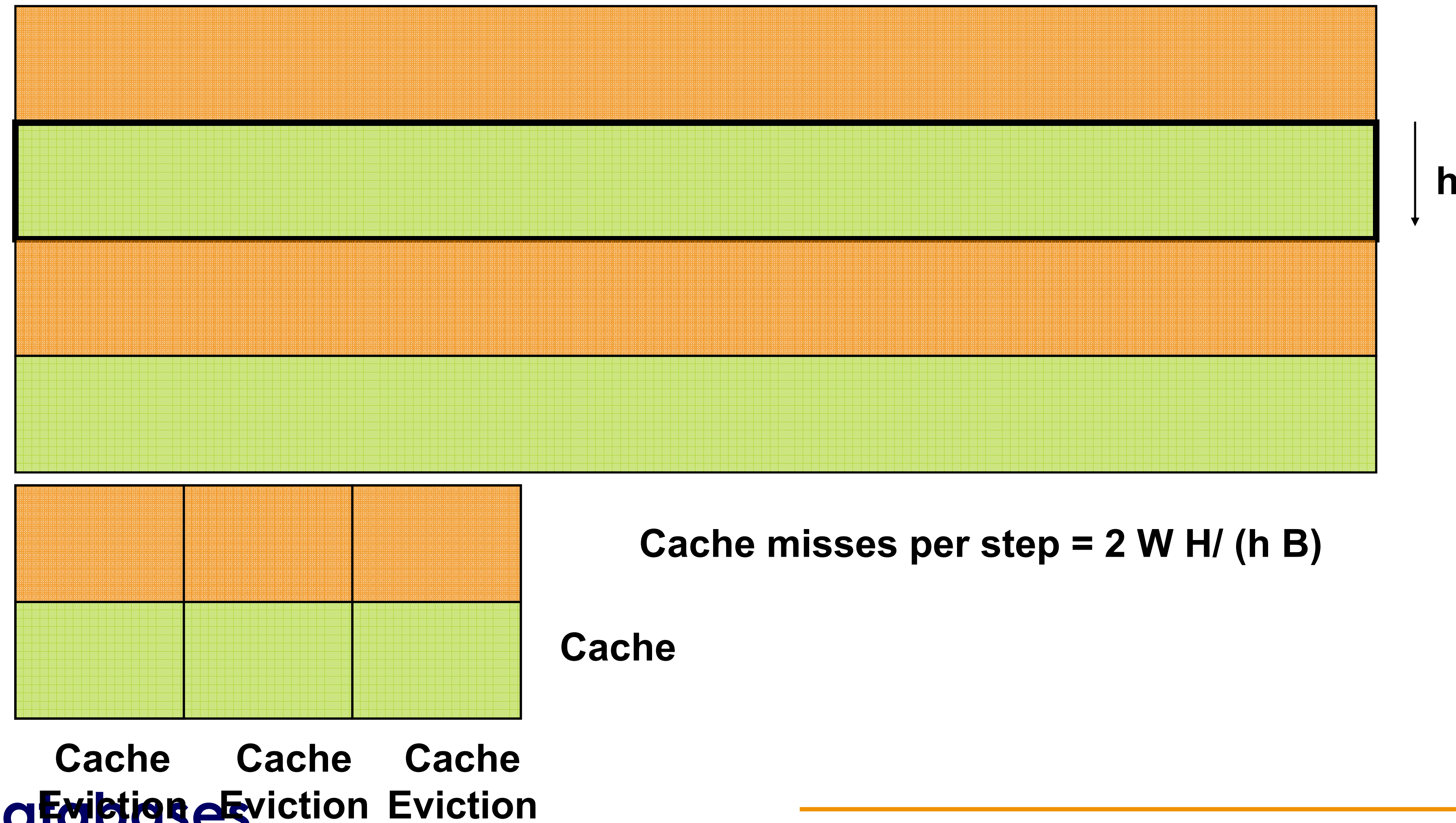
Cache

Cache Eviction

Cache Eviction

Cache Eviction

Cache issues



Cache misses per step = 2 W H/ (h B)

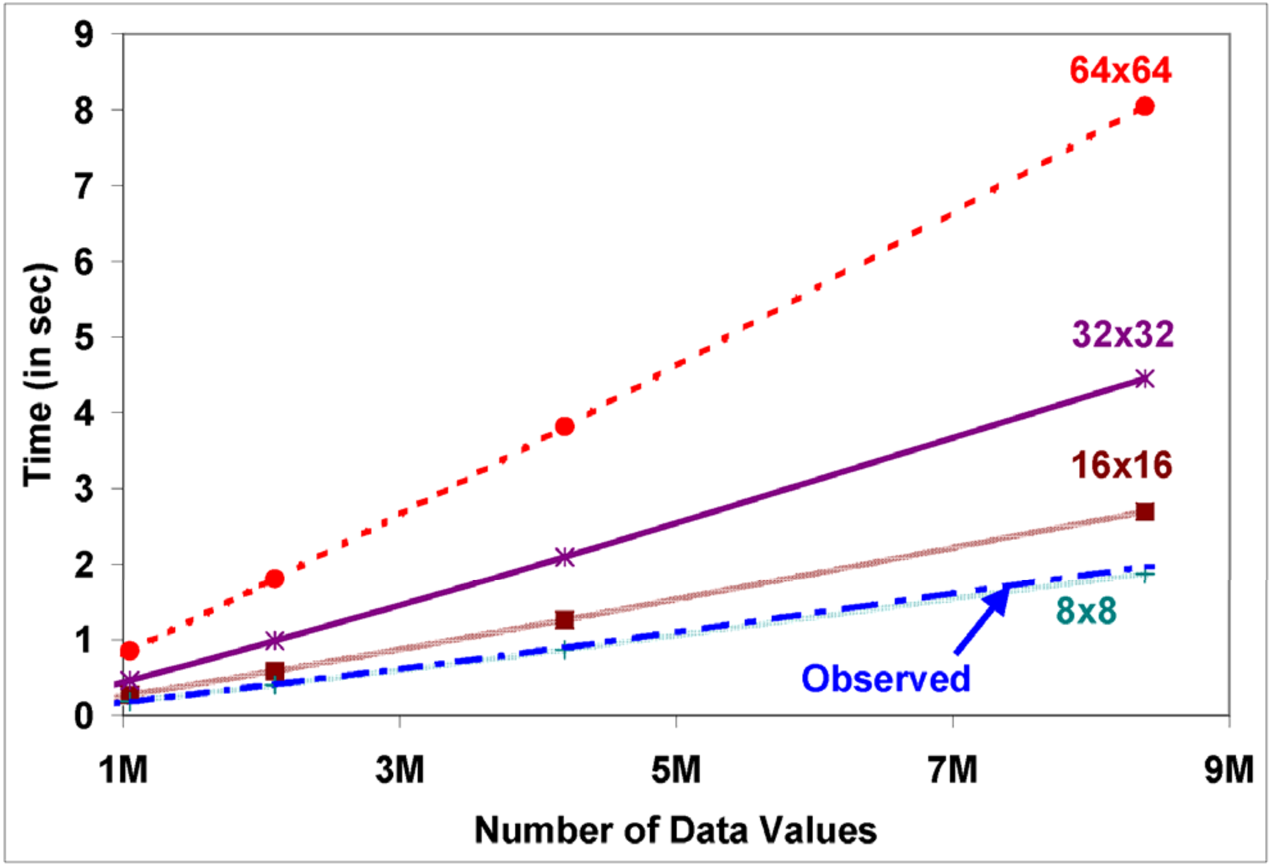
Analysis

- *lg* n possible steps in bitonic sorting network

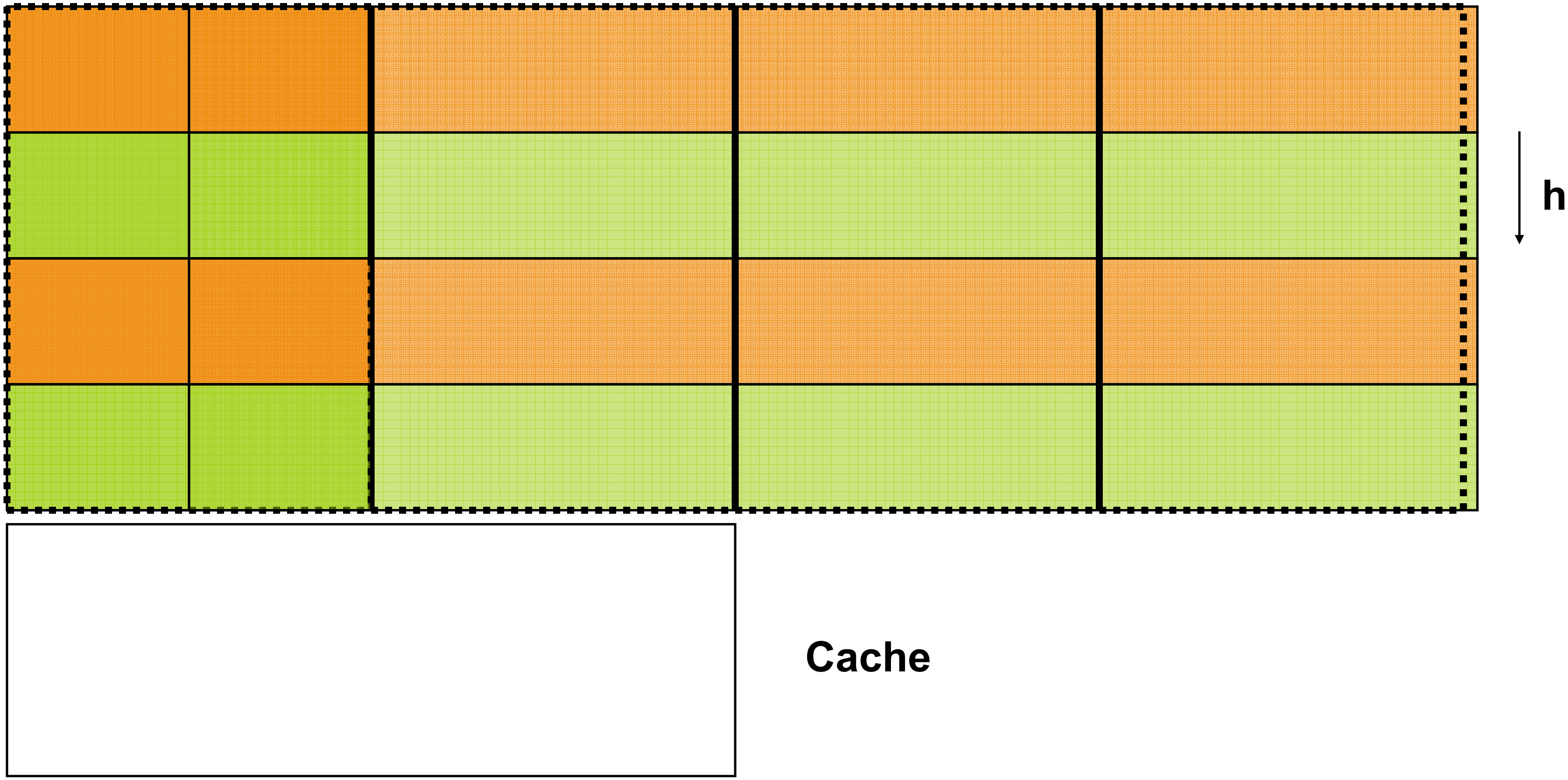
- Step k is performed (*lg* n – k+1) times and $h = 2^{k-1}$

- Data fetched from memory = **2 n f(B)** where
  f(B)=(B-1) (*lg* n -1) + 0.5 (*lg* n –*lg* B)²

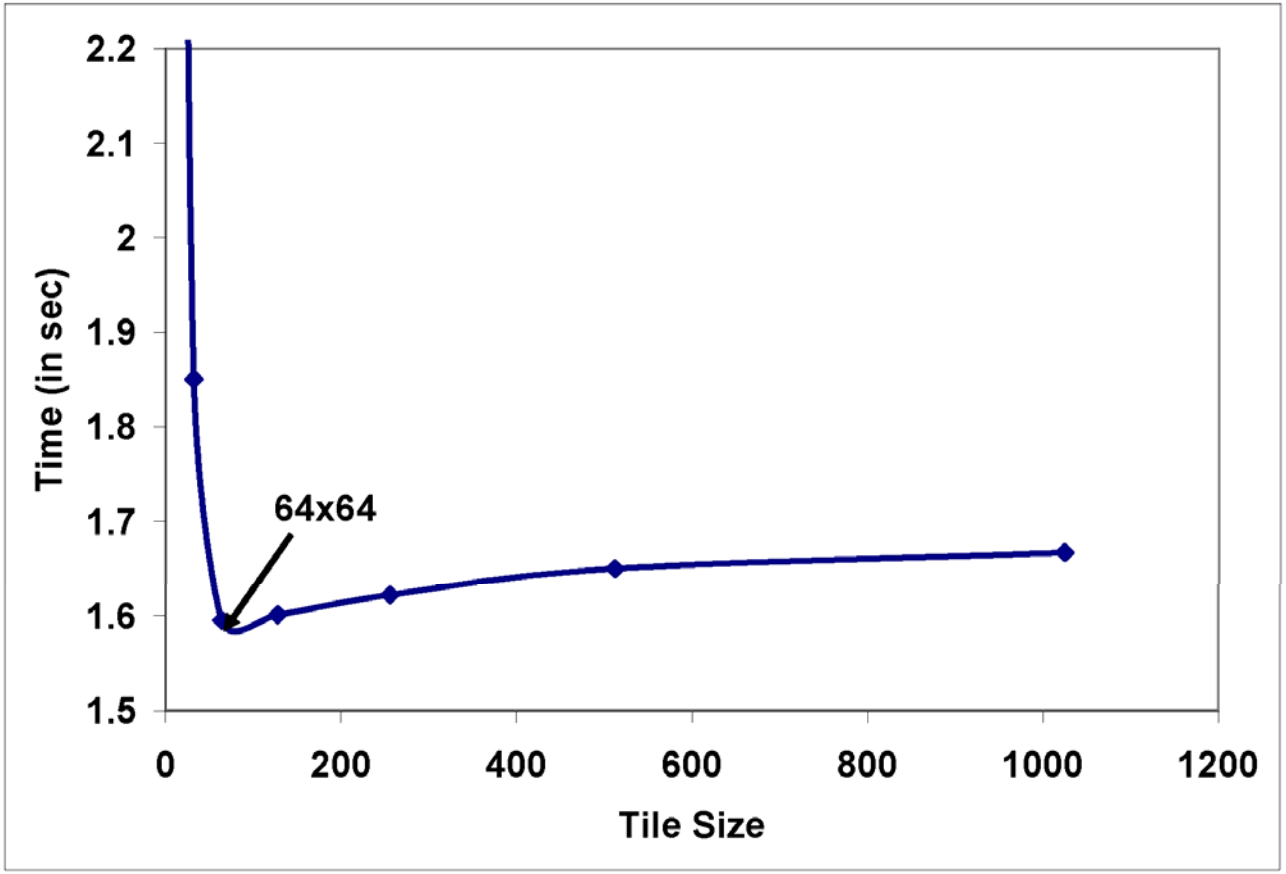
Block Sizes on GPUs



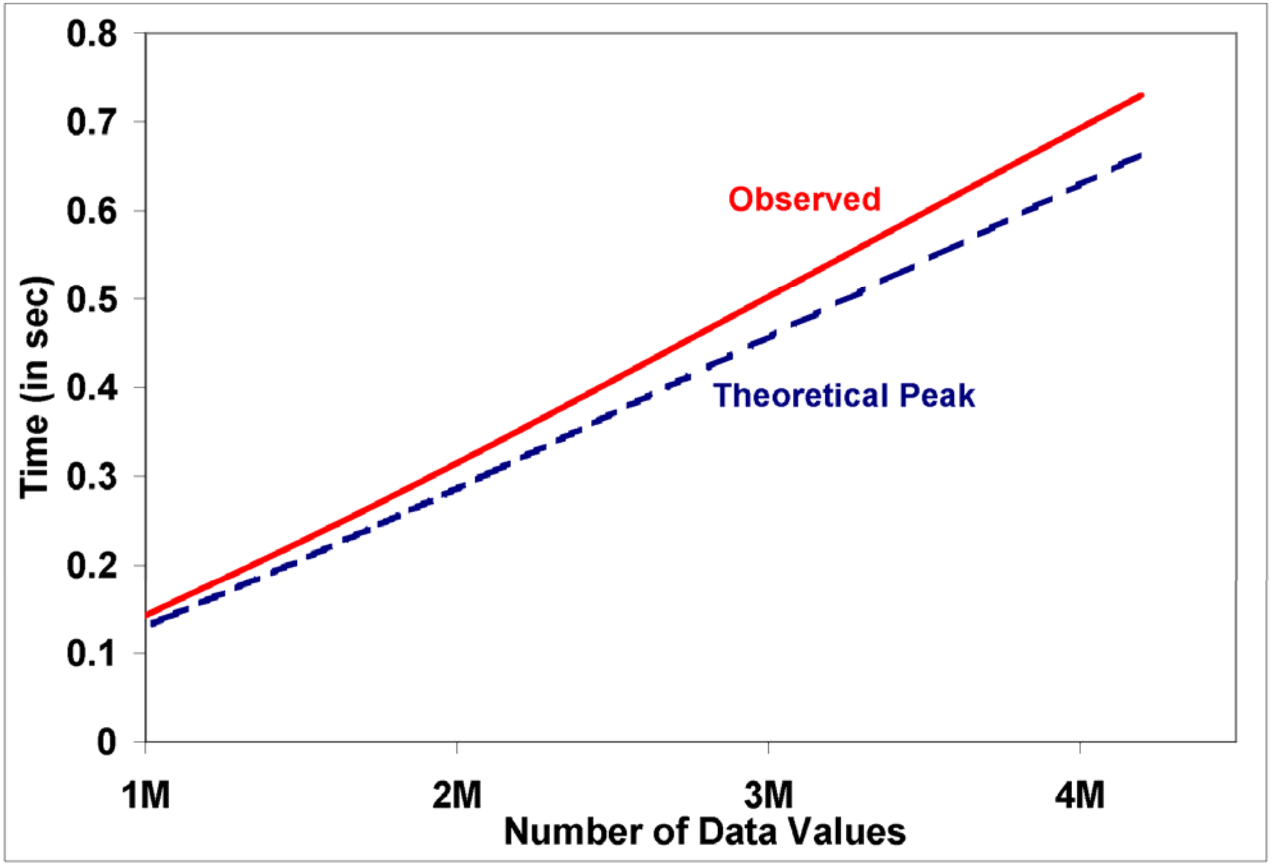
Cache-Efficient Algorithm



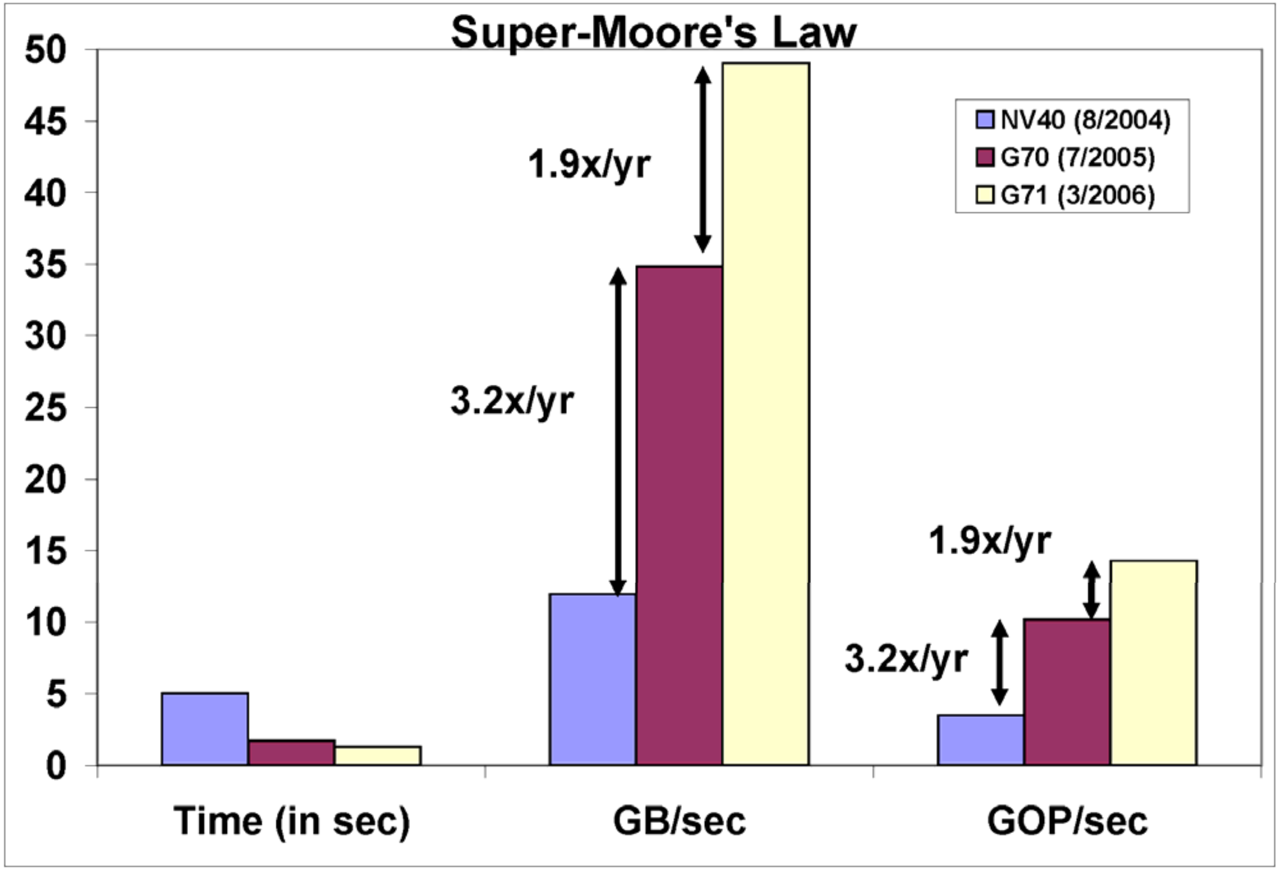
Cache Sizes on GPUs



Cache-Efficient Algorithm Performance



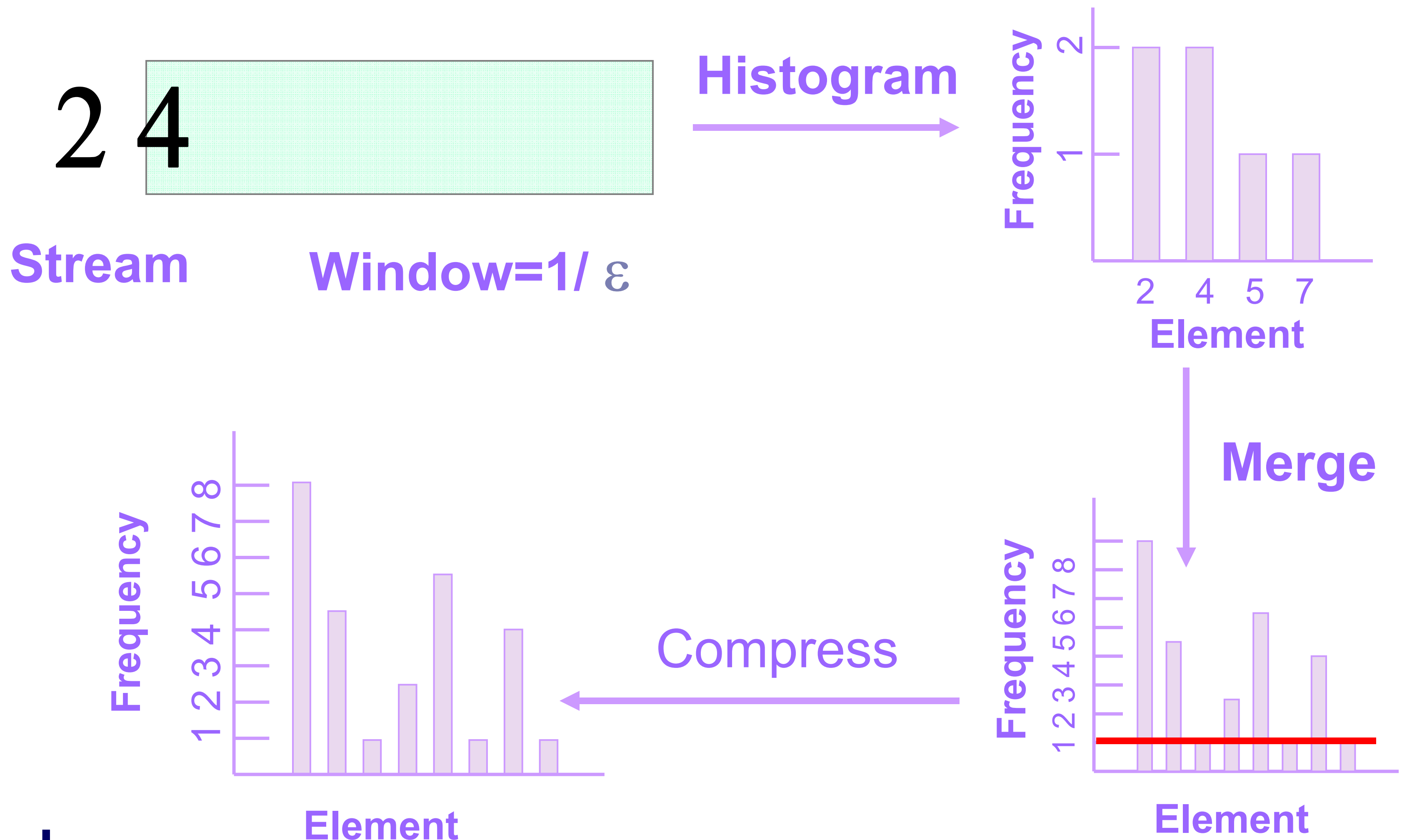
Super-Moore's Law Growth



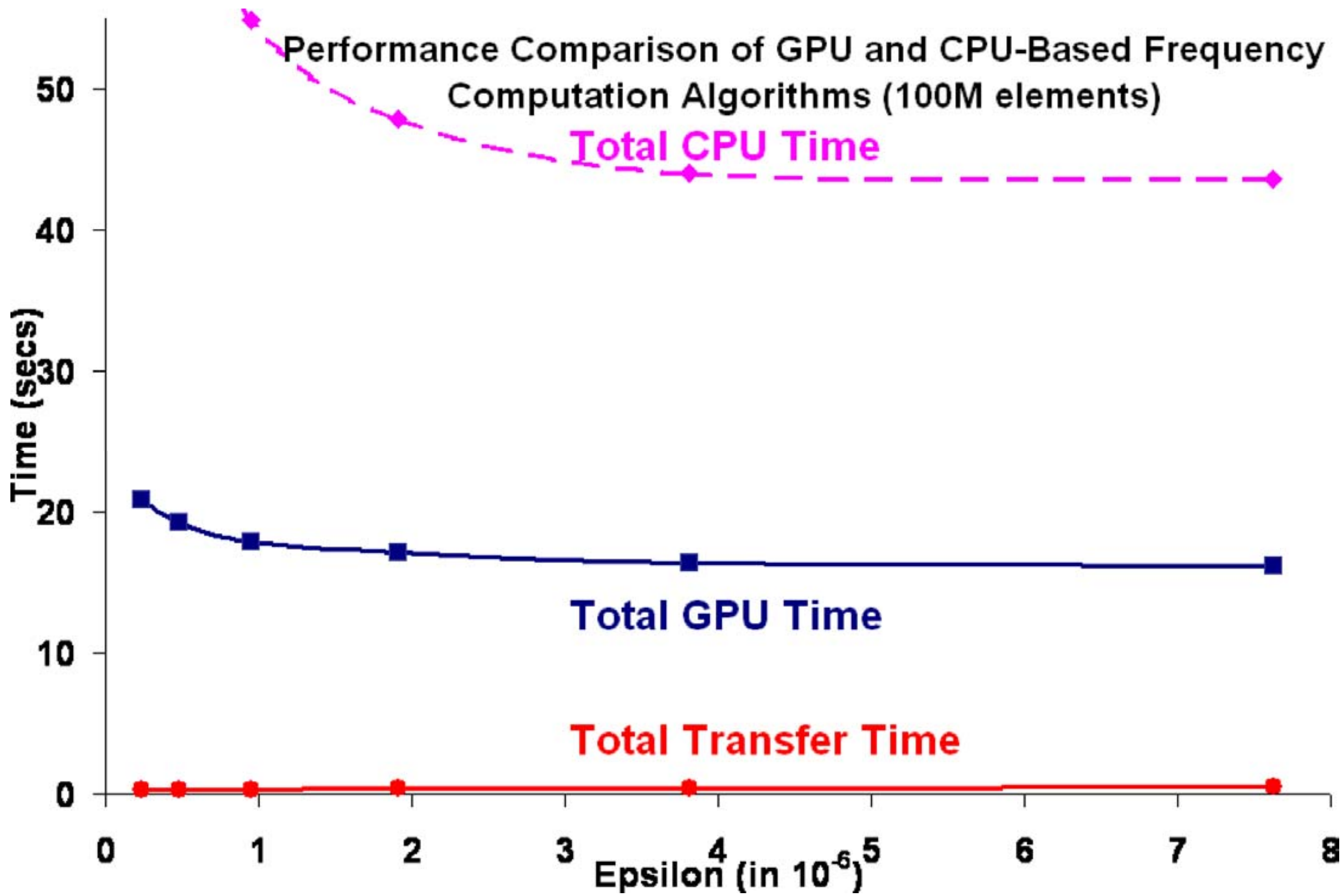
50 GB/s on a single GPU

Peak Performance: Effectively hide memory latency with 15 GOP/s

Frequency Estimation



Applications: Frequency Estimation



Quantile Estimation

6 4 2 5 7 2 4 $\xrightarrow{\text{Sort}}$ 2 2 4 4 5 6 7

Window

Sample (error=ε/2)

Merge Operation

3 5 6 7 9 11 14 merge with 4 8 10 12 13 16 19

3 4 5 6 7 8 9 10 11 12 13 14 16 19

error ε_1 error $\max(\varepsilon_1, \varepsilon_2)$ error ε_2

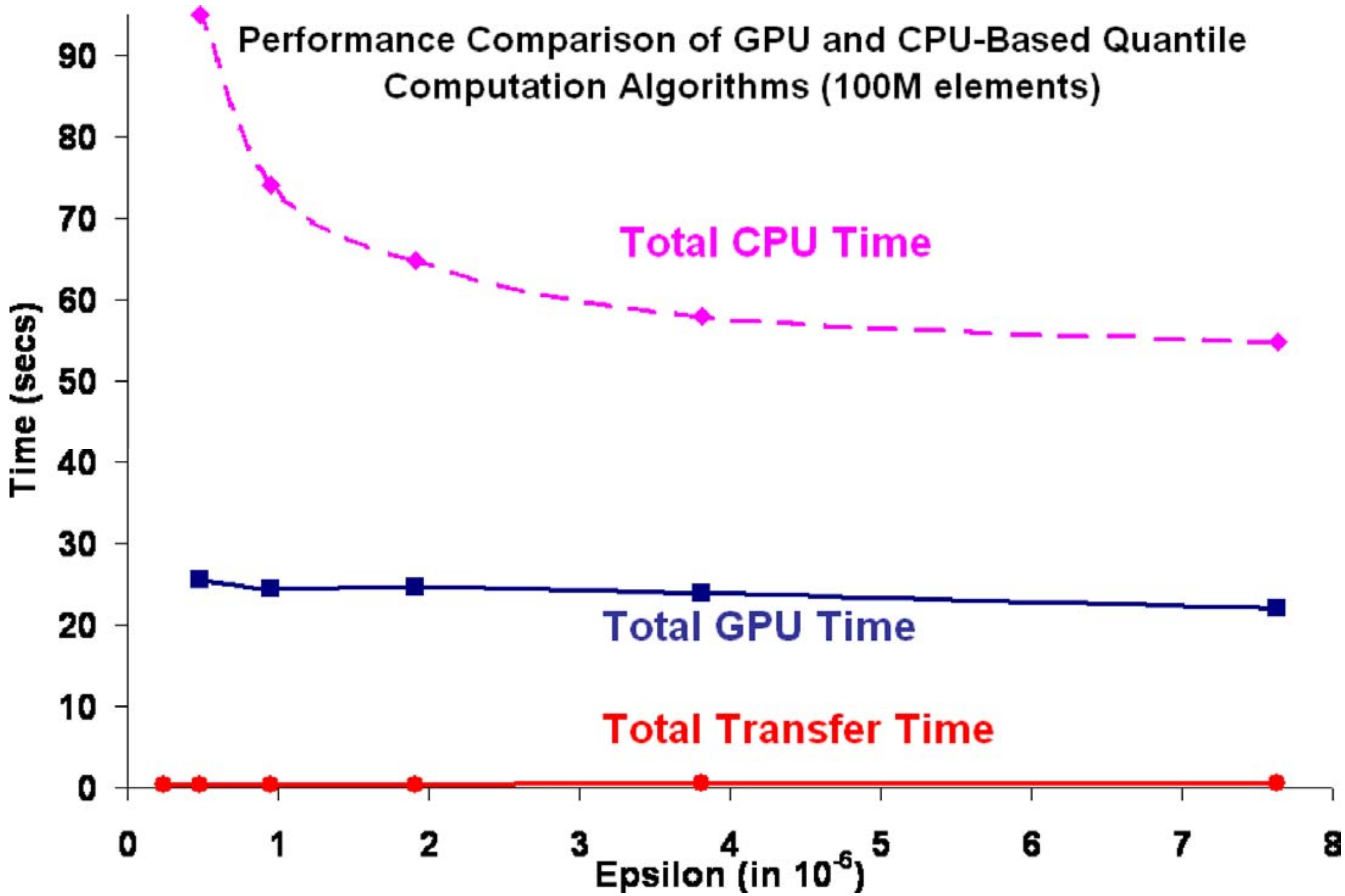
Compress Operation

3 5 6 7 9 11 14 $\xrightarrow[\text{B elements}]{\text{Uniformly sample}}$ 3 6 9 11

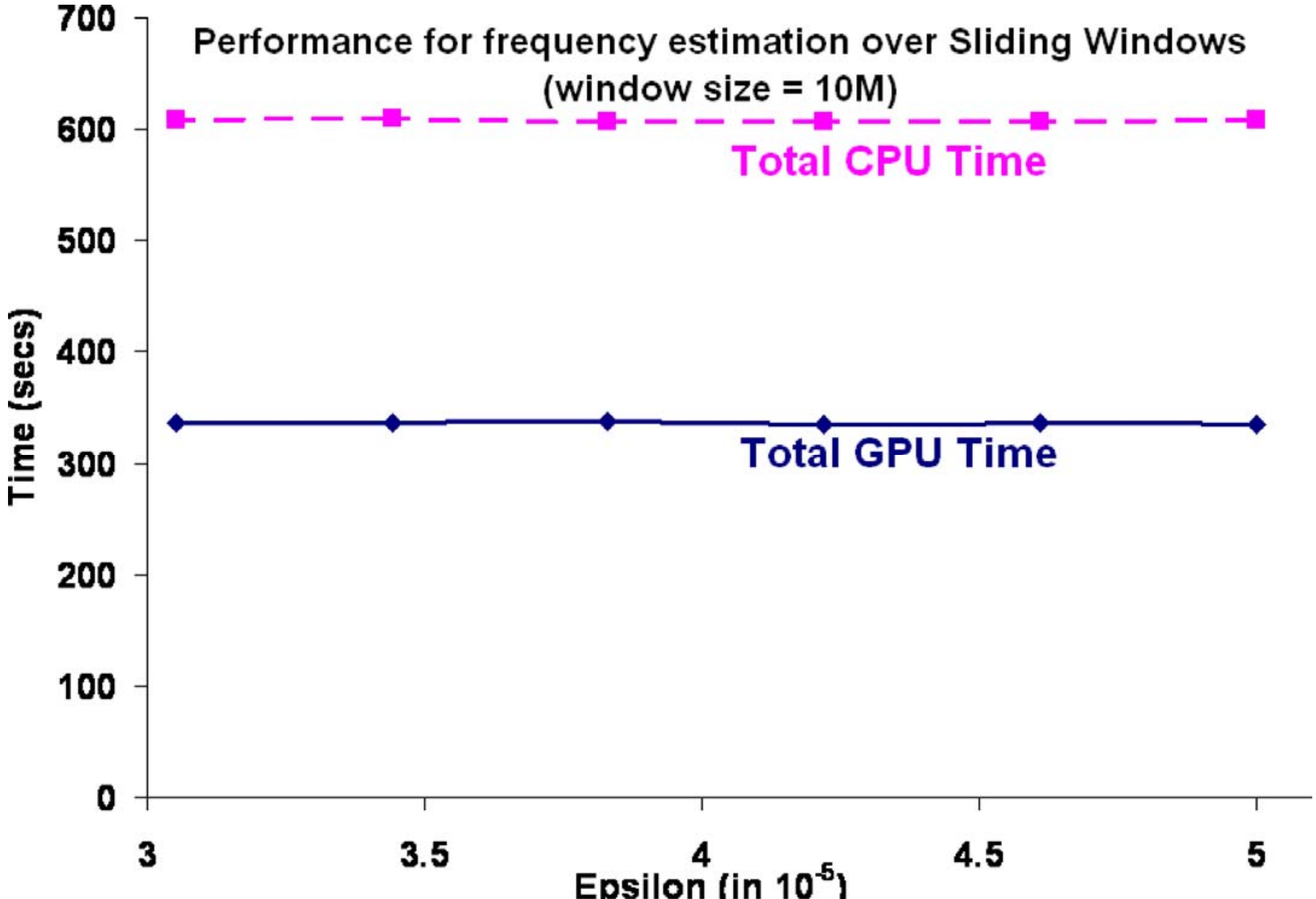
error ε

error <= ε+1/2B

Applications: Quantile Estimation



Applications: Sliding Windows



Advantages

- Sorting performed as stream operations entirely on GPUs
  - Uses specialized functionality of texture mapping and blending - **high performance**
- Low bandwidth requirement
  - <10% of total computation time

Outline

- Graphics Processor Overview
- Mapping Computation to GPUs
- Database and data mining applications
  - Database queries
  - Quantile and frequency queries
  - External memory sorting
  - Scientific computations
- Summary

N. Govindaraju, J. Gray, R. Kumar, D. Manocha,
Proc. Of ACM SIGMOD 06

External Memory Sorting

- Performed on Terabyte-scale databases
- Two phases algorithm [**Vitter01, Salzberg90, Nyberg94, Nyberg95**]
  - Limited main memory
  - First phase – partitions input file into large data chunks and writes sorted chunks known as "Runs"
  - Second phase – Merge the "Runs" to generate the sorted file

External Memory Sorting

- Performance mainly governed by I/O

  Salzberg Analysis: Given the main memory size M and the file size N, if the I/O read size per run is T in phase 2, external memory sorting achieves efficient I/O performance if and only if the run size R in phase 1 is given by $R \approx \sqrt{(TN)}$

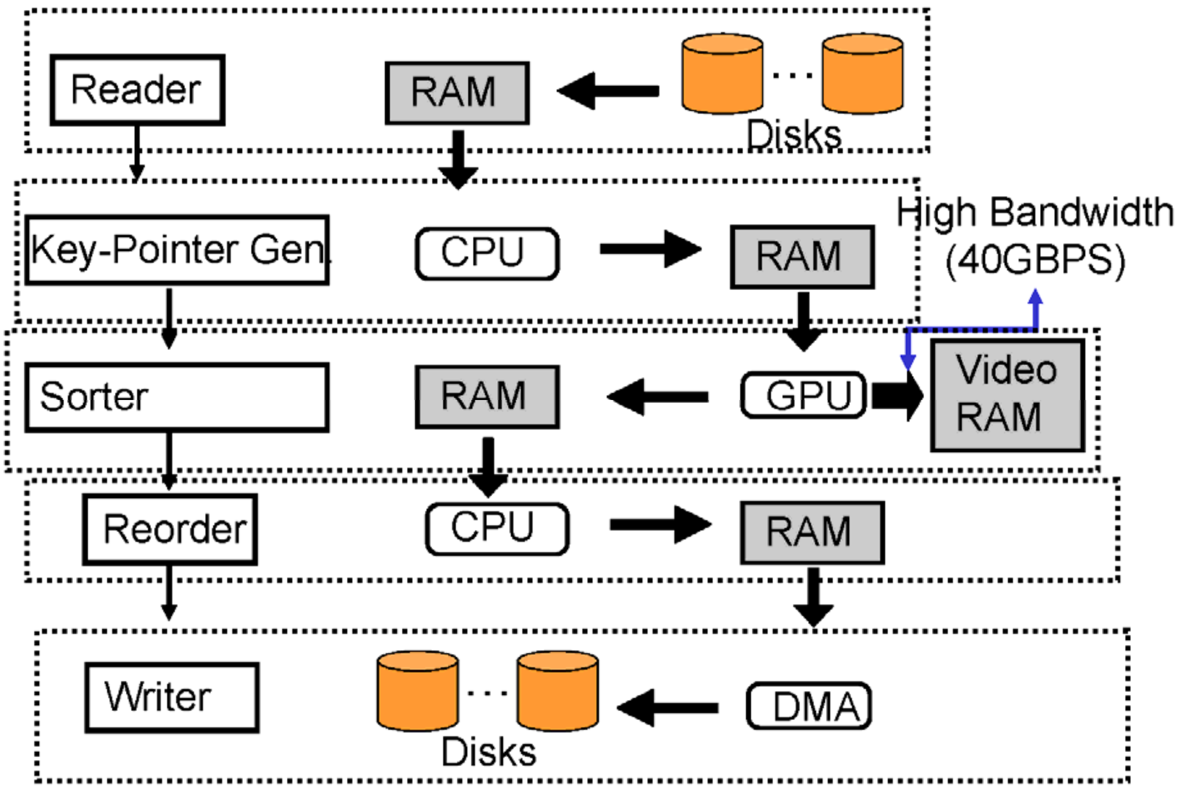
Salzberg Analysis

- If N=100GB, T=2MB, then
  R ≈ 230MB
- Large data sorting is inefficient on CPUs
  - R » CPU cache sizes – memory latency

External memory sorting

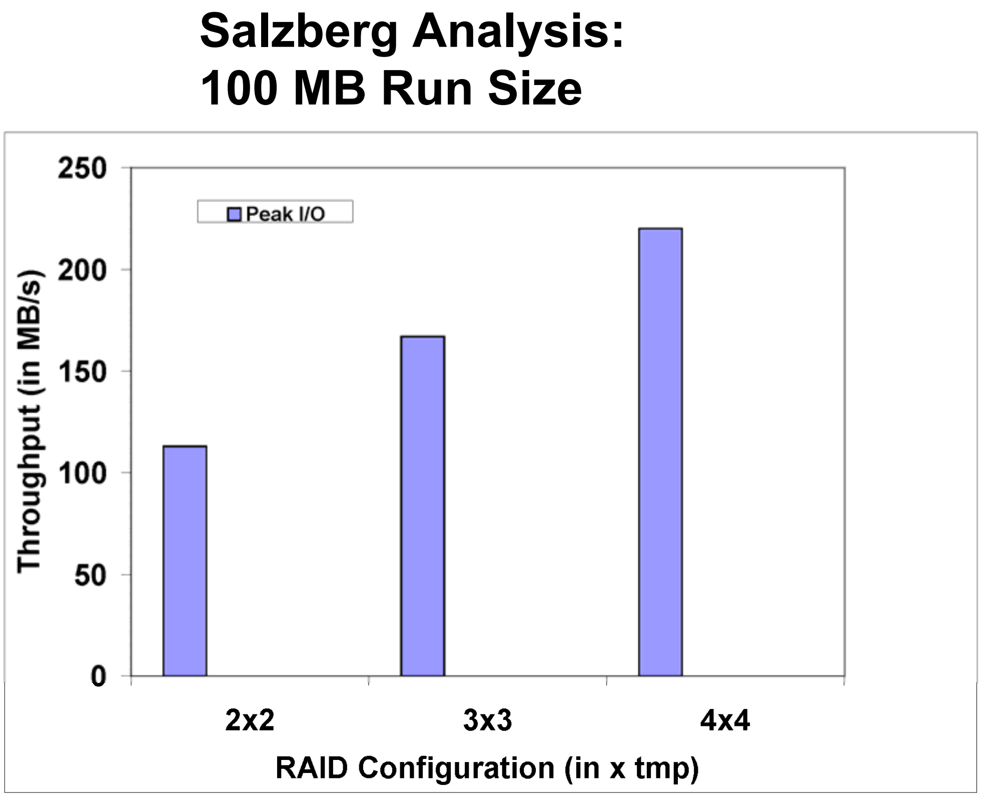
- External memory sorting on CPUs has low performance due to
  - High memory latency
  - Or low I/O performance
- Our algorithm
  - Sorts large data arrays on GPUs
  - Perform I/O operations in parallel on CPUs

GPUTeraSort

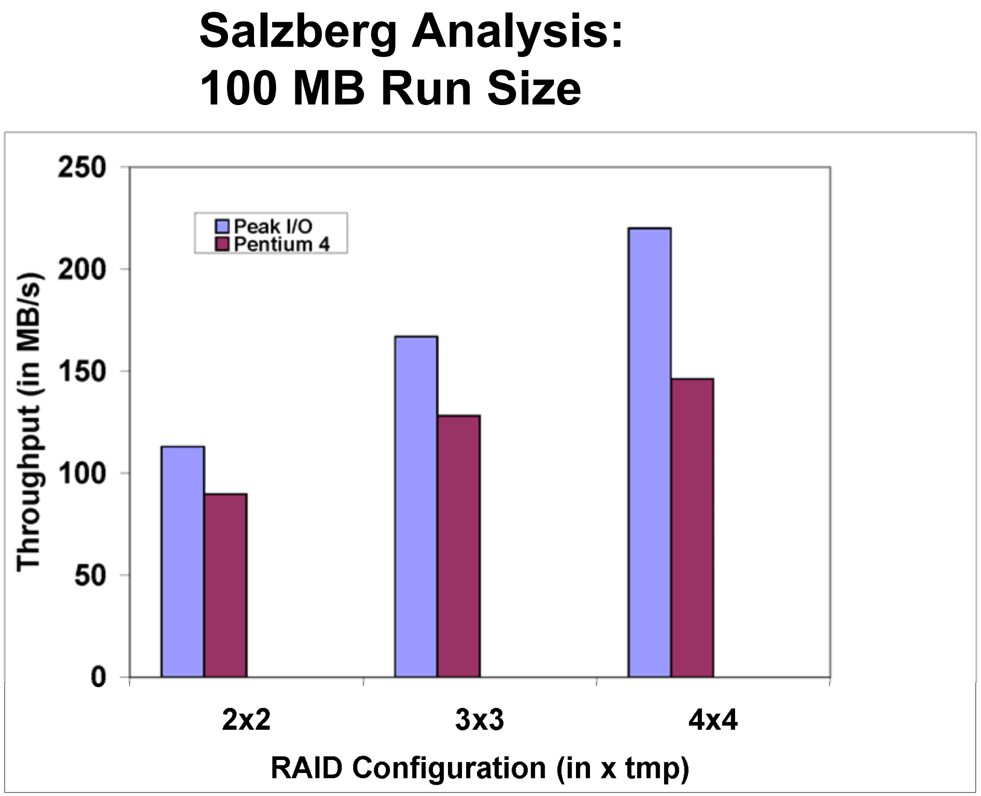


The UNIVERSITY *of* NORTH CAROLINA *at* CHAPEL HILL

I/O Performance



I/O Performance

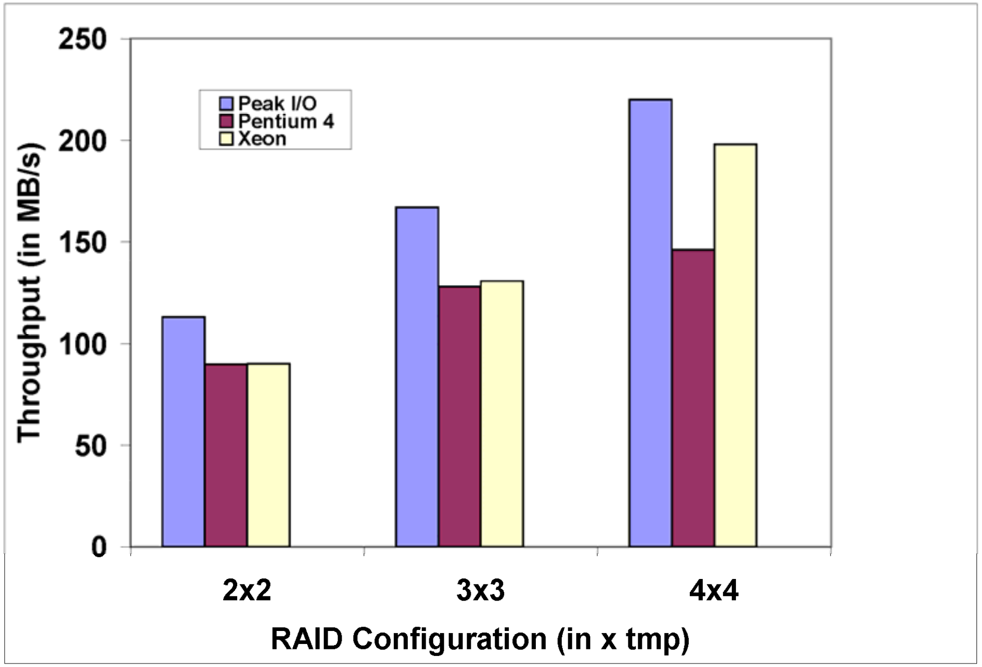


Pentium IV: 25MB Run Size

Less work and only 75% IO efficient!

I/O Performance

Salzberg Analysis:
100 MB Run Size

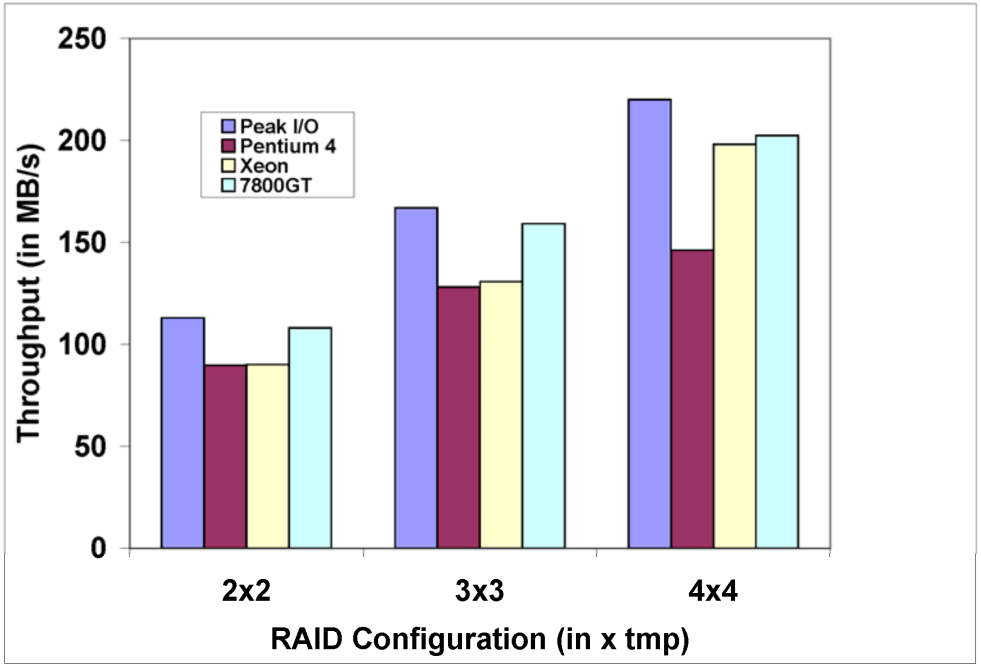


Dual 3.6 GHz Xeons: 25MB Run size

More cores, less work but only 85% IO efficient!

I/O Performance

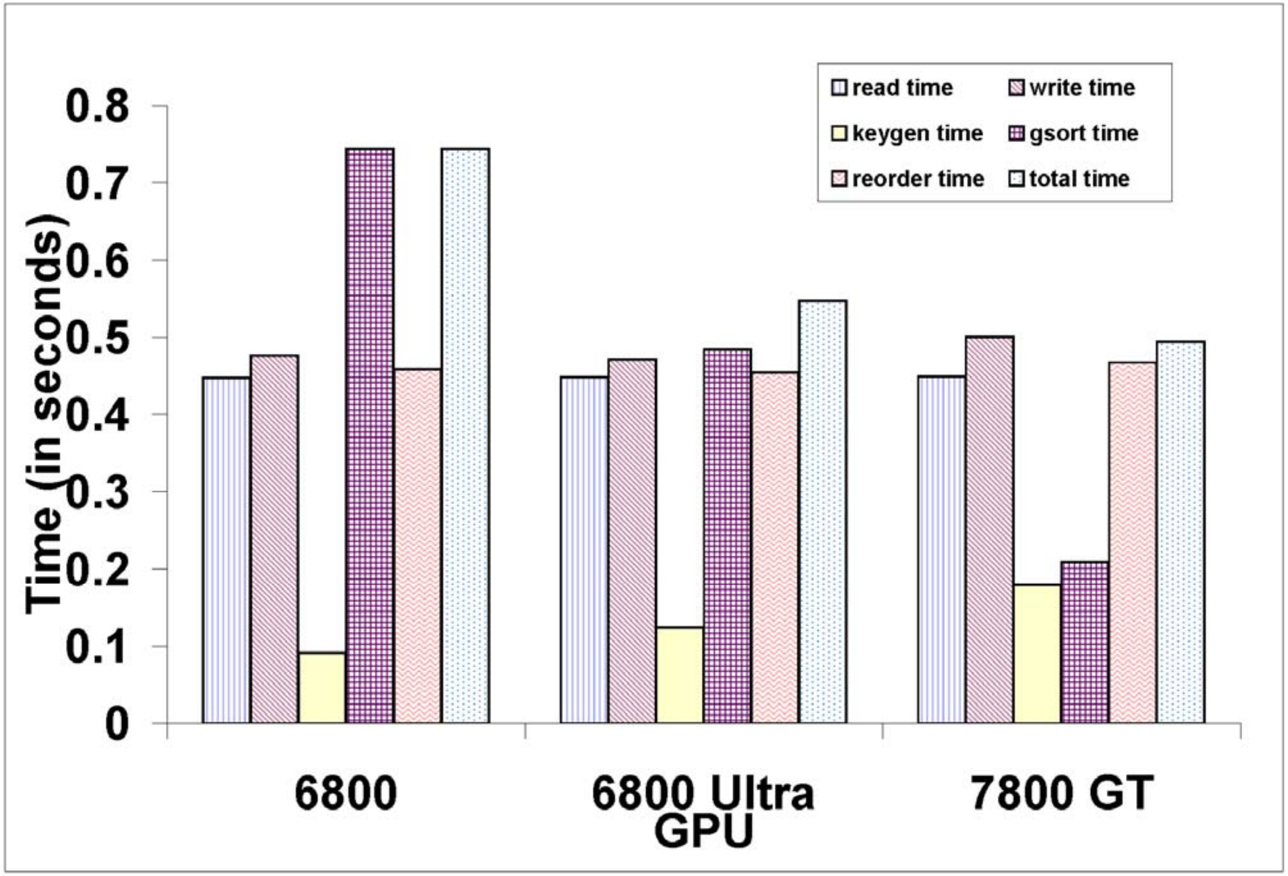
Salzberg Analysis:
100 MB Run Size



7800 GT: 100MB run size

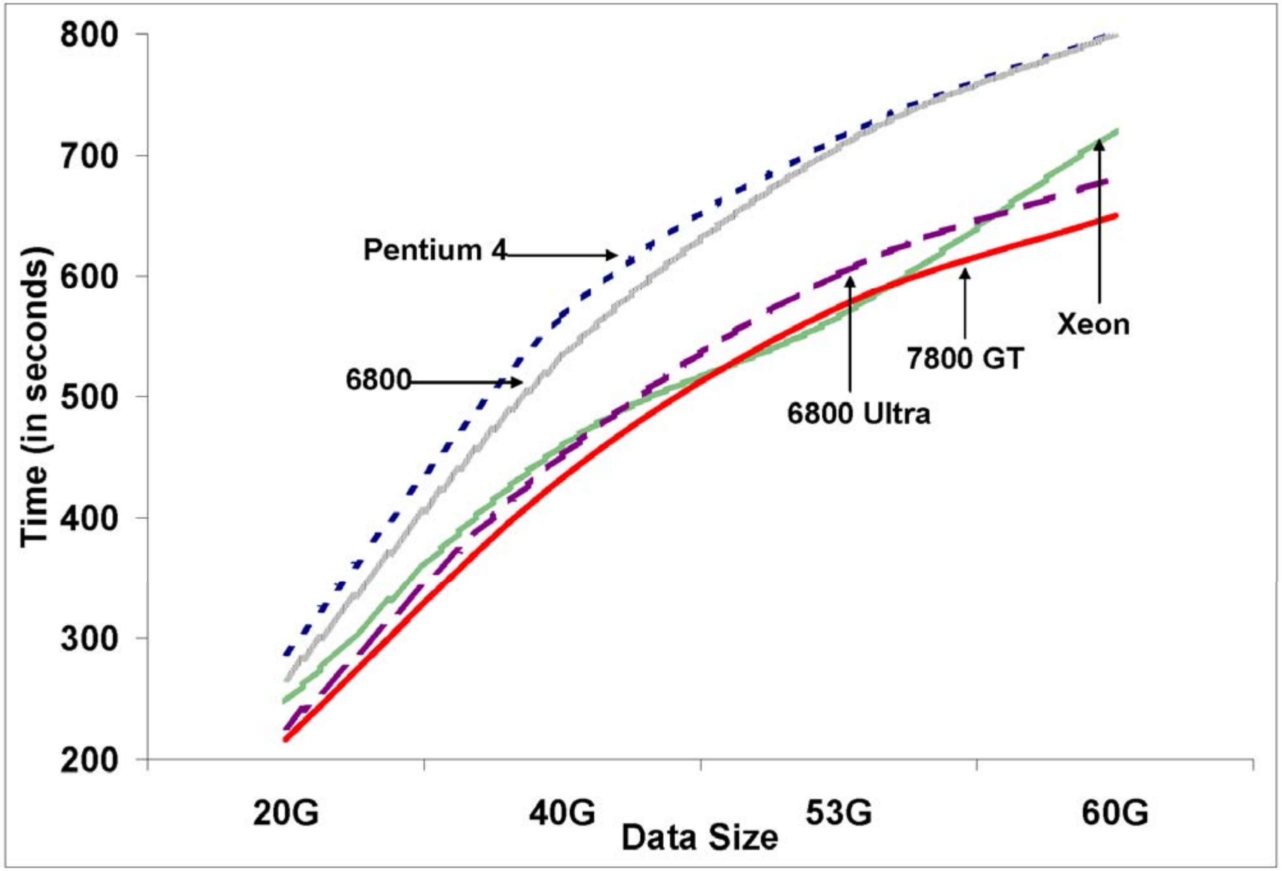
Ideal work, and 92% IO efficient with single CPU!

Task Parallelism



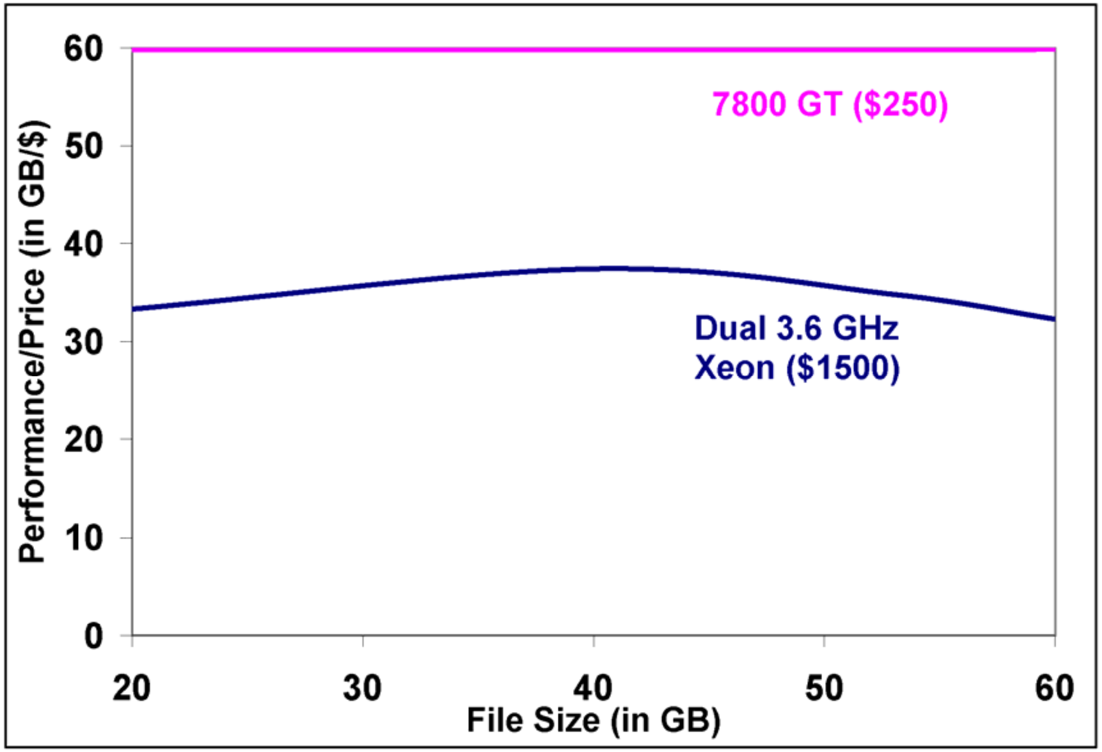
Performance limited by IO and memory

Overall Performance



Faster and more scalable than Dual Xeon processors (3.6 GHz)!

Performance/$



1.8x faster than current Terabyte sorter

World's best performance/$ system

N. Govindaraju, J. Gray, R. Kumar, D. Manocha, Proc. Of ACM SIGMOD 06

http://research.microsoft.com/barc/SortBenchMark/

Advantages

- Exploit high memory bandwidth on GPUs
  - Higher memory performance than CPU-based algorithms
- High I/O performance due to large run sizes

Advantages

- Offload work from CPUs
  - CPU cycles well-utilized for resource management
- Scalable solution for large databases
- Best performance/price solution for terabyte sorting

Applications

- Frequency estimation [Manku and Motwani 02]
- Quantile estimation [Greenwald and Khanna 01, 04]
- Sliding windows [Arasu and Manku 04]

Outline

- Graphics Processor Overview
- Mapping Computation to GPUs
- Database and data mining applications
  - Database queries
  - Quantile and frequency queries
  - External memory sorting
  - Scientific computations
- Summary

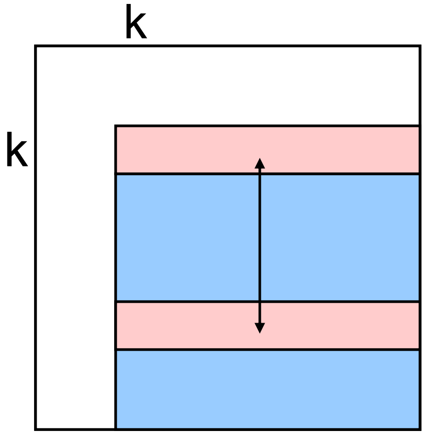
Scientific Computations

- Applied extensively in data mining algorithms
  - Least square fits, dimensionality reduction, classification etc.
- We present mapping of LU-decomposition on GPUs
  - Extensions to QR-decomposition, singular value decomposition (GPU-LAPACK)

N. Galoppo, N. Govindaraju, M. Henson, D. Manocha,
Proc. Of ACM SuperComputing 05

LU decomposition

- Sequence of row eliminations:
  - Scale and add: A(i,j) = A(i,j) – A(i,k) A(k,j)
  - Input data mapping: 2 distinct memory areas
  - No data dependencies
- Pivoting: row/column swap
  - Pointer-swap vs. data copy



LU decomposition

- Theoretical complexity: $2/3n^3 + O(n^2)$
- Performance <> Architecture
  - Order of operations
  - Data access (latency)
  - Memory bandwidth

Commodity CPUs

- LINPACK Benchmark:
  - Intel Pentium 4, 3.06 GHz: 2.88 GFLOPs/s

(Dongarra, Oct'05)

Motivation for LU-GPU

- LU decomposition maps well:
  - Stream program
  - Few data dependencies
- Pivoting
  - Parallel pivot search
  - Exploit large memory bandwidth

GPU based algorithms

- Data representation
- Algorithm mapping

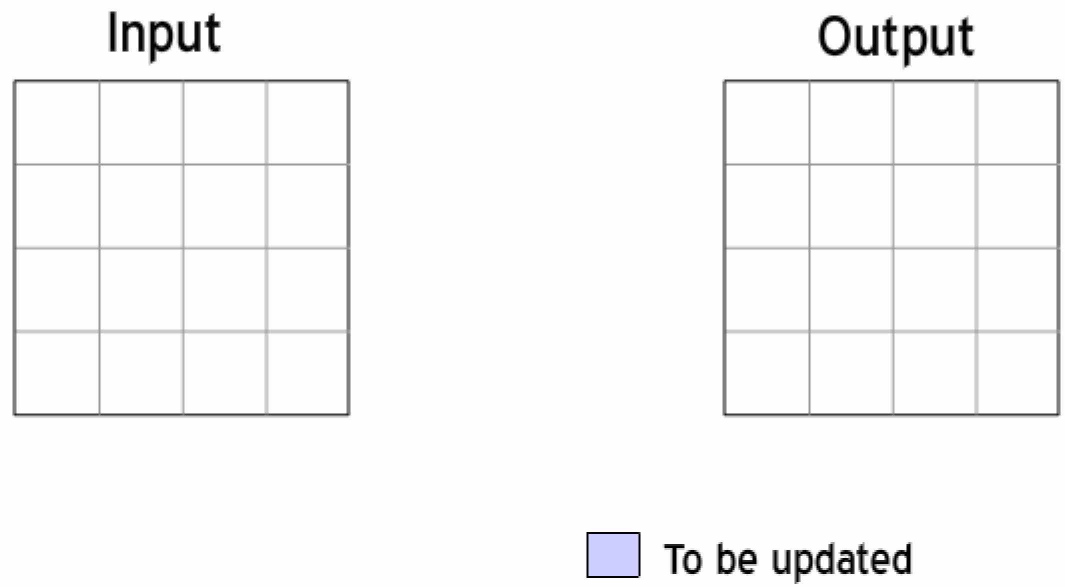
Data representation

- Matrix elements
  - 2D texture memory
  - One-to-one mapping
- Texture memory = on-board memory
  - Exploit bandwidth
  - Avoid CPU-GPU data transfer

Stream computation

- Rasterize quadrilaterals
  - Generates computation stream
  - Invokes SIMD units
  - Rasterization simulates blocking
- Rasterization pass = row elimination
- Alternating memory regions

Stream computation



Benchmarks

Commodity CPU

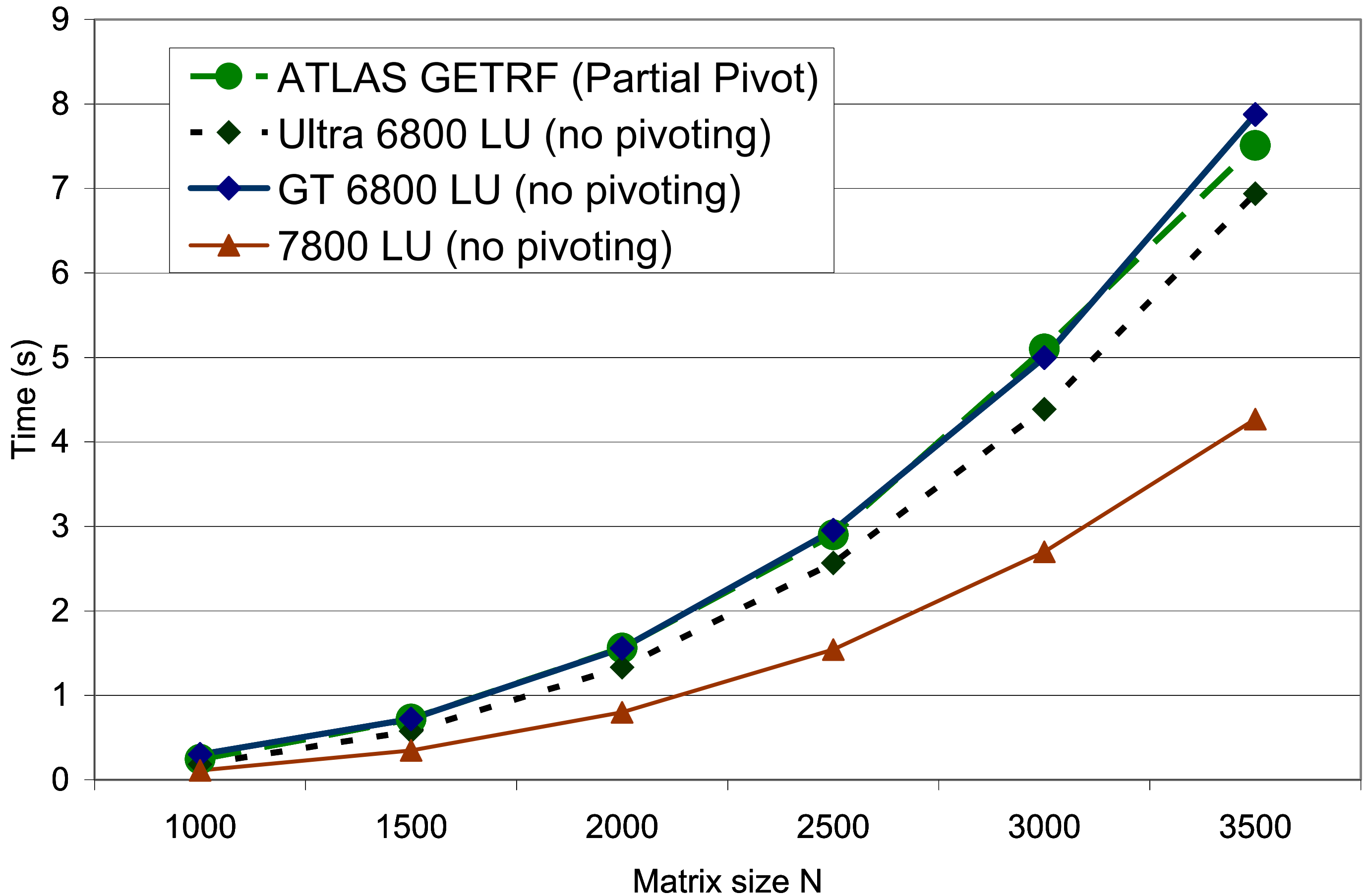
3.4 GHz Pentium IV with Hyper-Threading

1 MB L2 cache

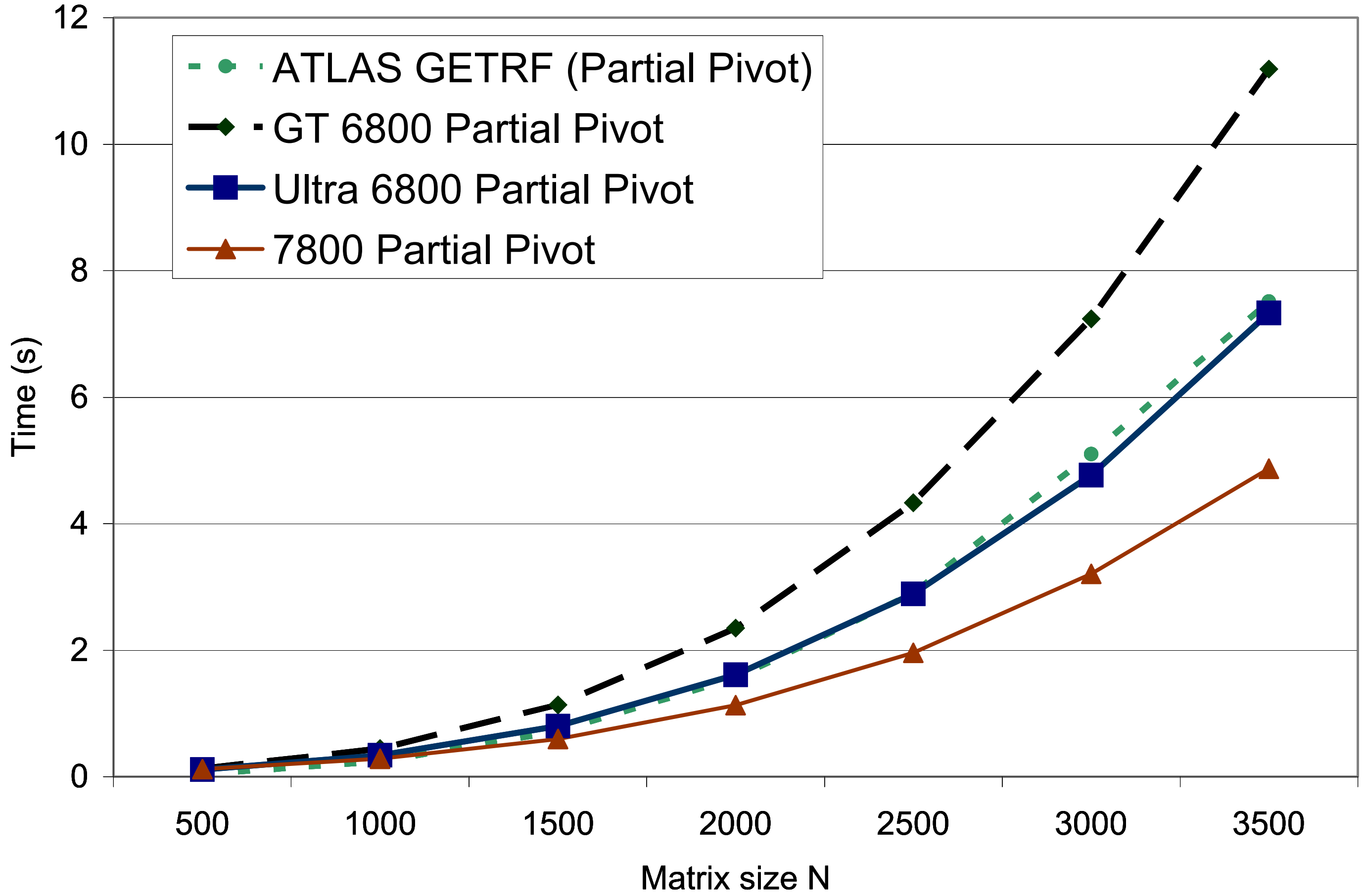
LAPACK getrf() (blocked algorithm, SSE-optimized ATLAS library)

GPU	SIMD units	Core clock	Memory	Memory clock
6800 GT	12	350 MHz	256 Mb	900 MHz
6800 Ultra	16	425 MHz	256 Mb	1100 MHz
7800 Ultra	24	430 MHz	256 Mb	1200 MHz

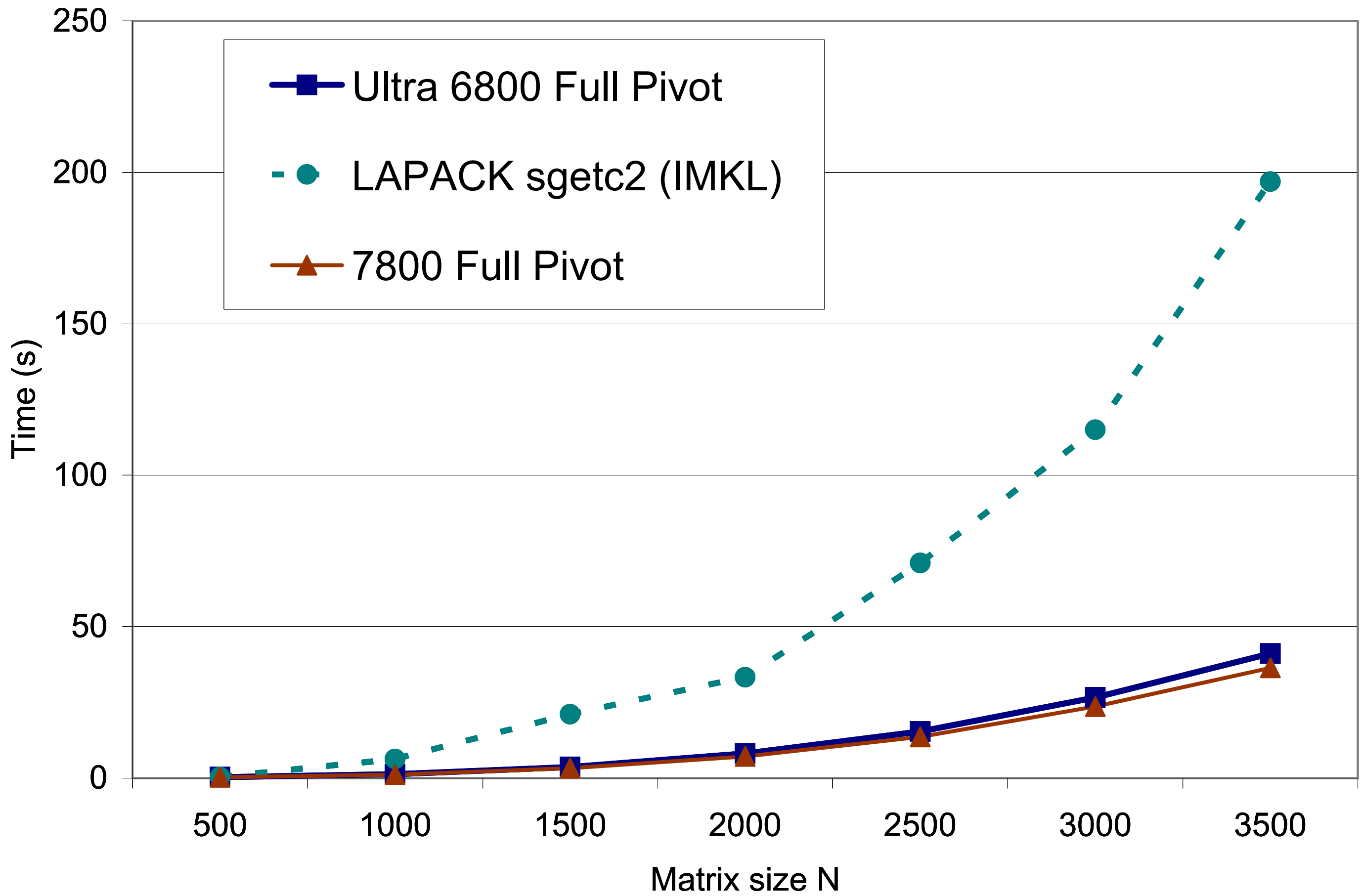
Results: No pivoting



Results: Partial pivoting



Results: Full Pivoting



LUGPU Library

http://gamma.cs.unc.edu/LUGPULIB

Outline

- Graphics Processor Overview
- Mapping Computation to GPUs
- Database and data mining applications
- Summary

Conclusions

Novel algorithms to perform

- Database management on GPUs
  - **Evaluation of predicates, boolean combinations of predicates, aggregations and join queries**
- Data streaming on GPUs
  - **Quantile and Frequency estimation**
- Terabyte data management
- Data mining applications
  - **LU decomposition, QR decomposition**

Conclusions

- Algorithms take into account GPU limitations
  - No data rearrangements
  - No frame buffer readbacks
- Preliminary comparisons with optimized CPU implementations is promising
- GPU as a useful co-processor

GPGP: GPU-based Algorithms

- Spatial Database computations
  - Sun, Agrawal, Abbadi 2003
  - Bandi, Sun, Agrawal, Abbadi 2004
- Data streaming
  - Buck et al. 04, McCool et al. 04
- Scientific computations
  - Bolz et al. 03, Kruger et al. 03
- Compilers
  - Brook-GPU (Stanford), Sh (U. Waterloo), Accelerator (Microsoft Research)
- …

More at http://www.gpgpu.org

Advantages

- Algorithms progress at GPU growth rate
- Offload CPU work
  - Streaming processor parallel to CPU
- Fast
  - Massive parallelism on GPUs
  - High memory bandwidth
- Commodity hardware!