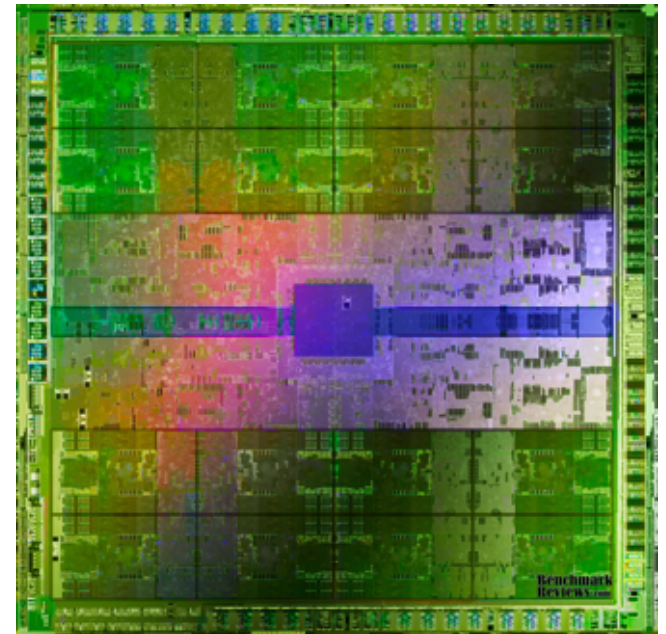# Large Scale Data Management on the GPU

Tim Kaldewey
Research Staff Member
IBM TJ Watson Research Center
tkaldew@us.ibm.com

Disclaimer

The author's views expressed in this presentation do not necessarily reflect the views of IBM.
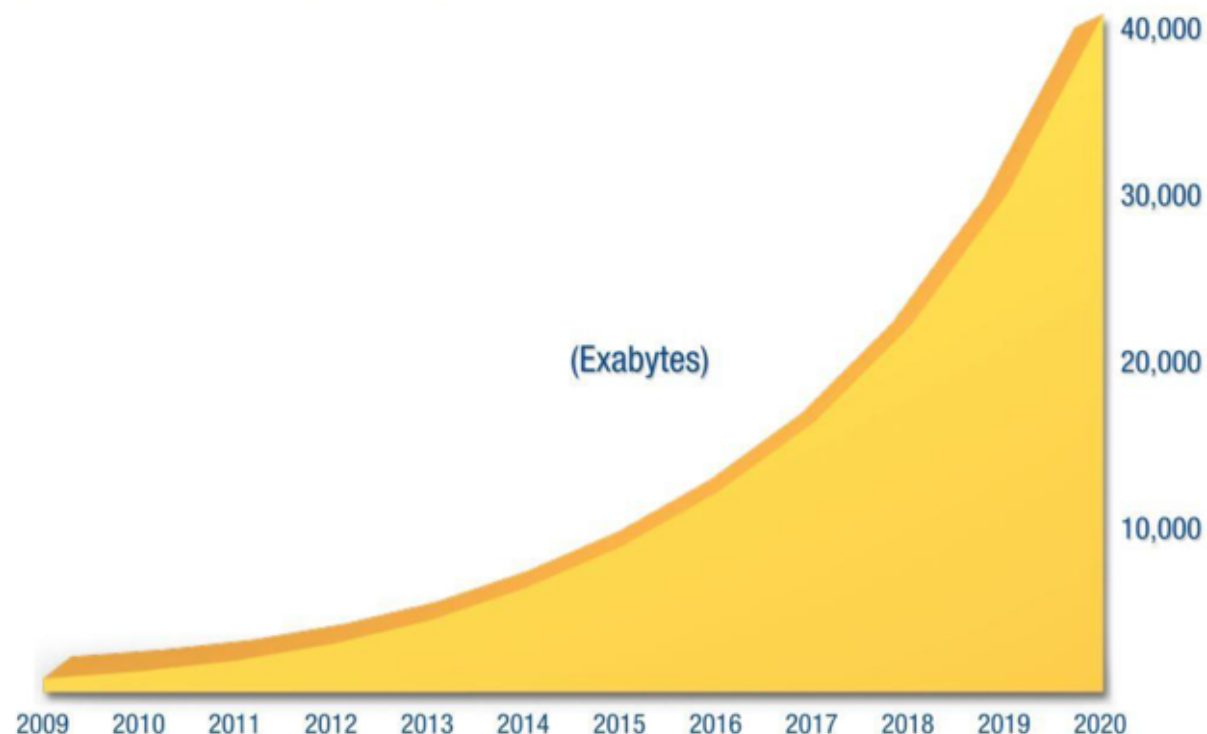
Acknowledgements

I would like to thank all my co-authors from IBM and my prior positions at Oracle and UCSC whose work I am showing in this presentation.

I would also like to thank Patrick Cozzi for inviting me to teach in this class multiple years in a row.

Managing Big (!) Data

The Digital Universe: 50-fold Growth from the Beginning of 2010 to the End of 2020

(Exabytes)

Source: IDC's Digital Universe Study, sponsored by EMC, December 2012

- Supercomputing is tackling the Exaflop

- Data Management got past the Exabyte range already

Where does all the data come from ?

Longer Data Retention Regulations

+

Unstructured Data: image, text, audio, video

+

Data maintenance for business analytics

+

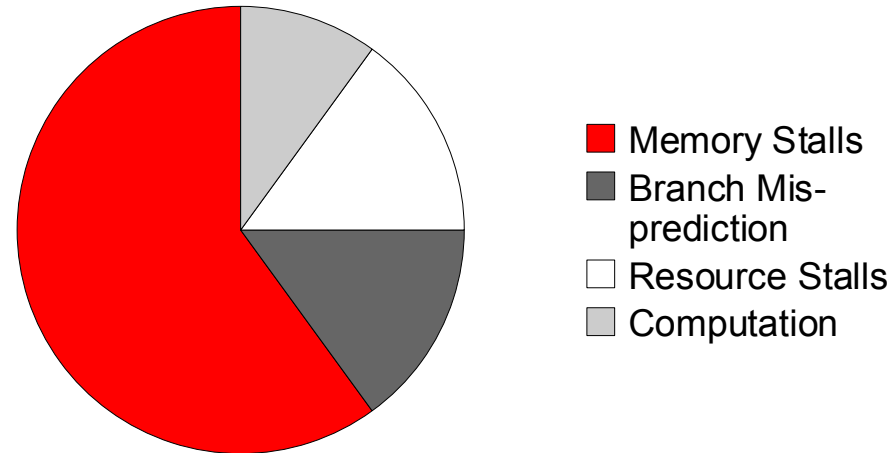Increased number of transactions

= DATA GROWTH

Source: Sybase Adaptive Server Enterprise Data Compression, SAP Sybase White paper 2012

- How to efficiently analyze/process rapidly increasing volumes of (business data) ?

- What is the problem ?
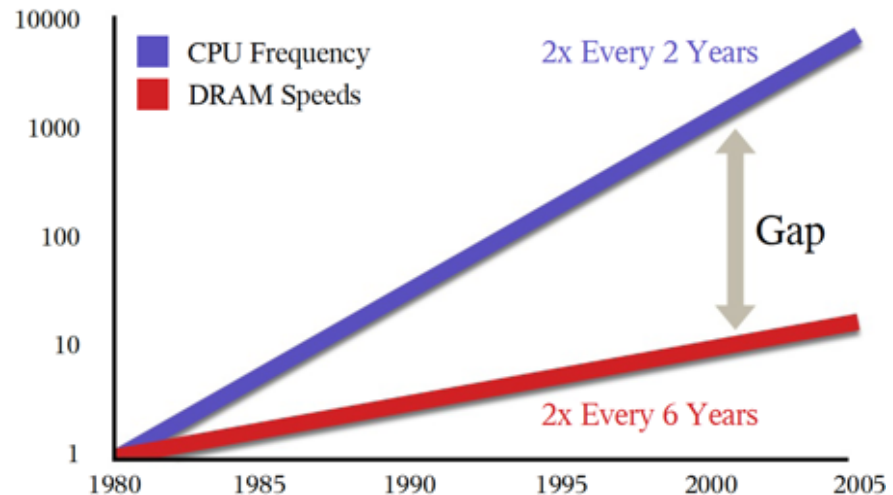 - Aren't processers getting faster every day ?

DB Performance – Where does time go ?

- CPU? I/O? Memory?
  - Select 10% of rows based on an index [1]

- Memory Stalls
- Branch Mis-prediction
- Resource Stalls
- Computation

- It's getting worse [2]

Relative Performance

- CPU Frequency
- DRAM Speeds

2x Every 2 Years
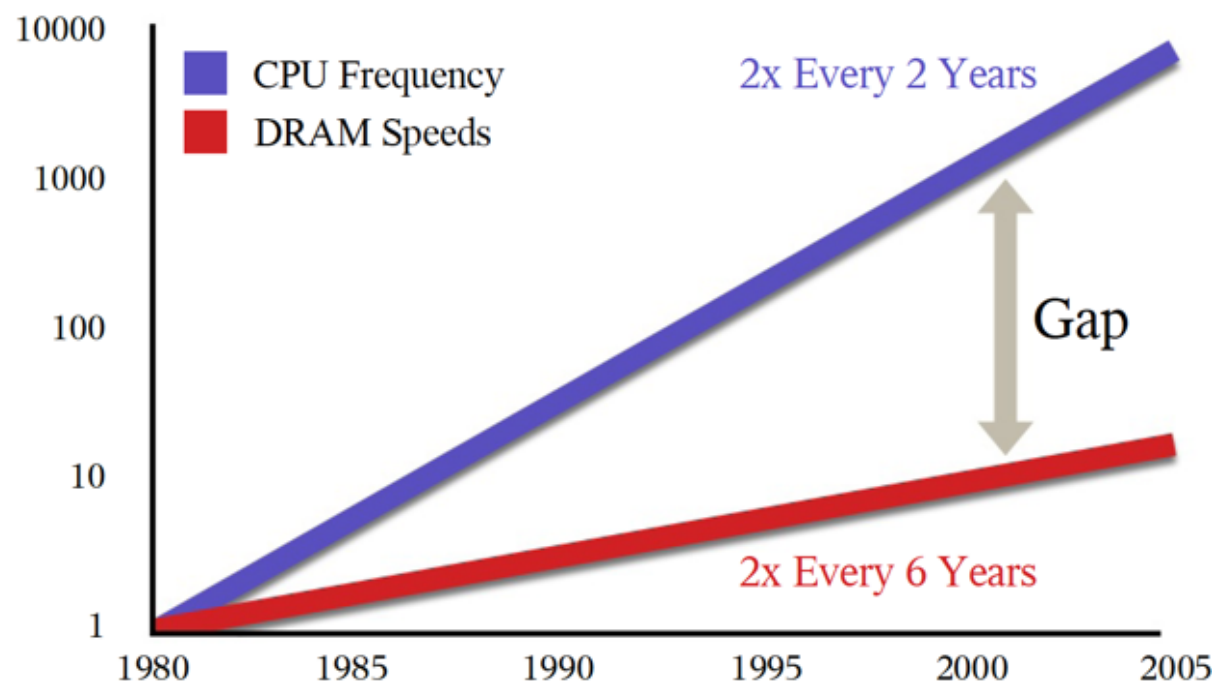
2x Every 6 Years

Gap

Sources:
[1] A. Ailamaki, et al. DBMSs on a modern processor: Where does time go? VLDB'99
[2] David Yen. Opening Doors to the MultiCore Era. MultiCore Expo 2006

The memory wall [1]

Relative Performance

CPU Frequency

DRAM Speeds

2x Every 2 Years

Gap

2x Every 6 Years

Sources:
[1] W.A.Wulf et al. Hitting the memory wall: implications of the obvious. SIGARCH - Computer Architecture News'95
[2] David Yen. Opening Doors to the MultiCore Era. MultiCore Expo 2006

The memory wall [1]

Relative Performance

2010:

- ■ CPU ~~Frequency~~ Cores — 2x Every 2 Years
- ■ DRAM ~~Speeds~~ Bandwidth
- ■ DRAM Latency

Gap

2x Every 6 Years

(Years: 1980, 1985, 1990, 1995, 2000, 2005; Performance axis: 1, 10, 100, 1000, 10000)
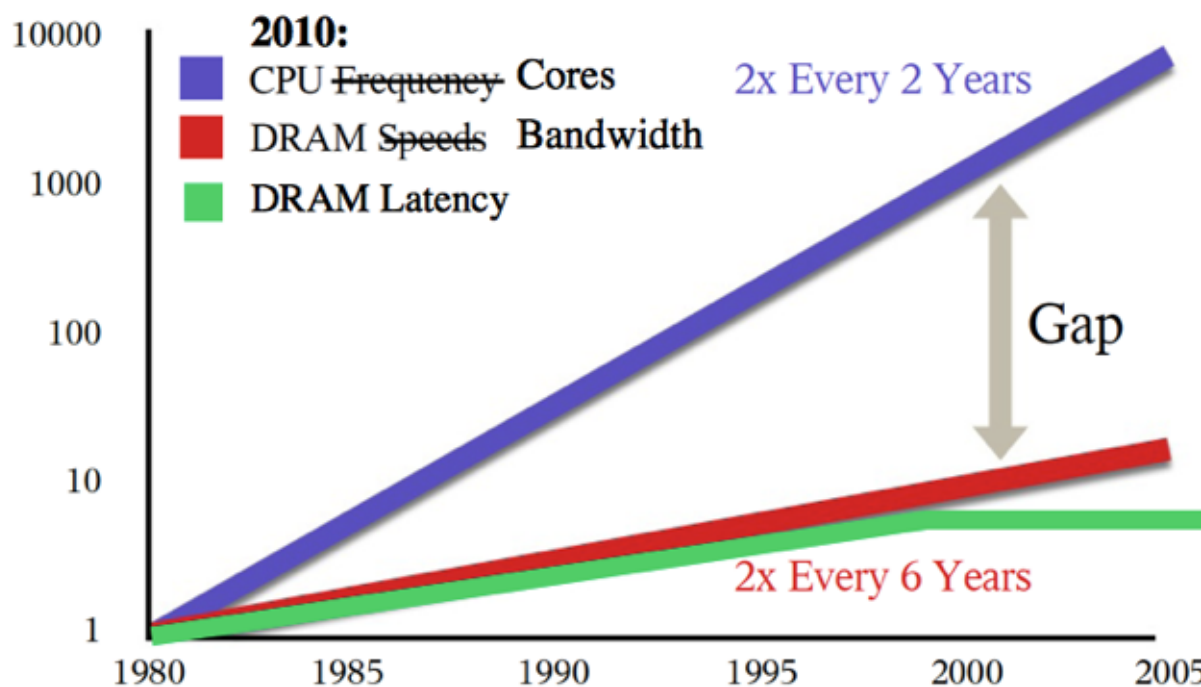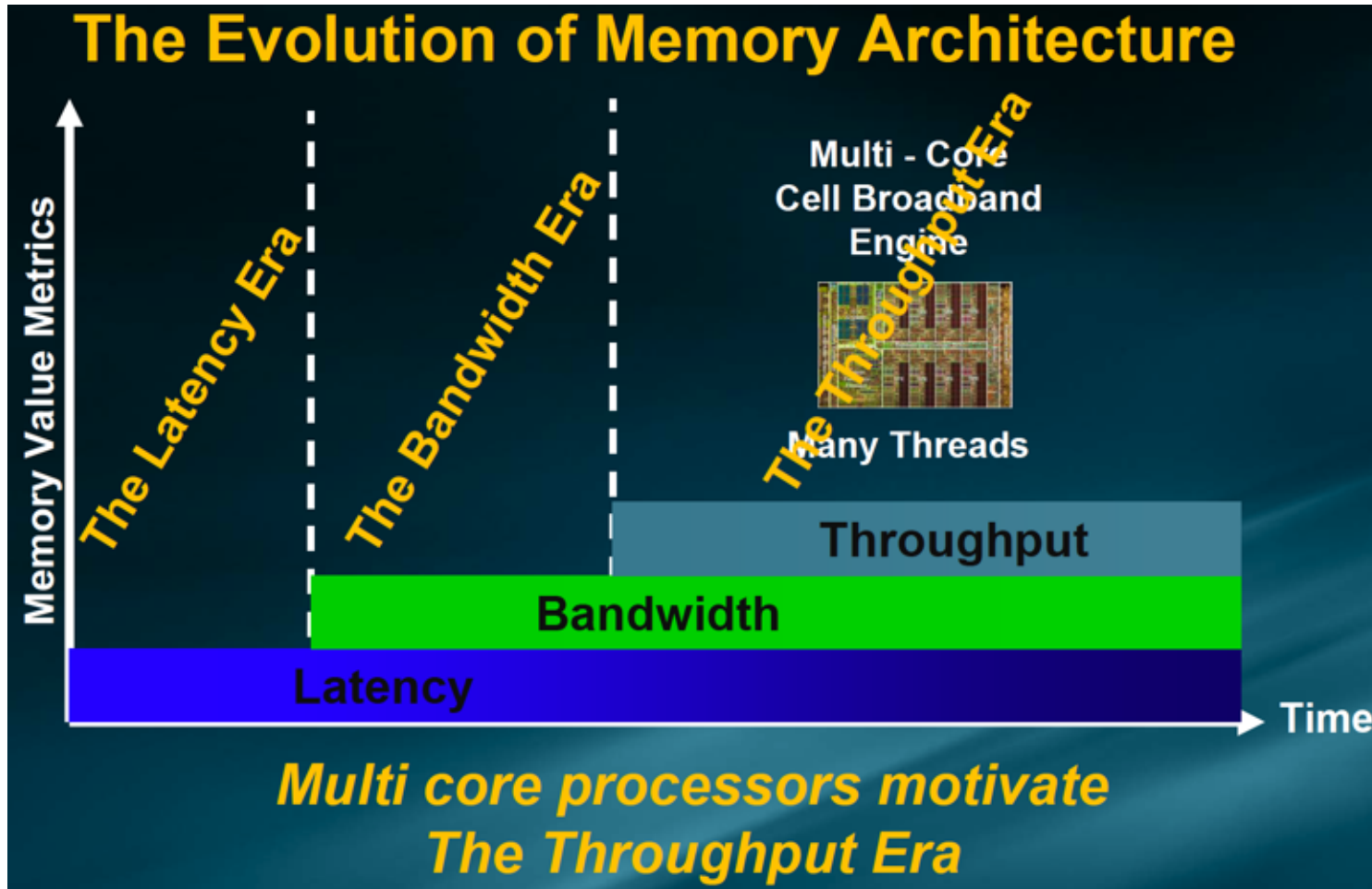
Sources:
[1] W.A.Wulf et al. Hitting the memory wall: implications of the obvious. SIGARCH - Computer Architecture News'95
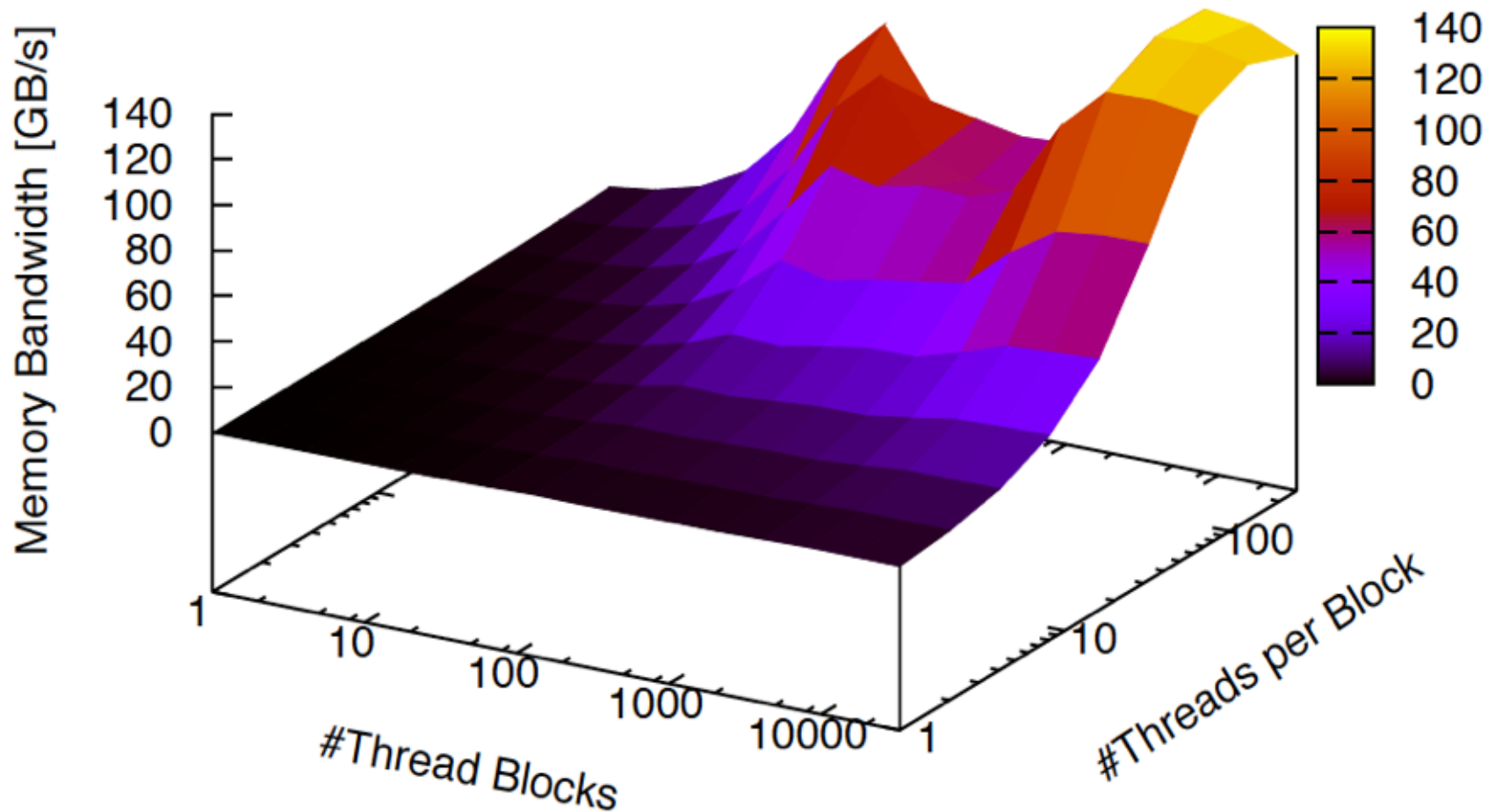[2] David Yen. Opening Doors to the MultiCore Era. MultiCore Expo 2006

Parallel Processing – How does it affect memory performance?

Source: Terabyte Bandwidth Initiative, Craig Hampel - Rambus, HotChips'08

GPU's are a Poster Child of the Throughput Era

Coalesced 32-bit read access on an nVidia GTX 285 1.5GHz, GDDR3 1.2GHZ.

- Required level of concurrency to reach peak performance depends on thread configuration.

Agenda

Large Scale Data Management on the GPU

- Why GPUs for information management workloads?
  - Search as an example

- Maximizing Device memory access performance
  - Coalescing
  - Thread configuration
  - Large(r) data sets

- GPU data transfers
  - Conventional
  - CUDA Streams
  - Zero Copy Access / Universal Virtual Addressing

- Search again
  - A naïve implementation
  - A novel parallel algorithm

Let's Pick a Simple, but Omnipresent Task … Search

- Honestly, how many times a day do you visit

Search – Memory Access Pattern

- Binary search (in a sorted list)
  - Choose Pivot Element at position *#elements / 2*
  - Compare if equal, larger, or smaller that search key (g)
    - Equal ➔ Done
    - Smaller ➔ proceed with lower half of the list
    - Larger ➔ proceed with upper half

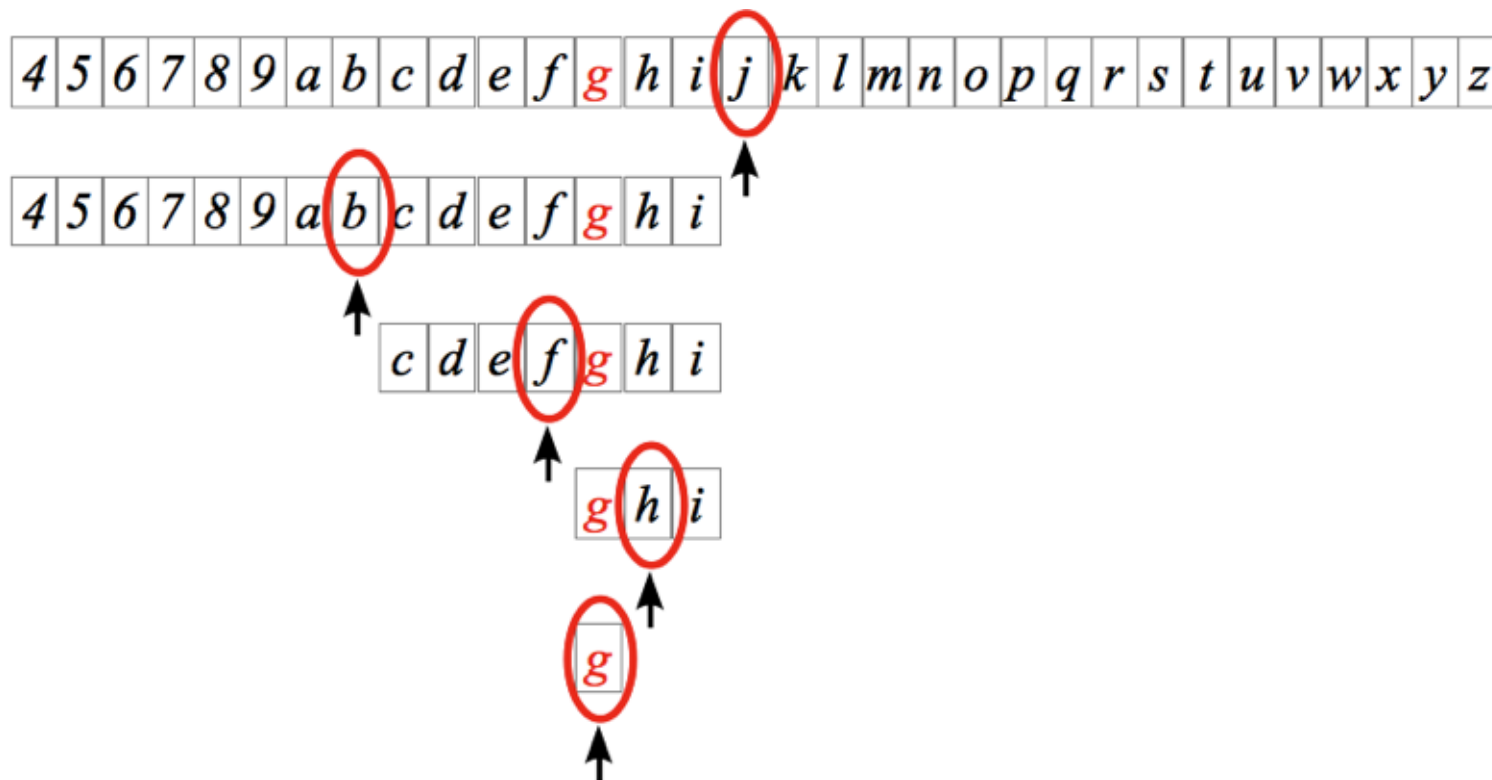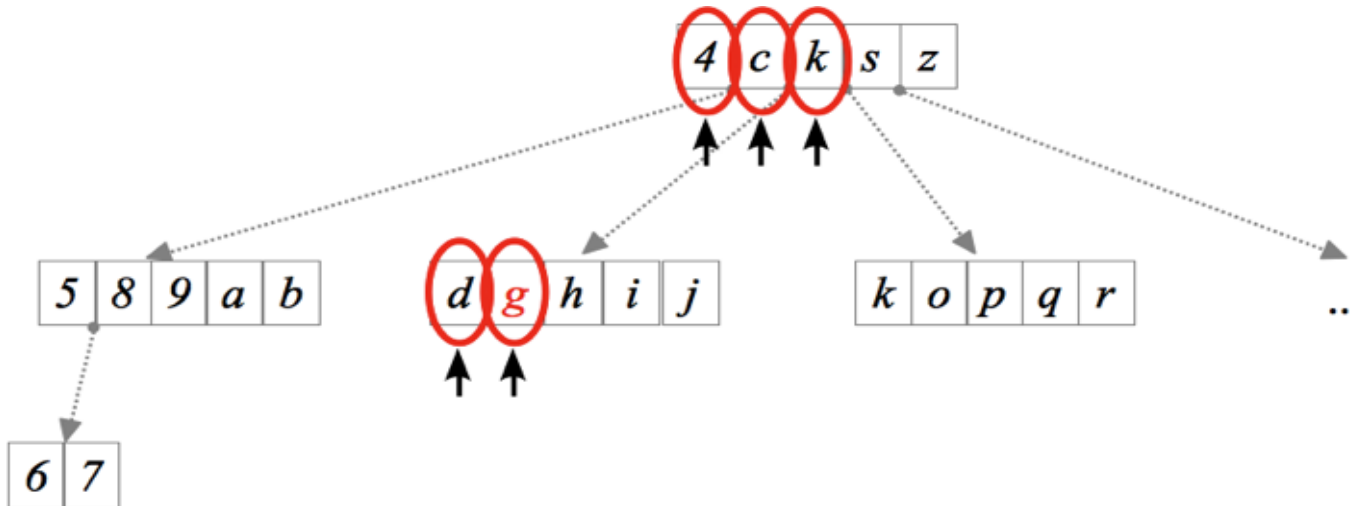| 4	5	6	7	8	9	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z

Search – Memory Access Pattern

- Binary search (in a sorted list)
  - Choose Pivot Element at position *#elements / 2*
  - Compare if equal, larger, or smaller that search key (g)
    - Equal ➔ Done
    - Smaller ➔ proceed with lower half of the list
    - Larger ➔ proceed with upper half

- Data dependent, quasi-random access pattern

Search – Memory Access Pattern

- B-trees group pivot elements = making access pattern more linear:

- How do you store B-trees?
- Still have quasi random access when traversing nodes!
- Overhead of maintaining trees?
  - Requires atomics if done in parallel …

GPU – Memory access performance

vs.

How do you achieve this?

	GPU (GTX580)	CPU (i7-2600)
Peak memory bandwidth [spec] [1]	179 GB/s	21 GB/s
Peak memory bandwidth [measured] [2]	153 GB/s	18 GB/s
Random access [measured] [2]	6.6 GB/s	0.8 GB/s
Compare and swap [measured] [3]	4.6 GB/s	0.4 GB/s

~ 1 order of magnitude

(1) Nvidia: 192.4×10^6 B/s $\approx$ 179.2 GB/s
(2) 64-bit accesses over 1 GB of device memory
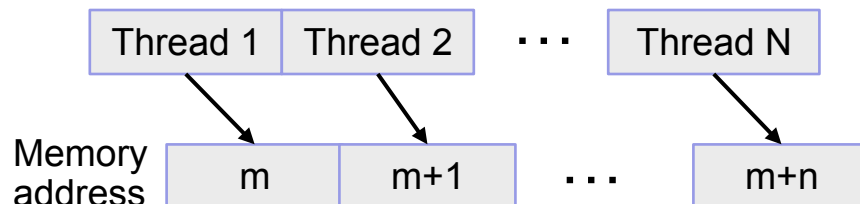(3) 64-bit compare-and-swap to random locations over 1 GB device memory

Large Scale Data Management on the GPU

- Why GPUs for information management workloads?
  - Search as an example

- Maximizing Device memory access performance
  - Coalescing
  - Thread configuration
  - Large(r) data sets

- GPU data transfers
  - Conventional
  - CUDA Streams
  - Zero Copy Access / Universal Virtual Addressing

- Search again
  - A naïve implementation
  - A novel parallel algorithm

Coalesced Memory access (reminder)

- Ideal memory access pattern is coalesced memory access
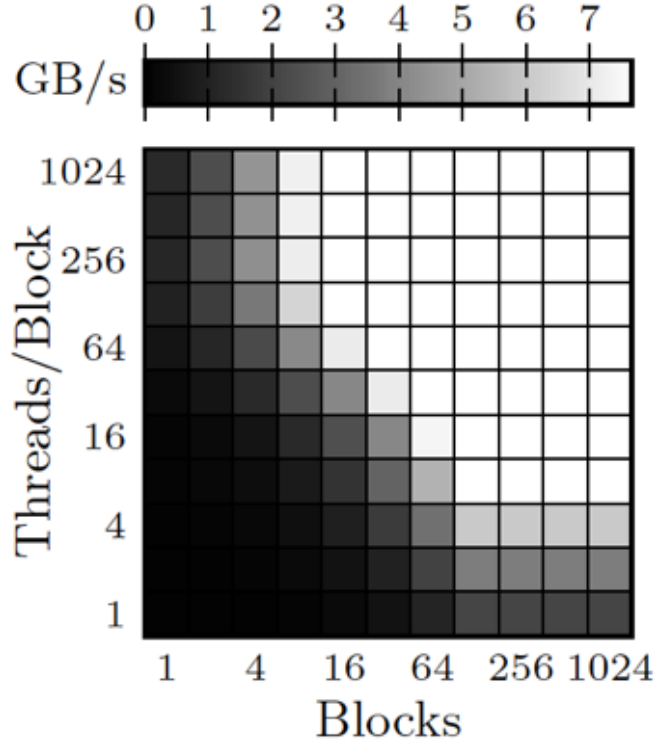 - Threads of a block/warp access consecutive memory addresses


```
for (position = blockIdx.x*blockDim.x+threadIdx.x;

     position < length;

     position += blockDim.x*gridDim.x){
x = list[position];

...
```

- This works great if we just read the entire data set, e.g. image processing

- Don't caches (Fermi and newer) make this obsolete?

- Random access if often the norm for information management workloads =(

"Fast" random access

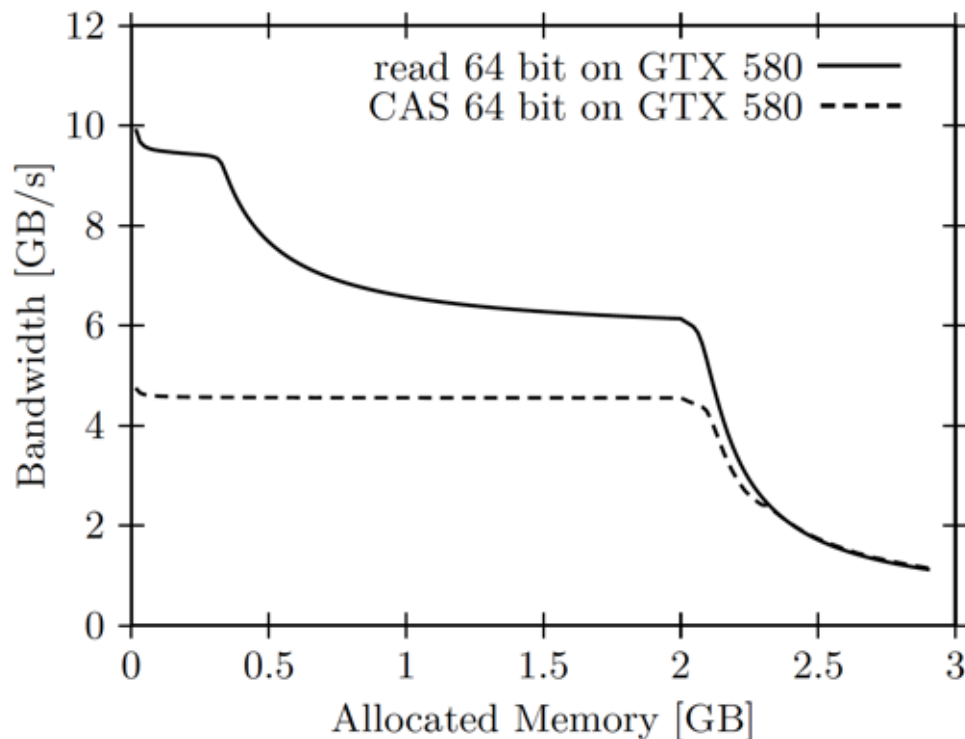- Performance depends on thread configuration

64-bit random access within 512 MB of device memory on GTX 580

- Assume that each access incurs device memory latency

- Use sufficient threads to hide memory latency

- How many?
  - Depends on specific GPU: memory interface, latency, …

Random access in large data sets

- Performance depends on data set size

64-bit random access within 512 MB of device memory on GTX 580

- Assume sufficient threads/blocks to hide memory latency
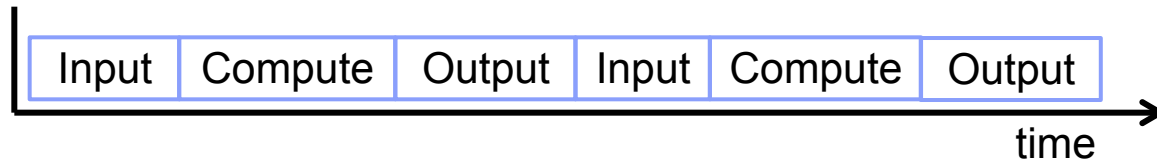
- What causes the 3 dips?

Agenda

Large Scale Data Management on the GPU

- Why GPUs for information management workloads?
  – Search as an example

- Maximizing Device memory access performance
  – Coalescing
  – Thread configuration
  – Large(r) data sets

- GPU data transfers
  – Conventional
  – CUDA Streams
  – Zero Copy Access / Universal Virtual Addressing

- Search again
  – A naïve implementation
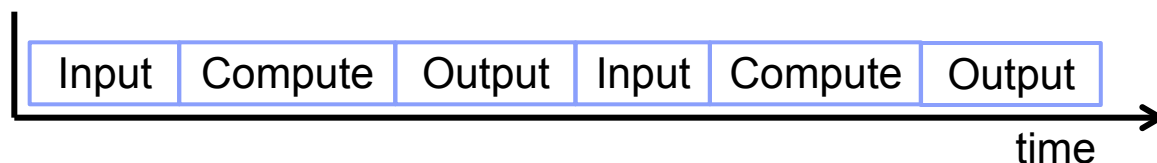 –A novel parallel algorithm

How to get data to the GPU (and back):

- Synchronous:

Input	Compute	Output	Input	Compute	Output

time

– Copy data to GPU, compute, copy results back to CPU

How to get data to the GPU (and back):

▪ Synchronous:

| Input | Compute | Output | Input | Compute | Output |

time

– Copy data to GPU, compute, copy results back to CPU

```
// make space for input data & results
cudaMalloc(...
// copy input data to GPU
cudaMemCpy(Input_Dev,Input_Host,size,cudaMemCpyHostToDevice);
// call compute Kernel
computeKernel<<dimGrid,dimBlock>>(Input_Dev, ...)
...
// copy results
cudaMemCpy(Output_Host,Ouput_dev,size,cudaMemCpyDeviceToHost);
```

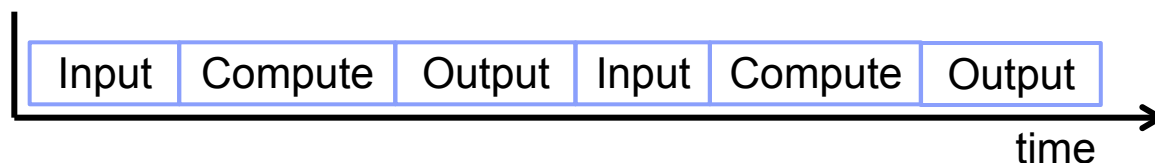# How to get data to the GPU (and back):

- Synchronous:

- Copy data to GPU, compute, copy results back to CPU

```
// make space for input data & results
cudaMalloc(...
// copy input data to GPU
cudaMemCpy(Input_Dev,Input_Host,size,cudaMemCpyHostToDevice);
// call compute Kernel
computeKernel<<dimGrid,dimBlock>>(Dev*...)
...
// copy results
cudaMemCpy(Output_Host,Ouput_dev,size,cudaMemCpyDeviceToHost);
```

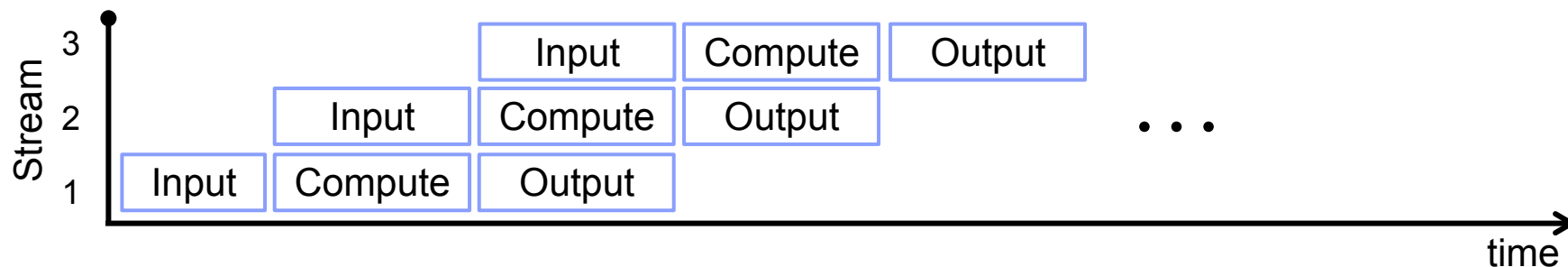- Common for most problems where the compute phase is long
- Caveats:
  - Data set size (input + output) limited by GPU memory size!
  - PCI-E idle during compute, cores idle during data transfer

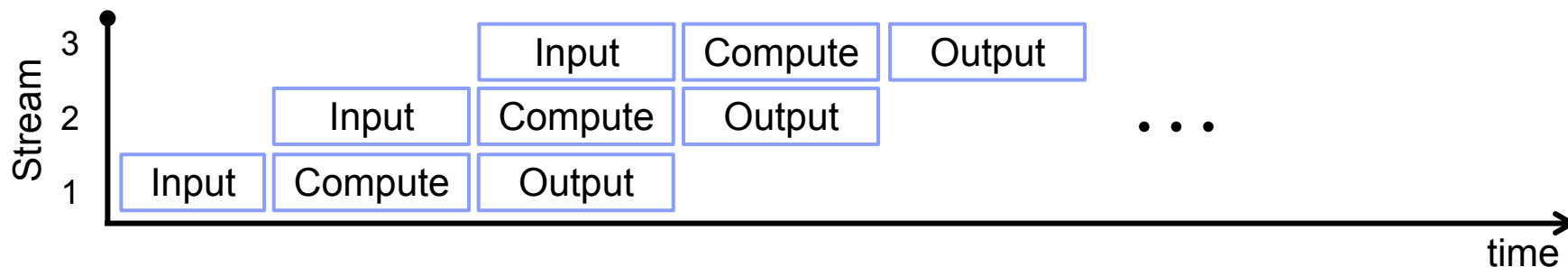# How to get data to the GPU (and back):

▪ Asynchronous data transfers, aka CUDA Streams:

– While one stream is executing a kernel, others can copy data in (and out)
  • Effectively overlap data copies with compute

How to get data to the GPU (and back):

- Asynchronous data transfers, aka CUDA Streams:

- While one stream is executing a kernel, others can copy data in (and out)
  - Effectively overlap data copies with compute

```
for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  cudaMemcpyAsync(&d_ipt[offset], &h_ipt[offset], streamBytes,
                  cudaMemCpyHostToDevice, stream[i]);
  computeKernel<<..., stream[i]>>(d_ipt, d_opt, offset);
  cudaMemcpyAsync(&h_opt[offset], &d_opt[offset], streamBytes,
                  cudaMemCpyDeviceToHost, stream[i]);
}
```

How to get data to the GPU (and back):
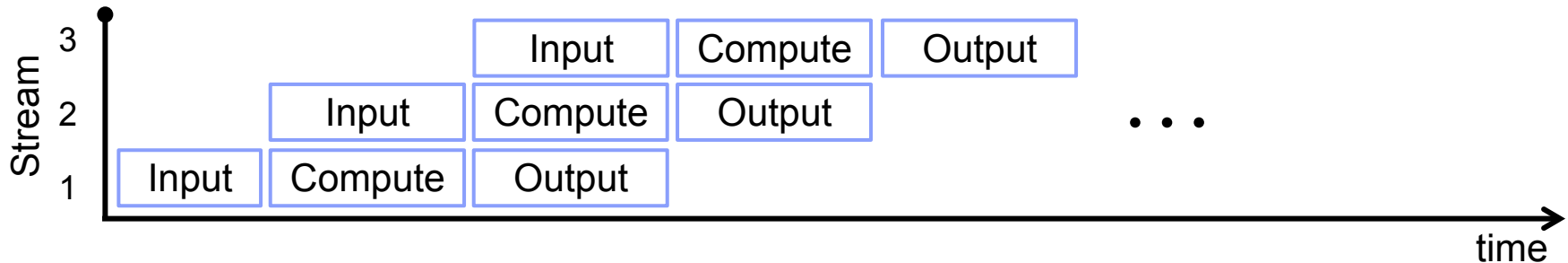
▪ Asynchronous data transfers, aka CUDA Streams:

– While one stream is executing a kernel, others can copy data in (and out)

  • Effectively overlap data copies with compute

```
for (int i = 0; i < nStreams; ++i) {
  int offset = i * streamSize;
  cudaMemcpyAsync(&d_ipt[offset], &h_ipt[offset], streamBytes,
                  cudaMemCpyHostToDevice, stream[i]);
  computeKernel<<..., stream[i]>>(d_ipt, d_opt, offset);
  cudaMemcpyAsync(&h_opt[offset], &d_opt[offset], streamBytes,
                  cudaMemCpyDeviceToHost, stream[i]);
}
```

– If data transfers faster than compute, 100% time available for compute
– Caveats:

  • 2 * Input set + 2 * output(results) < GPU memory size!
  • What if arithmetic intensity is low (like most database operations)?

How to get data to the GPU (and back):

- Zero Copy Access (ZCA)
  - Allows the GPU to access CPU memory "directly", i.e., without placing a copy in GPU in device memory:
    1. CPU thread pins memory page & obtains phys. address
    2. Function call contains a pointer to the phys. address
    3. GPU(CUDA) threads execute load/store instructions

How to get data to the GPU (and back):

- Zero Copy Access (ZCA)
  1. CPU thread pins memory page & creates pointer to phys. address

```
cudaHostRegister(h_ipt,ipt_size,cudaHostRegisterMapped);
cudaHostRegister(h_opt,opt_size,cudaHostRegisterMapped);
cudaHostGetDevicePointer(&ipt_ptr,h_ipt,0);
cudaHostGetDevicePointer(&opt_ptr,h_opt,0);
```

Zero Copy Access & Virtual memory management

Process view

Virtual
address:

0

4KB
page

4095

4096

8191

8192

12287

12288

Physical Memory

Physical
address:

4096

8191
8192

12287
12288

other
process

524288

Disk

- OS provides virtual to physical address translation

- Memory pages (usually 4KB) can be physically stored on disk

- GPU kernel knows nothing about virtual memory

Zero Copy Access & Virtual memory management

Process view

Virtual
address:

0

4KB
page

4095
4096

8191
8192

12287
12288

Physical Memory

Physical
address:

4096

8191
8192

12287
12288

other
process

524288

Disk

- OS provides virtual to physical address translation
- Memory pages (usually 4KB) can be physically stored on disk
- GPU kernel knows nothing about virtual memory
- `cudaHostRegister()` makes a call to **mlock()** which marks page(s) as not page-able and assures pages are contiguous
- Ensures that page(s) will ALWAYS be present in **physical** memory
- cudaHostGetDevicePointer() obtains **physica**l memory address
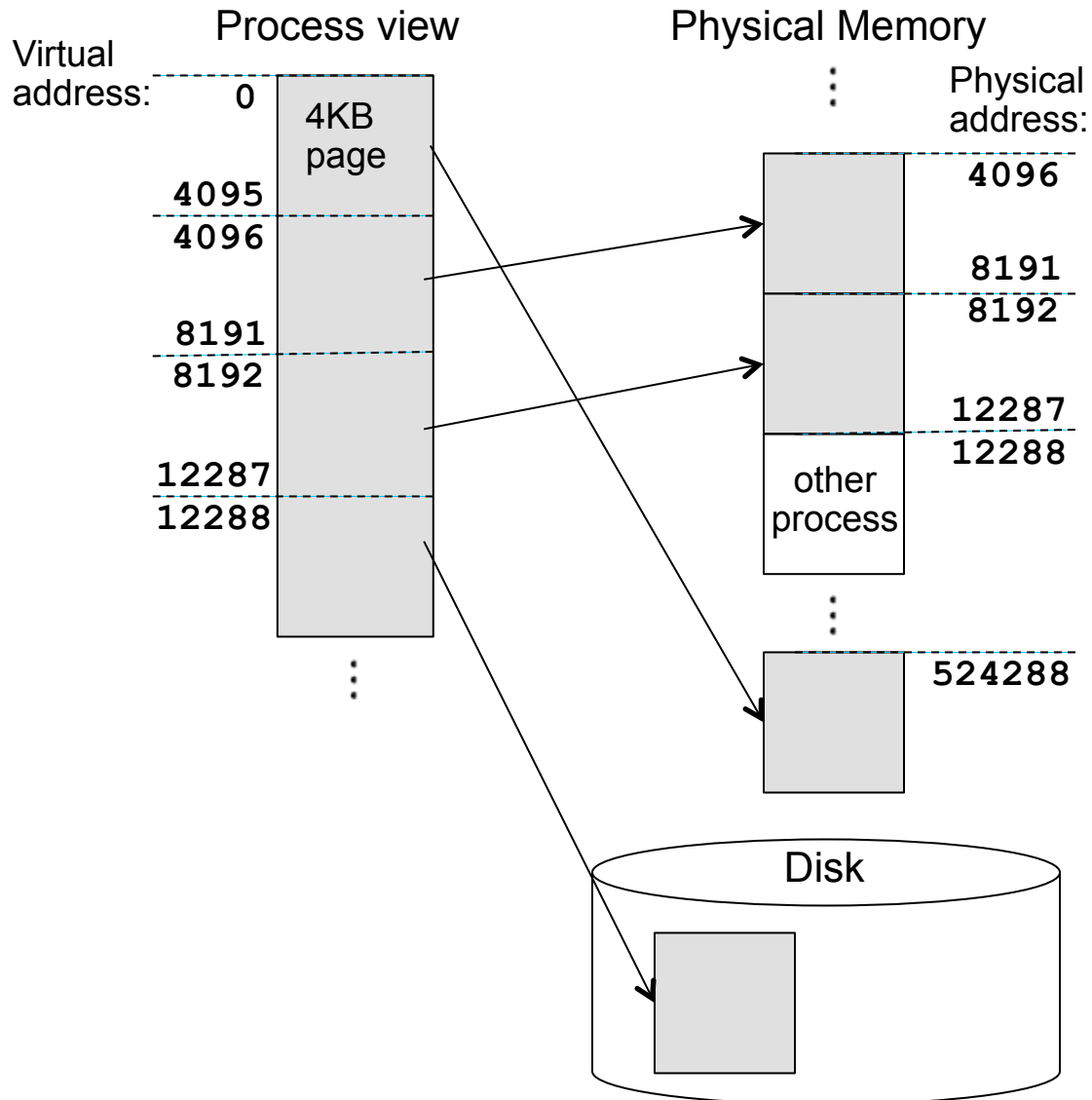
How to get data to the GPU (and back):

- Zero Copy Access (ZCA)
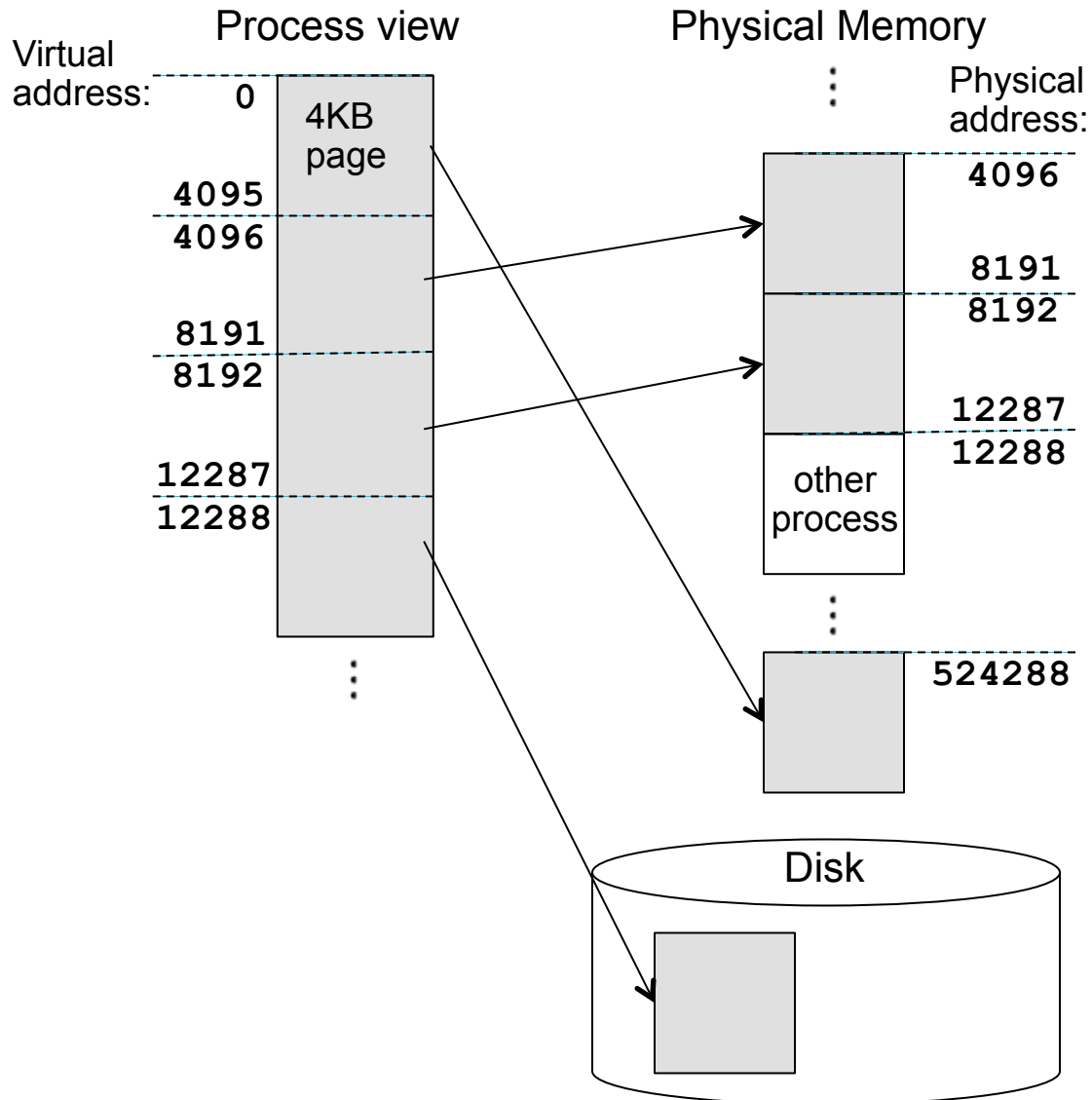 1. CPU thread pins memory page & creates pointer to phys. address

```
cudaHostRegister(h_ipt,ipt_size,cudaHostRegisterMapped);
cudaHostRegister(h_opt,opt_size,cudaHostRegisterMapped);
cudaHostGetDevicePointer(&ipt_ptr,h_ipt,0);
cudaHostGetDevicePointer(&opt_ptr,h_opt,0);
```

  2. Function call contains pointer(s) to host address(es)

```
computeKernel<<<dimGrid,dimBlock>>>
    (ipt_ptr, ipt_size, opt_ptr);
```

How to get data to the GPU (and back):

- Zero Copy Access (ZCA)

  1. CPU thread pins memory page & creates pointer to phys. address

```
cudaHostRegister(h_ipt,ipt_size,cudaHostRegisterMapped);
cudaHostRegister(h_opt,opt_size,cudaHostRegisterMapped);
cudaHostGetDevicePointer(&ipt_ptr,h_ipt,0);
cudaHostGetDevicePointer(&opt_ptr,h_opt,0);
```

  2. Function call contains pointer(s) to host address(es)

```
computeKernel<<<dimGrid,dimBlock>>>
    (ipt_ptr, ipt_size, opt_ptr);
```

  3. CUDA kernel/threads execute load/stores instructions as before

```
__global__ computeKerne(float* ipt_ptr, int ipt_size,
                        float* opt_ptr){
int entry=blockIdx.x*blockDim.x + threadIdx.x;
while(entry < ipt_size){
   opt_ptr[entry] = ipt_ptr[entry] + 42.0;
   entry += blockIdx.x*blockDim.x;
}
```
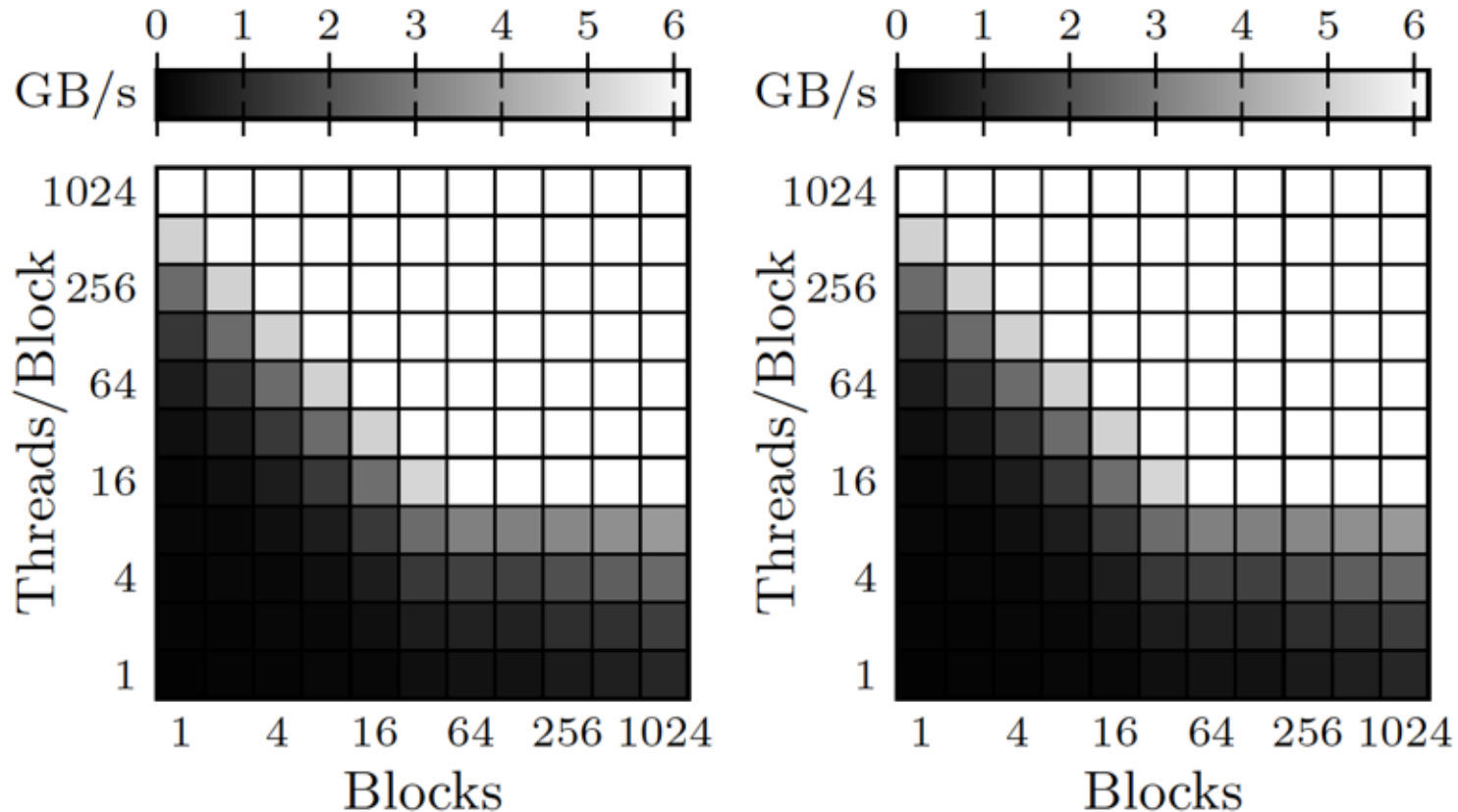
How to get data to the GPU (and back):

- Zero Copy Access (ZCA)

- What is the maximum (theoretical) performance ?
  - PCI-E 2.0 operates at 5 GigaTransfers(bit) / second per lane
  - Most GPUs use 16 lanes
  - 5 Gbit/s * 16 lanes = 80 Gbit /s
  - 8/10 bit ecoding reduces this to 64 Gbit/s payload or 8 GB/s
  - Packet protocol with 128B payload + 24B header reduces it to 6.27GB/s

  - PCI-E 3.0 operates at 8 GT/s
  - 16-lane GPU interface
  - 128/130 encoding
  - Bandwidth = ?

- Can this be achieved in practice (for PCI-E 2.0) ?

How to get data to the GPU (and back):

- Zero Copy Access (ZCA) – measured performance

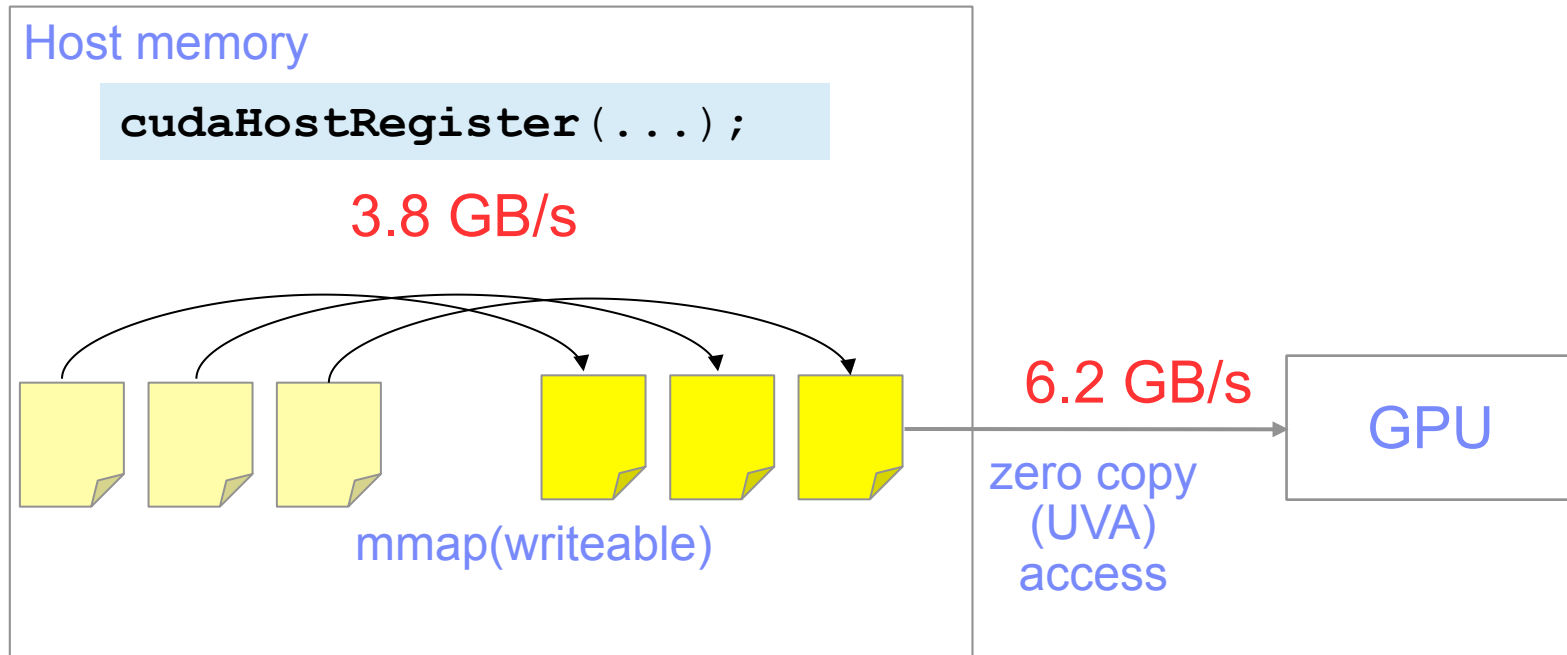64-bit coalesced read/write access to host memory using ZCA on a GTX580

- What about read & write at the same time ?

How to get data to the GPU (and back):

- Zero Copy Access (ZCA) – pinning

Register Pages
→ GPU accessible pages must be pinned (why?)
→ GPU accessible pages pages must be writable (why?)
　　→ copy page before pinning

Registering (and un-registering) is a bottleneck!
→ 3.8 GB/s < 6.2 GB/s on PCIe

How to get data to the GPU (and back):
Zero Copy Access (ZCA) –pinning "workaround"

Idea
- Use double buffering scheme
  - Pre-allocate pinned memory buffer
  - Copy data into one pre-allocated buffer
  - GPU kernel can operate on the second

→ Bottleneck is now is PCI-E again

How about PCI-E 3.0?

Large Scale Data Management on the GPU

- Why GPUs for information management workloads?
  – Search as an example

- Maximizing Device memory access performance
  – Coalescing
  – Thread configuration
  – Large(r) data sets

- GPU data transfers
  – Conventional
  – CUDA Streams
  – Zero Copy Access / Universal Virtual Addressing

- Search again
  – A naïve implementation
  – A novel parallel algorithm

A Simple implementation of (index) search

Keyword	Document ID
Adam	1,2,3
Bethlehem	4,5
Character	1,2,3,301,5790
Drachenflieger	301,317,5790
Eva	1,2
Flughafenbahnhof	5790
Grabdenkmal	2,5790
Haubentaucher	300,5790

A Simple implementation of (index) search

Keyword	Document ID
Adam	1,2,3
Bethlehem	4,5
Character	1,2,3,301,5790
Drachenflieger	301,317,5790
Eva	1,2
Flughafenbahnhof	5790
Grabdenkmal	2,5790
Haubentaucher	300,5790

sorted

16 characters max.

A Simple implementation of (index) search

Keyword

Adam
Bethlehem
Character
Drachenflieger
Eva
Flughafenbahnhof
Grabdenkmal
Haubentaucher

sorted

16 characters max.

```
char indexCPU[4711];
indexCPU[0]
indexCPU[16]
indexCPU[32]
...
```

A Simple implementation of (index) search

Keyword

Adam
Bethlehem
Character
Drachenflieger
Eva
Flughafenbahnhof
Grabdenkmal
Haubentaucher

sorted

16 characters max.

```
char indexCPU[4711];
indexCPU[0]
indexCPU[16]
indexCPU[32]
...
```

- On the CPU we use a few library calls and we are done

```
char searchkey[16]= "Flughafenbahnhof";
result = bsearch((void*)searchkey,indexCPU,
                 numentries,sizeof(char)*16,
                 (int(*)(const void*,const void*)) strcmp);
```

A Simple implementation of (index) search

Keyword

```
char indexCPU[4711];
indexCPU[0]
indexCPU[16]
indexCPU[32]
...
```

Adam
Bethlehem
Character
Drachenflieger
Eva
Flughafenbahnhof
Grabdenkmal
Haubentaucher

sorted

16 characters max.

- On the CPU we use a few library calls and we are done

```
char searchkey[16]= "Flughafenbahnhof";
result = bsearch((void*)searchkey,indexCPU,
                  numentries,sizeof(char)*16,
                  (int(*)(const void*,const void*)) strcmp);
```

- Can we just port a CPU implementation?

A Simple GPU implementation

- Get the data to the GPU

```
char* indexGPU;
char* searchkeysGPU;
char* resultsGPU;
// copy the data
cudaMalloc((void**)&indexGPU, sizeof(char)*wordlength*entries);
cudaMemcpy(indexGPU, indexCPU, sizeof(char)*wordlength*entries,
          CudaMemcpyHostToDevice);
// copy the searchkey(s)
cudaMalloc((void**)&searchkeysGPU, …
cudaMemcpy(searchkeysGPU, searchkeysCPU,
          sizeof(char)*wordlength*numsearches,
          CudaMemcpyHostToDevice);
// make room for the results
cudaMalloc((void**)&resultsGPU, …
```

A Simple GPU implementation

- Get the data to the GPU

```
char* indexGPU;
char* searchkeysGPU;
char* resultsGPU;
// copy the data
cudaMalloc((void**)&indexGPU, sizeof(char)*wordlength*entries);
cudaMemcpy(indexGPU, indexCPU, sizeof(char)*wordlength*entries,
          CudaMemcpyHostToDevice);
// copy the searchkey(s)
cudaMalloc((void**)&searchkeysGPU, …
cudaMemcpy(searchkeysGPU, searchkeysCPU,
          sizeof(char)*wordlength*numsearches,
          CudaMemcpyHostToDevice);
// make room for the results
cudaMalloc((void**)&resultsGPU, …
```

- Know your hardware (GTX 285, 30 SMs, 8 cores each, 240 cores)
  - Set up an execution configuration & call global function

```
dim3 Dg = dim3(30);
dim3 Db = dim3(8);
searchGPU< < < Dg,Db > > >(indexGPU, entries...
```

A Simple GPU implementation

- The GPU kernel

```
__global__ void searchGPU(char* index, int entries, int wordlength,
                          char* search_keys, int* results) {
  char* res;
  // use block and thread numbers for indexing
  res = bsearch(&search_keys[((blockIdx.x*BLOCK_SIZE)+threadIdx.x)
                             *wordlength],
               index,
               entries,
               wordlength);
  // use block and thread numbers for indexing
  results[(blockIdx.x*BLOCK_SIZE)+threadIdx.x] = (res-data)/
                                                 MAX_WORD_LENGTH;
}
```

A Simple GPU implementation

- The GPU kernel

```
__global__ void searchGPU(char* index, int entries, int wordlength,
                            char* search_keys, int* results) {
  char* res;
  // use block and thread numbers for indexing
  res = bsearch(&search_keys[((blockIdx.x*BLOCK_SIZE)+threadIdx.x)
                              *wordlength],
                index,
                entries,
                wordlength);
  // use block and thread numbers for indexing
  results[(blockIdx.x*BLOCK_SIZE)+threadIdx.x] = (res-data)/
                                                  MAX_WORD_LENGTH;
}
```

- There is no libc on the GPU =(
- Just stick __device__ in front of the libc code?
- "bsearch" is recursive, but there is no recursion on the GPU
➔ Write a iterative one ...

A Simple GPU binary search

```
__device__ char* bsearchGPU(char *key, char *base, int n, int size){
    char *mid_point;
    int  cmp;

    while (n > 0) {
        mid_point = (char *)base + size * (n >> 1);
        if ((cmp = strcmpGPU(key, mid_point)) == 0)
            return (char *)mid_point;
        if (cmp > 0) {
            base  = (char *)mid_point + size;
            n     = (n - 1) >> 1;
        } // cmp < 0
        else n >>= 1;
    }
    return (char *)NULL;
}
```

- Still need strcmp

A Simple GPU binary search

```
__device__ char* bsearchGPU(char *key, char *base, int n, int size){
    char *mid_point;
    int  cmp;

    while (n > 0) {
        mid_point = (char *)base + size * (n >> 1);
        if ((cmp = strcmpGPU(key, mid_point)) == 0)
            return (char *)mid_point;
        if (cmp > 0) {
            base  = (char *)mid_point + size;
            n     = (n - 1) >> 1;
        } // cmp < 0
        else n >>= 1;
    }
    return (char *)NULL;
}
```

- Still need strcmp
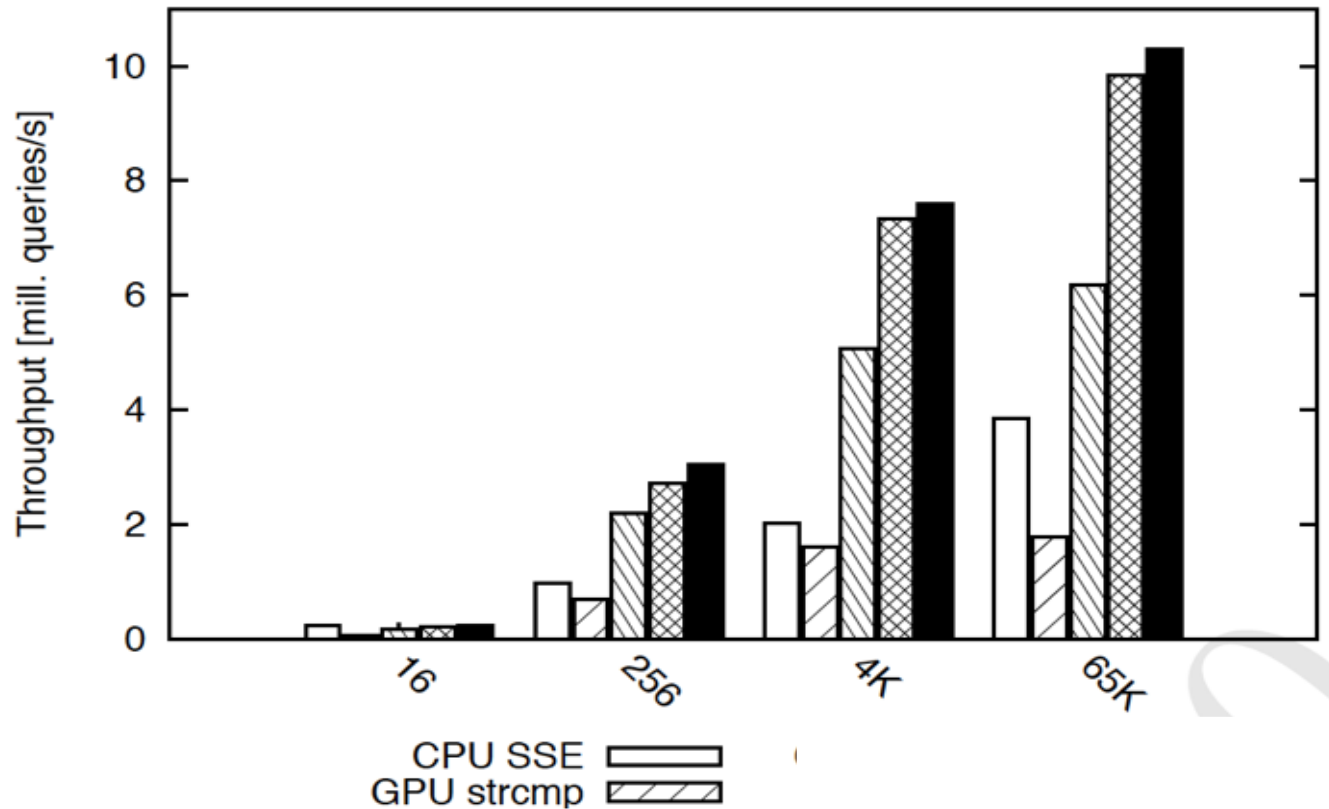
- Again, stick __device__ in front of the libc code

```
__device__ int strcmpGPU(char* s1, char* s2){
    while (*s1 == *s2++)
        if (*s1++ == 0) return 0;
    return (*s1 - *(s2 - 1));
}
```
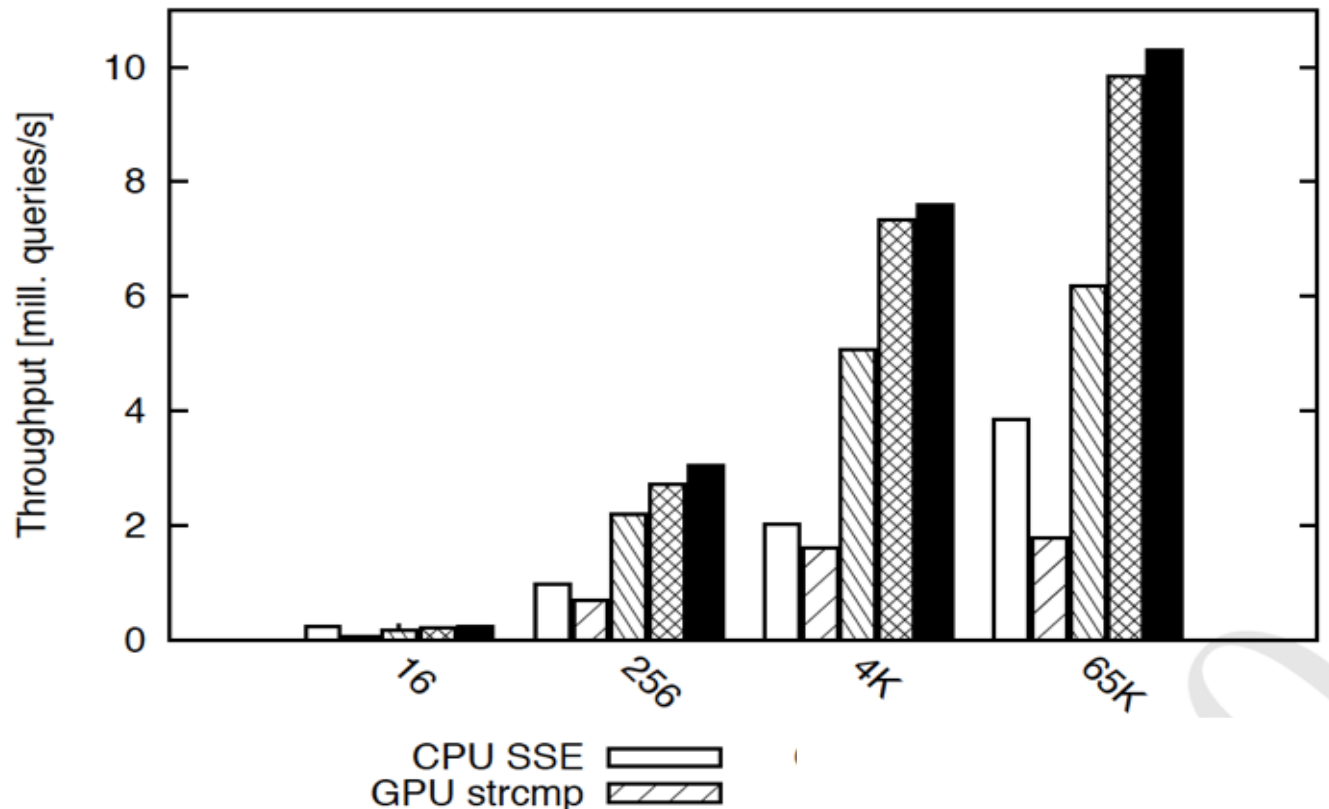
Binary Search on the GPU

- Searching a large data set (512MB) with 33 million (225) 16-character strings

Binary Search on the GPU – Why is it slow?

- Searching a large data set (512MB) with 33 million (2^{25}) 16-character strings
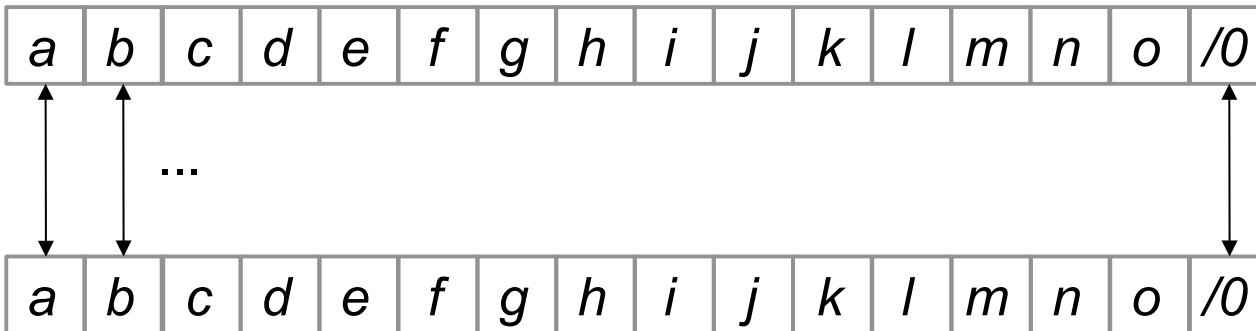

- It's slower than a CPU implementation for all data set sizes!
  - Let's try some optimizations ...

Search requires to compare

- Search naturally requires MANY comparisons
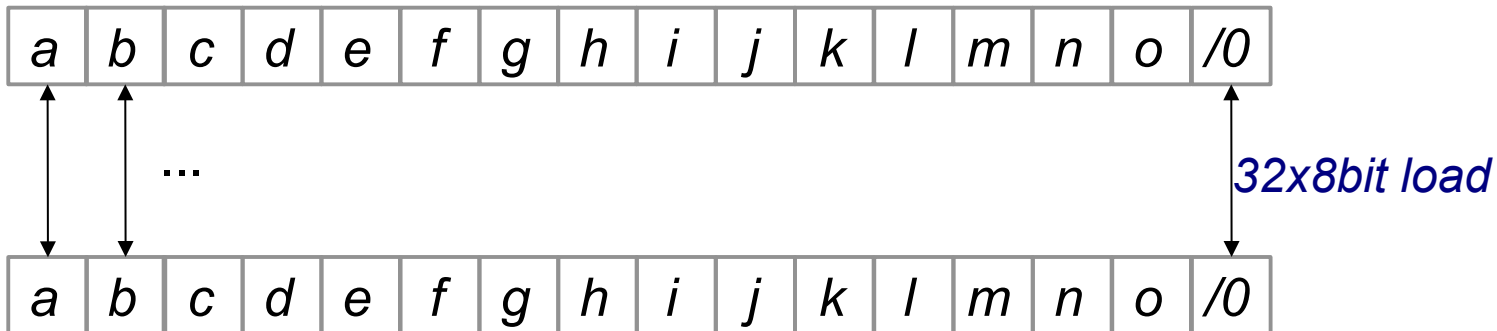
- The strcmp() library function:

```
int strcmp(const char* s1, const char* s2){
        while (*s1 == *s2++)
                if (*s1++ == 0)return 0;
        return (*s1 - *(s2 - 1));
}
```

| a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | /0 |

...

| a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | /0 |

Search requires to compare

- Search naturally requires MANY comparisons
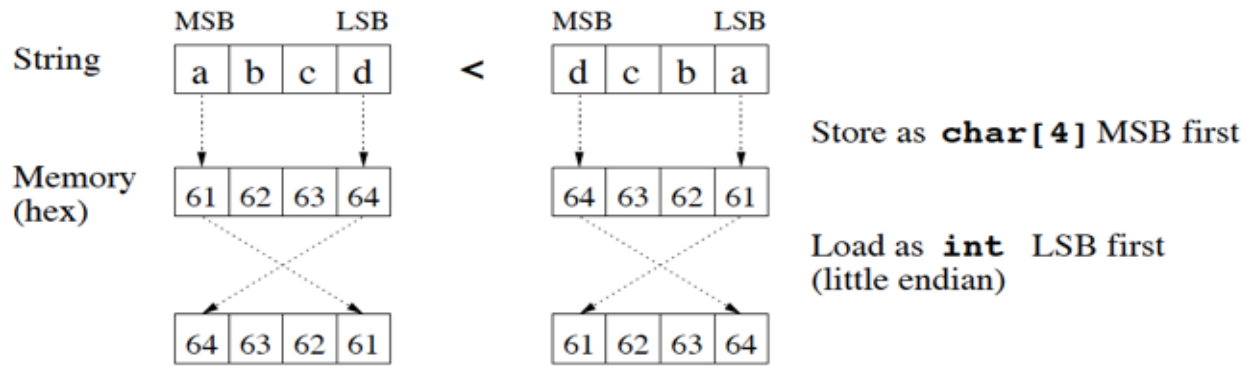
- The strcmp() library function:

```
int strcmp(const char* s1, const char* s2){
        while (*s1 == *s2++)
                if (*s1++ == 0)return 0;
        return (*s1 - *(s2 - 1));
}
```

| a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | /0 |

...   *32x8bit load*

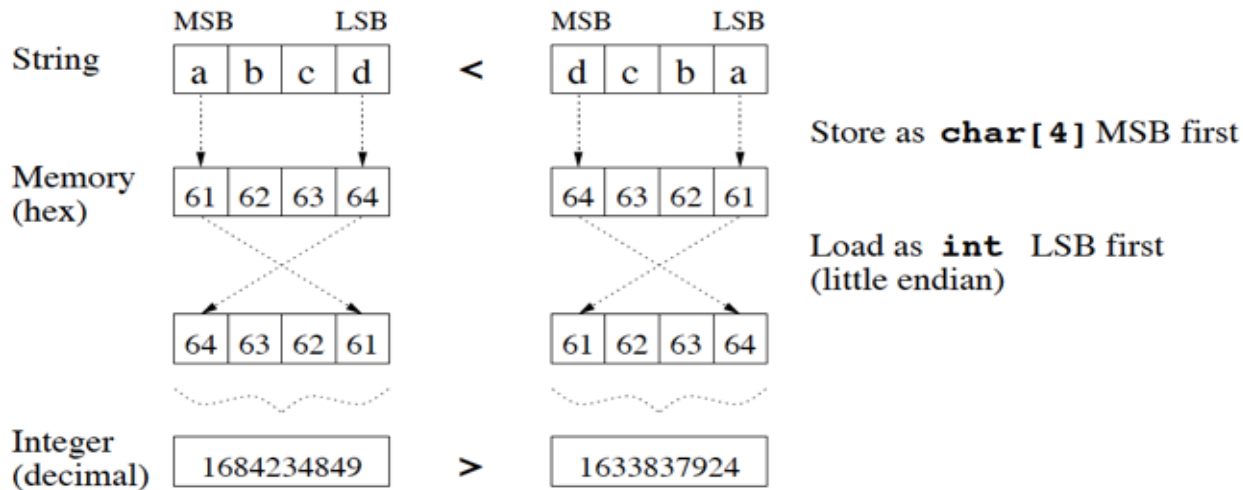| a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | /0 |

- Byte-wise memory access is known to be slow

Optimizing compare operations

- How about vector string comparison, a la SSE?

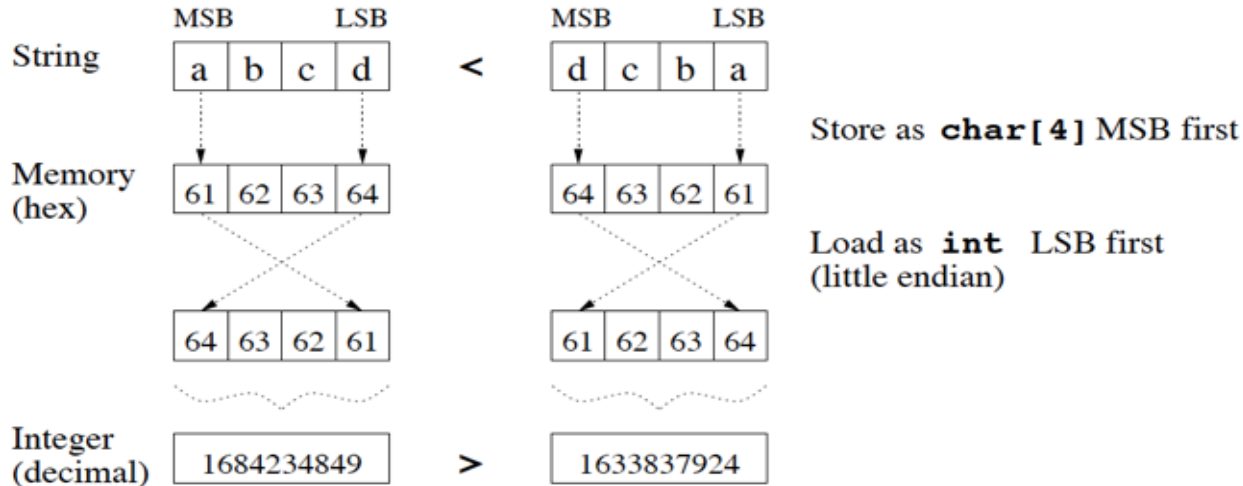- No Byte vectors on the GPU … but Integer vectors

String: a b c d < d c b a

MSB ... LSB

Store as char[4] MSB first

Memory (hex): 61 62 63 64 | 64 63 62 61

Load as int LSB first (little endian)

64 63 62 61 | 61 62 63 64

Optimizing compare operations

- How about vector string comparison, a la SSE?

- No Byte vectors on the GPU … but Integer vectors

	MSB			LSB			MSB			LSB	
String	a	b	c	d	<		d	c	b	a	

Store as **char[4]** MSB first

| Memory (hex) | 61 | 62 | 63 | 64 | | 64 | 63 | 62 | 61 |

Load as **int** LSB first (little endian)

| | 64 | 63 | 62 | 61 | | 61 | 62 | 63 | 64 |

| Integer (decimal) | 1684234849 | > | 1633837924 |

Optimizing compare operations

- How about vector string comparison, a la SSE?

- No Byte vectors on the GPU … but Integer vectors

- Loading character strings as int changes endianness
- CPU has bswap, on the GPU we have to write it:

```
#define BSWP( x ) ; \
temp = ( x ) << 24 ; \
temp = temp | ( ( ( x ) << 8) & 0x00FF0000 ) ; \
temp = temp | ( ( ( unsigned ) ( x ) >> 8) & 0x0000FF00 ) ; \
x = temp | ( ( unsigned ) ( x ) >> 24 ) ;
```
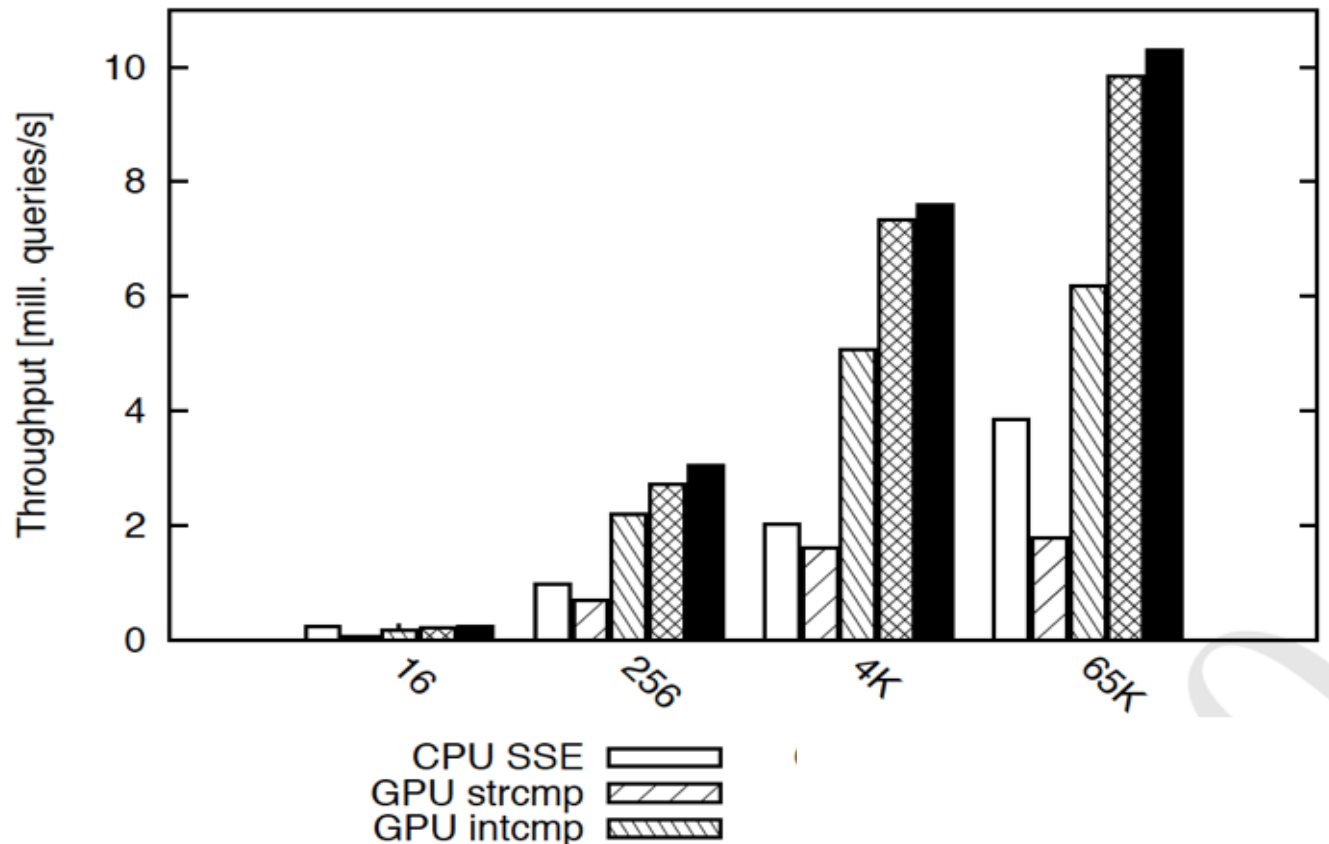
Optimizing compare operations

- Comparing integer vectors (bswap for <> skipped for clarity)

```
__device__ int intcmp(uint4* a, uint4* b){

    int r =1;
    if ((*a).x < (*b).x)
        r=-1;
    else if ((*a).x == (*b).x) {
        if ((*a).y  < (*b).y)
            r=-1;
        else if ((*a).y == (*b).y) {
            if ((*a).z  < (*b).z)
                r=-1;
            else if ((*a).z == (*b).z) {
                if ((*a).w < (*b).w)
                    r=-1;
                else if ((*a).w == (*b).w)
                    r=0;
            }
        }
    }
    return r;
}
```

- Still dereferencing 16 memory pointers ...

Binary Search on the GPU – Why is it slow?

- Searching a large data set (512MB) with 33 million (225) 16-character strings

- With intcmp it's only marginally faster than a CPU implementation
- We still do pointer chasing, i.e. roundtrips to memory ...

Reducing global memory access

- Intcmp is memory latency sensitive

Processor	L1 [cyc]	L2 [cyc]	L3 [cyc]	mem [cyc]
Intel Core i7 2.6GHz	4	10	40	350
nVidia GT200b 1.5 GHz	4	n/a	n/a	500

x 16 for each comparison !!!

- We can use shared memory like L1

Reducing global memory access

- Intcmp is memory latency sensitive

Processor	L1 [cyc]	L2 [cyc]	L3 [cyc]	mem [cyc]
Intel Core i7 2.6GHz	4	10	40	350
nVidia GT200b 1.5 GHz	4	n/a	n/a	500

x 16 for each comparison !!!

- We can use shared memory like L1

```
__shared__ uint4 cache[NUM_THREADS*2];

__device__ uint4* bsearchGPU( uint4 *key,  uint4 *base,
        size_t nmemb,  size_t size)
{
  uint4 *mid_point;
  int  cmp;
  cache[threadIdx.x*2]= *key;

  while (nmemb > 0) {
     mid_point = (uint4 *)base + size * (nmemb >> 1);
     cache[threadIdx.x*2+1]= *mid_point;
     if ((cmp = intcmp(&cache[threadIdx.x*2],
                 &cache[threadIdx.x*2+1]))== 0)
         return (uint4 *)mid_point;
```

Binary Search on the GPU – optimized

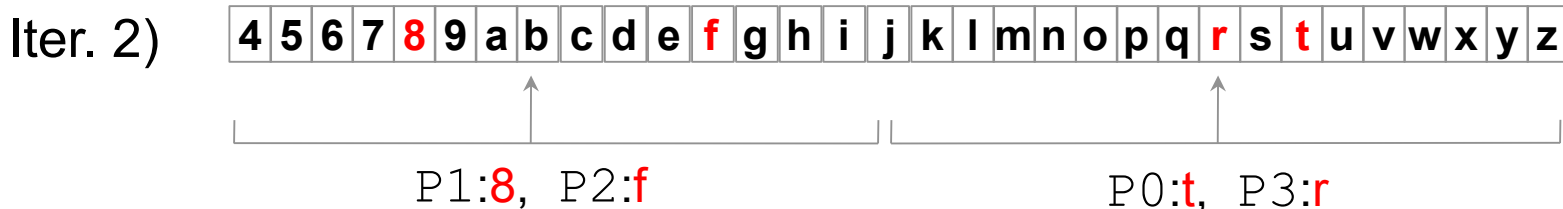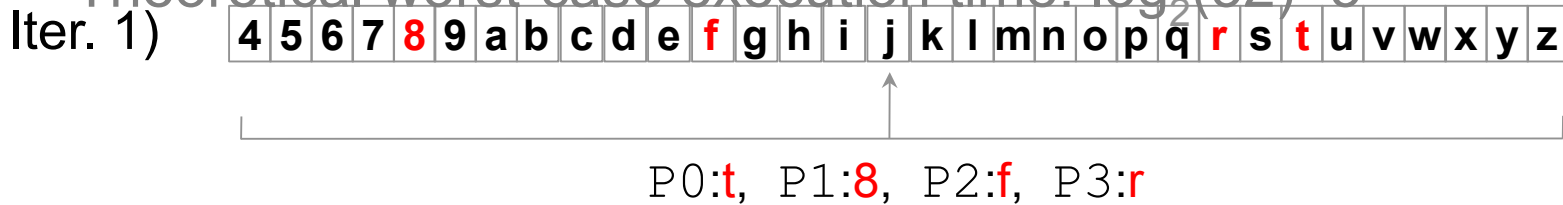- Searching a large data set (512MB) with 33 million (225)
  16-character strings

Multi-threaded Binary Search – Example

- Index: a sorted char array 32 entries

- 4 queries: t , 8 , f , r

- 4 processor cores: `P1-P4`

- 1 processor core – 1 search: `P0:`t , `P1:`8 , `P2:`f , `P3:`r
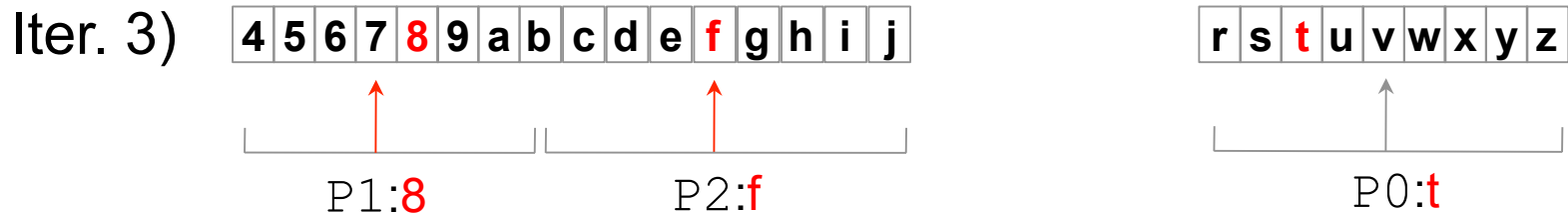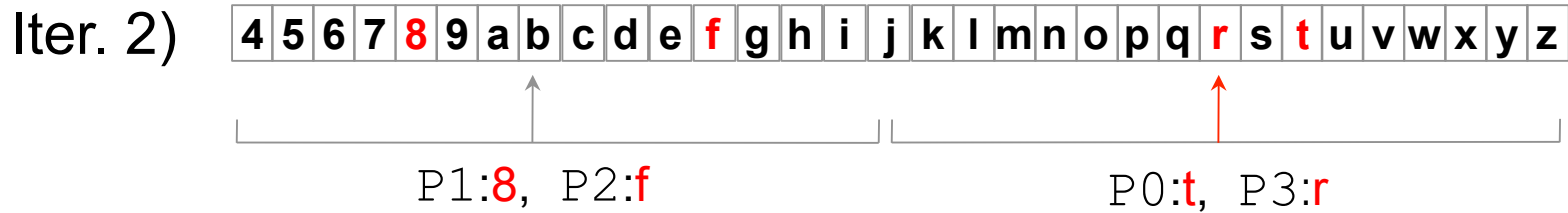
- Theoretical worst-case execution time: $\log_2(32)=5$

4	5	6	7	8	9	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z

Multi-threaded Binary Search – Example

- Index: a sorted char array 32 entries

- 4 queries: t , 8 , f , r

- 4 processor cores: `P1-P4`

- 1 processor core – 1 search: `P0:`t , `P1:`8 , `P2:`f , `P3:`r

- Theoretical worst-case execution time: $\log_2(32)=5$

Iter. 1)

4	5	6	7	8	9	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z

`P0:`t, `P1:`8, `P2:`f, `P3:`r

Iter. 2)

4	5	6	7	8	9	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z

`P1:`8, `P2:`f

`P0:`t, `P3:`r

© 2013 IBM Corporation

Multi-threaded Binary Search – Example

Iter. 2) 4 5 6 7 **8** 9 a b c d e **f** g h i j k l m n o p q **r** s **t** u v w x y z

P1:8, P2:f   P0:t, P3:r

Iter. 3) 4 5 6 7 **8** 9 a b c d e **f** g h i j   **r** s **t** u v w x y z

P1:8   P2:f   P0:t

Iter. 4) 7 **8** 9 a b   **r** s **t** u v

P1:8   P0:t

Iter. 5) 7 **8** 9

P1:8

Conventional multi-threading – Analysis

- 100% utilization requires #cores concurrent queries

- Queries finishing early

  ➔ utilization < 100%

- Memory access collisions

  ➔ serialized memory access

- #memory accesses $\log_2(n)$

- More threads
  ➔ more results

  ➔ response time likely to be worst case: $\log_2(n)$

Can we improve the worst case?

Large Scale Data Management on the GPU

- Why GPUs for information management workloads?
  – Search as an example

- Maximizing Device memory access performance
  – Coalescing
  – Thread configuration
  – Large(r) data sets

- GPU data transfers
  – Conventional
  – CUDA Streams
  – Zero Copy Access / Universal Virtual Addressing

- Search again
  – A naïve implementation
  – A novel parallel algorithm

Binary Search

- How Do you (efficiently) search an index?

- Open phone book ~middle

- 1st name = whom you are looking for?

- $<$, $>$?

- Iterate

  – Each iteration: #entries/2 (n/2)
  – Total time:
    ➔ $\log_2(n)$

Parallel (Binary) Search

- What if you have some friends (3) to help you ?

- Divide et impera !

- Give each of them ¼ *

  – Each is using binary search takes $\log_2(n/4)$
- All can work in parallel ➔ faster: $\log_2(n/4) < \log_2(n)$

Parallel (Binary) Search

- What if you have some friends (3) to help you ?

- Divide et impera !

- Give each of them ¼ *

  - Each is using binary search takes $\log_2(n/4)$
  - All can work in parallel ➔ faster: $\log_2(n/4) < \log_2(n)$
  - 3 of you are wasting time !

* You probably want to tear it a little more intelligent than that, e.g. at the binding ;-)
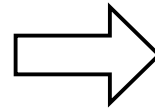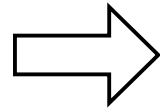
P-ary Search

- Divide et impera !!

- How do we know who has the right piece ?

P-ary Search

- Divide et impera !!

- How do we know who has the right piece ?

- It's a sorted list:
  - Look at first and last entry of a subset
  - If first entry < searched name < last entry
    - Redistribute
    - Otherwise … throw it away
  - Iterate

P-ary Search

- What do we get?

+

- Each iteration: n/4
  ➔ $\log_4(n)$
- Assuming redistribution time is negligible:
  $\log_4(n) < \log_2(n/4) < \log_2(n)$
- But each does 2 lookups !
- How time consuming are lookup and redistribution ?

P-ary Search

- What do we get?

+

- Each iteration: n/4
  ➔ $\log_4(n)$
- Assuming redistribution time is negligible:
  $\log_4(n) < \log_2(n/4) < \log_2(n)$
- But each does 2 lookups !
- How time consuming are lookup and redistribution ?

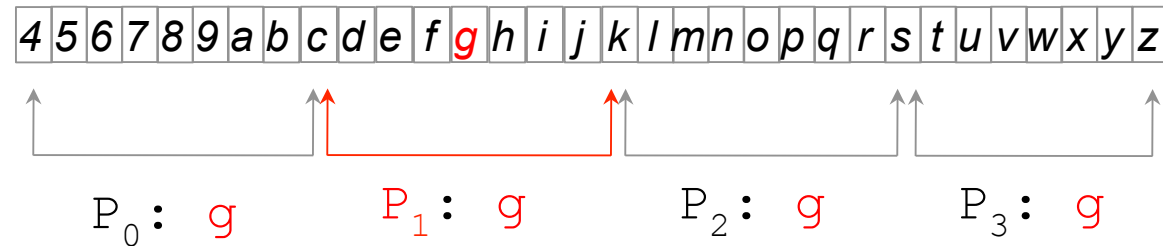 $=$ $=$

memory   synchronization
access

P-ary Search

- What do we get?

+

- Each iteration: n/4
  ➔ $\log_4(n)$
- Assuming redistribution time is negligible:
  $\log_4(n) < \log_2(n/4) < \log_2(n)$
- But each does 2 lookups !
- How time consuming are lookup and redistribution ?

$\parallel$ $\parallel$

memory   synchronization
access

- Searching a database index can be implemented the same way
  – Friends = Processor cores (threads)
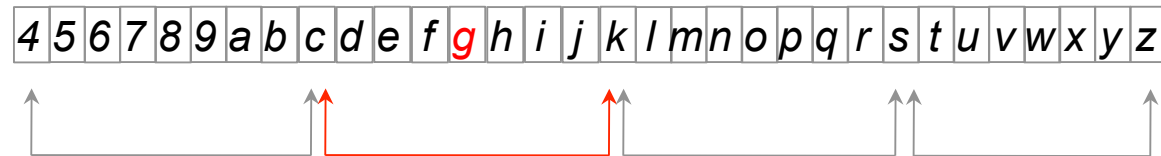 – Without destroying anything ;-)
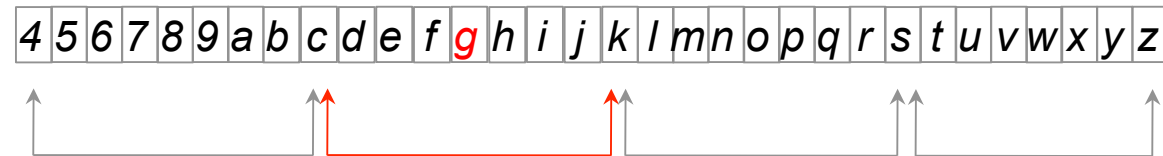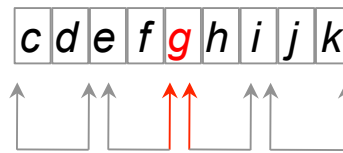
P-ary Search - Implementation

- Strongly relies on fast synchronization
  - friends = threads / vector elements

| 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z |

Iteration 1) P_0: g P_1: g P_2: g P_3: g

P-ary Search - Implementation

- Strongly relies on fast synchronization
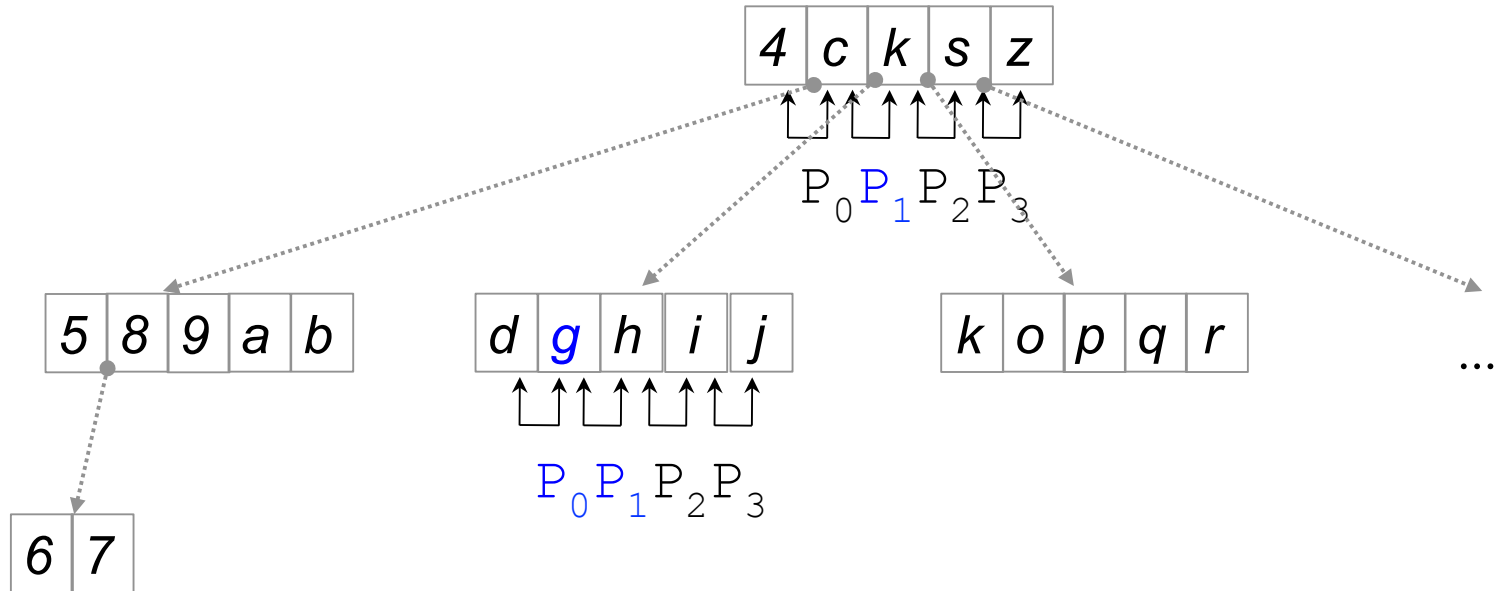 - friends = threads / vector elements

| 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z |

Iteration 1) P_0: g P_1: g P_2: g P_3: g

| c | d | e | f | g | h | i | j | k |

Iteration 2) P_0 P_1 P_2 P_3: g

P-ary Search - Implementation

- Strongly relies on fast synchronization
  - friends = threads / vector elements

| 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z |

Iteration 1) P_0: g P_1: g P_2: g P_3: g

| c | d | e | f | g | h | i | j | k |

Iteration 2) P_0 P_1 P_2 P_3: g

- Synchronization ~ repartition cost
- pthreads ($$), `cmpxchng`($)
- SIMD SSE-vector, GPU threads via shared memory (~0)

- Implementation using a B-tree is similar and (obviously) faster

P-ary Search - Implementation

- B-trees group pivot elements into nodes

| 4 | c | k | s | z |

$$P_0 \; P_1 \; P_2 \; P_3$$

| 5 | 8 | 9 | a | b | | d | g | h | i | j | | k | o | p | q | r | ...

$$P_0 \; P_1 \; P_2 \; P_3$$

| 6 | 7 |

- Access to pivot elements is coalesced instead of a gather
- Nodes can also be mapped to
  - Cache Lines (CSB+ trees)
  - Vectors (SSE)
  - #Threads per block

```
__shared__ int offset;
__shared__ int cache[BLOCKSIZE+2]


__global__ void parySearchGPU(int* data, int length,
                                int* list_of_search_keys, int* results)

    int start, sk;
    int old_length = length;
// initialize search range starting with the whole data set
    if (threadIdx.x ==0 ) {
       offset = 0;
       // cache search key and upper bound in shared memory
       cache[BLOCKSIZE] = 0x7FFFFFFF;
       cache[BLOCKSIZE+1] = list_of_search_keys[blockIdx.x];
       results[blockIdx.x] = -1;
    }
    __syncthreads();
    //
    sk = cache[BLOCKSIZE+1];
```

```
__shared__ int offset;
__shared__ int cache[BLOCKSIZE+2]


__global__ void parySearchGPU(int* data, int length,
                              int* list_of_search_keys, int* results)

   int start, sk;
   int old_length = length;
// initialize search range starting with the whole data set
   if (threadIdx.x ==0 ) {
      offset = 0;
      // cache search key and upper bound in shared memory
      cache[BLOCKSIZE] = 0x7FFFFFFF;
      cache[BLOCKSIZE+1] = list_of_search_keys[blockIdx.x];
      results[blockIdx.x] = -1;
   }
   _syncthreads();
   //
   sk = cache[BLOCKSIZE+1];
```

Why?

P-ary Search on a sorted list – Implementation (2)

```
// repeat until the #keys in the search range < #threads
while (length > BLOCKSIZE){
    // calculate search range for this thread
    length = length/BLOCKSIZE;
    if (length * BLOCKSIZE < old_length) length += 1;
    old_length = length;
    // why don't we just use floating point?
    start = offset + threadIdx.x * length;
    // cache the boundary keys
    cache[threadIdx.x] = data[start];
    __syncthreads();
    // if the searched key is within this thread's subset,
    // make it the one for the next iteration
    if (sk >= cache[threadIdx.x] && sk < cache[threadIdx.x+1]){
         offset = start;
    }
    __syncthreads();
    // all threads start next iteration with the new subset
}
```
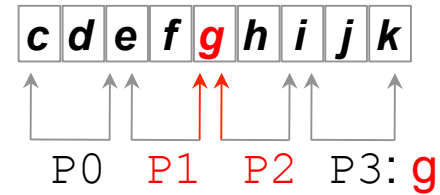
P-ary Search on a sorted list – Implementation (2)

```
// repeat until the #keys in the search range < #threads
while (length > BLOCKSIZE){
    // calculate search range for this thread
    length = length/BLOCKSIZE;
    if (length * BLOCKSIZE < old_length) length += 1;
    old_length = length;
    // why don't we just use floating point?
    start = offset + threadIdx.x * length;
    // cache the boundary keys
    cache[threadIdx.x] = data[start];
    _syncthreads();
    // if the searched key is within this thread's subset,
    // make it the one for the next iteration
    if (sk >= cache[threadIdx.x] && sk < cache[threadIdx.x+1]){
        offset = start;
    }
    syncthreads();
    // all threads start next iteration with the new subset
}
```

Why?

```
    // last iteration
    start = offset + threadIdx.x;
    if (sk == data[start])
        results[blockIdx.x] = start;
}
```
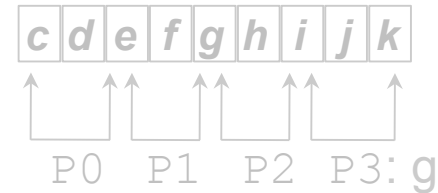
P-ary Search – Analysis

- 100% processor utilization for each query

- Multiple threads can find a result
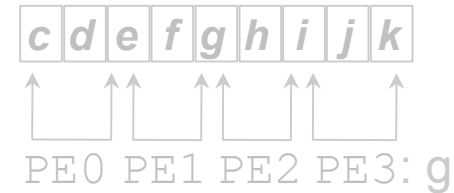 - How does this impact correctness?

| c | d | e | f | g | h | i | j | k |

P0   P1   P2   P3: g

P-ary Search – Analysis

- 100% processor utilization for each query

- Multiple threads can find a result
  - How does this impact correctness?

- Convergence depends on #threads

- GTX285: 1 SM, 8 cores(threads) → p=8
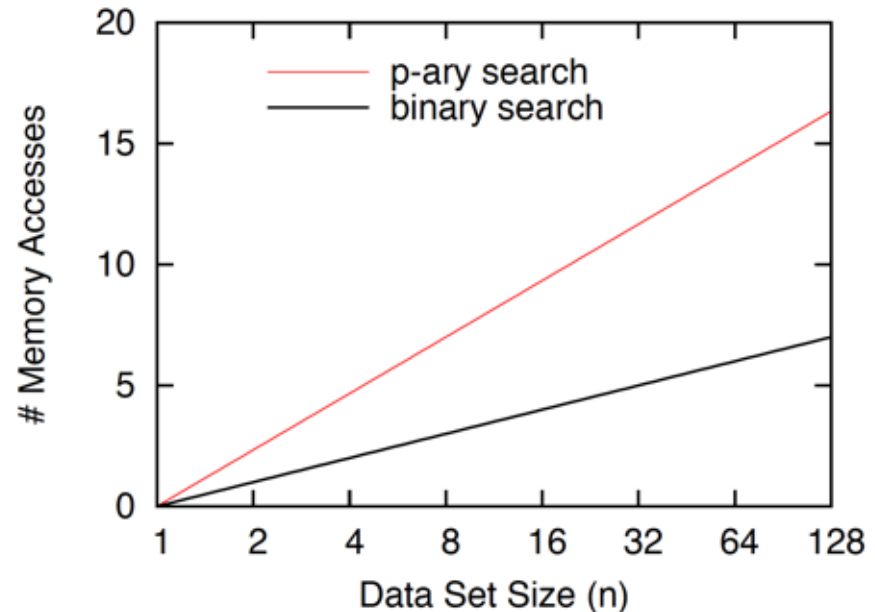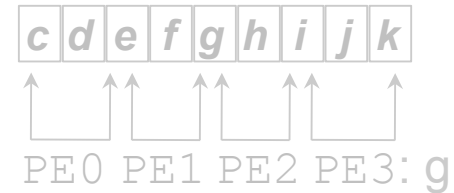
- Better Response time
  - $\log_p(n)$ vs $\log_2(n)$

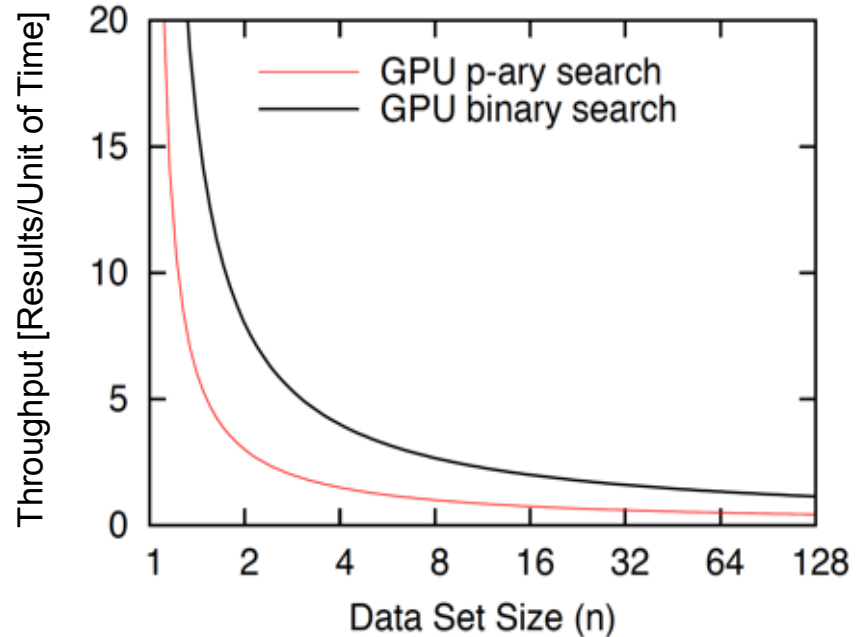| c | d | e | f | g | h | i | j | k |

P0   P1   P2   P3: g

P-ary Search – Analysis

- 100% processor utilization for each query

- Multiple threads can find a result
  - Does not change correctness

- Convergence depends on #threads

  GTX285: 1 SM, 8 cores(threads) $\rightarrow$ p=8

- Better Response time
  - $\log_p(n)$ vs $\log_2(n)$

- More memory access
  - (p*2 per iteration) * $\log_p(n)$
  - Caching
  (p-1) * $\log_p(n)$ vs. $\log_2(n)$

c	d	e	f	g	h	i	j	k

PE0 PE1 PE2 PE3: g

P-ary Search – Analysis

- 100% processor utilization for each query

- Multiple threads can find a result
  - Does not change correctness
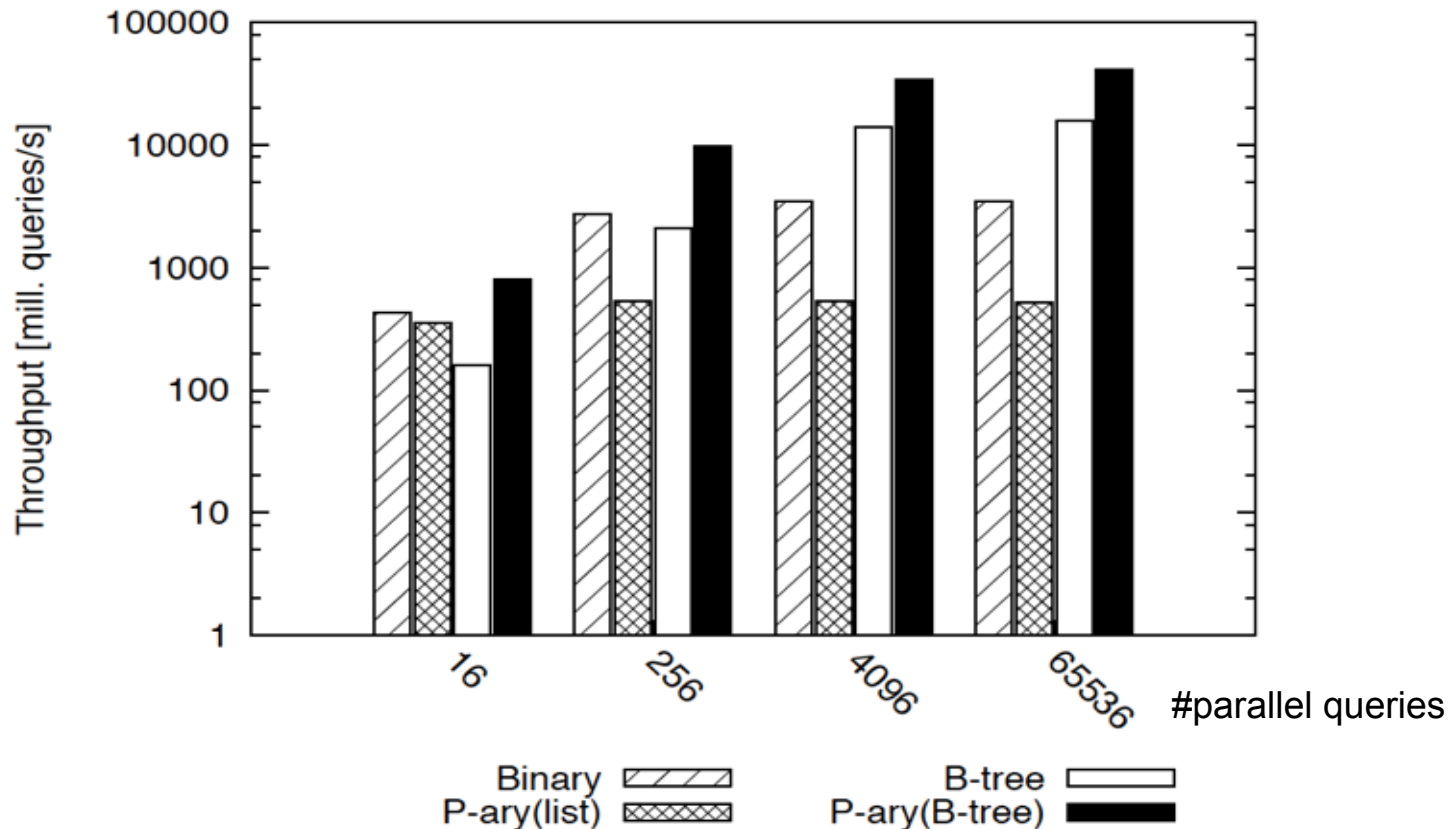
- Convergence depends on #threads

  GTX285: 1 SM, 8 cores(threads) $\rightarrow$ p=8

- Better Response time
  - $\log_p(n)$ vs $\log_2(n)$

- More memory access
  - p*2 per iteration * $\log_p(n)$
  - Caching

  (p-1) * $\log_p(n)$ vs. $\log_2(n)$

- **Lower Throughput**
  - **$1/\log_p(n)$ vs $p/\log_2(n)$**

c	d	e	f	g	h	i	j	k

PE0 PE1 PE2 PE3: g

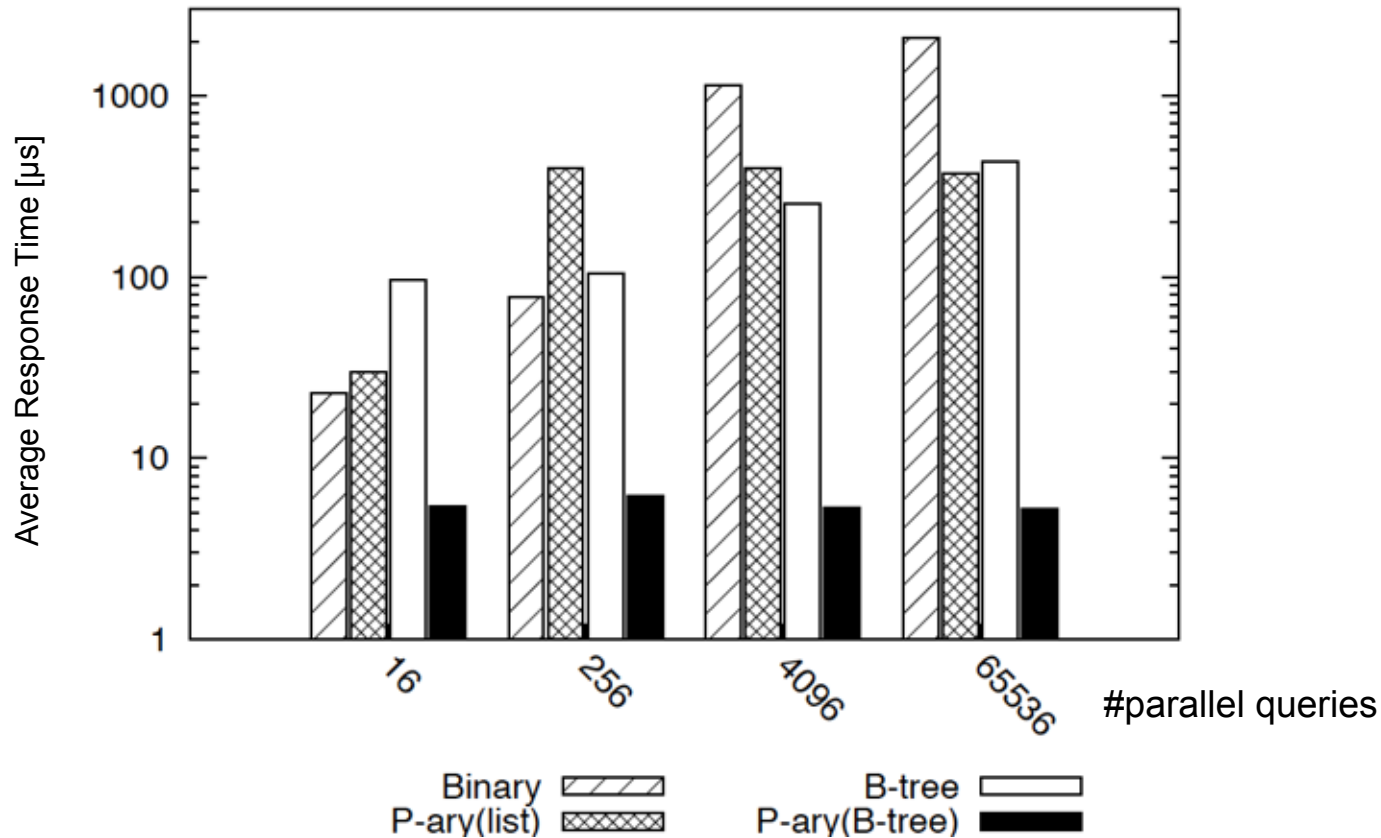# P-ary Search (GPU) – Throughput

- Superior throughput compared to conventional algorithms

Searching a 512MB data set with 134mill. 4-byte integer entries,
Results for a nVidia GT200b, 1.5GHz, GDDR3 1.2GHz.
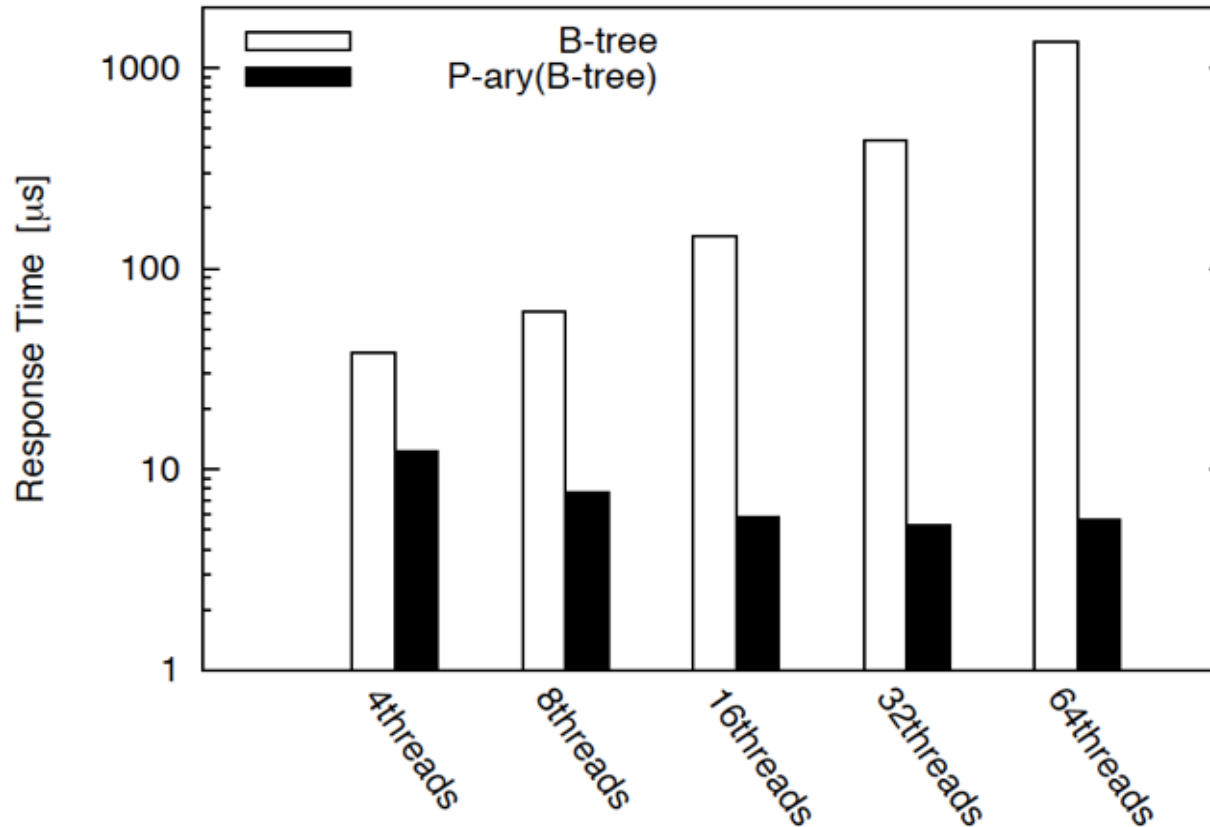
P-ary Search (GPU) – Response Time

- Response time is workload independent for B-tree implementation

Searching a 512MB data set with 134mill. 4-byte integer entries,
Results for a nVidia GT200b, 1.5GHz, GDDR3 1.2GHz.

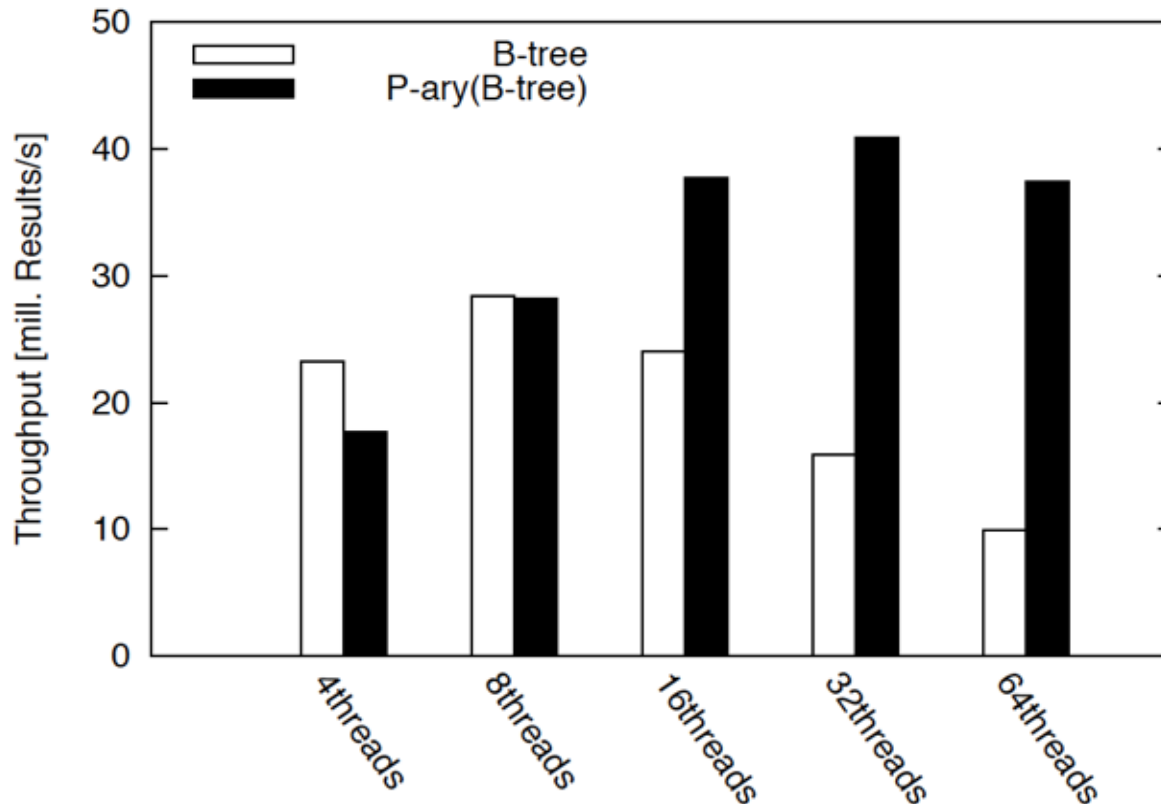# P-ary Search (GPU) – Scalability

- GPU Implementation using SIMT (SIMD threads)
- Scalability with increasing #threads (P)

64K search queries against a 512MB data set with 134mill. 4-byte integer entries,
Results for a nVidia GT200b, 1.5GHz, GDDR3 1.2GHz.
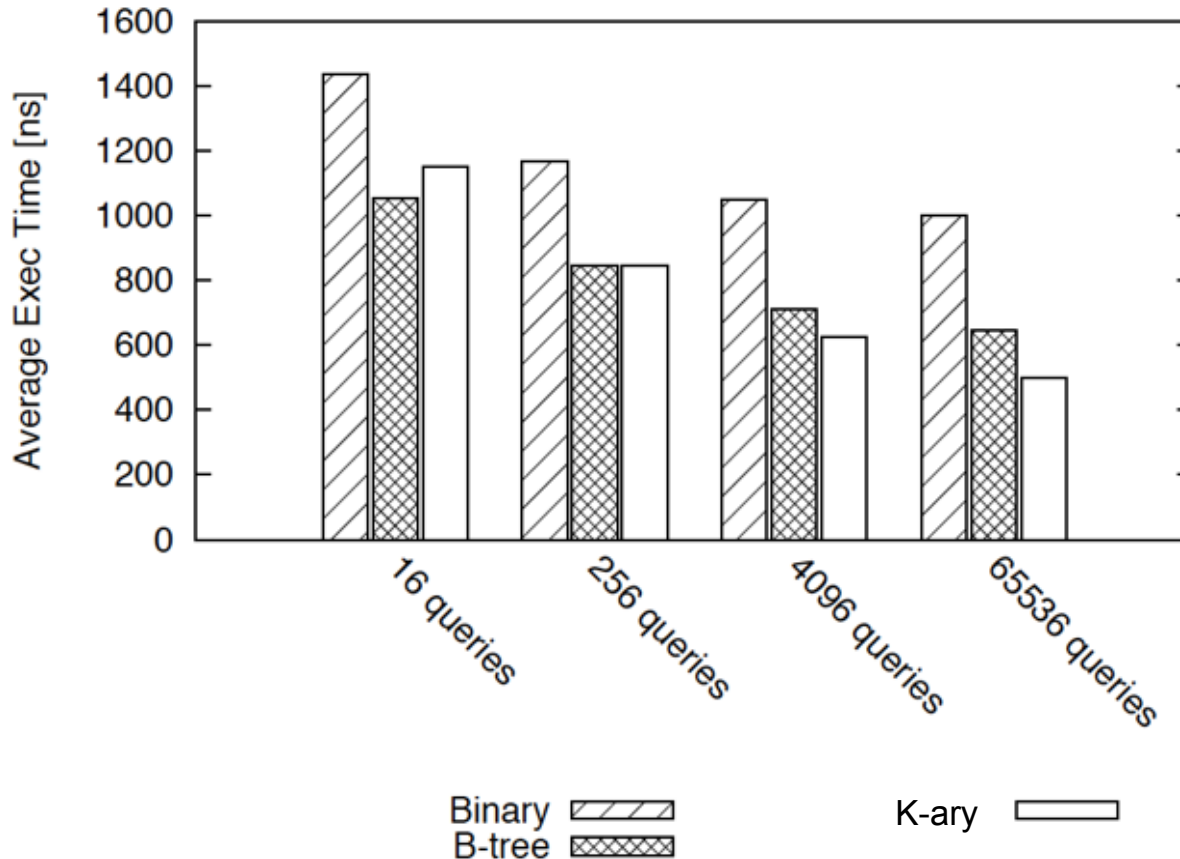
P-ary Search (GPU) – Scalability

- GPU Implementation using SIMT (SIMD threads)
- Scalability with increasing #threads (P)

64K search queries against a 512MB data set with 134mill. 4-byte integer entries, Results for a nVidia GT200b, 1.5GHz, GDDR3 1.2GHz.

P-ary Search(CPU) = K-ary Search[1]

- K-ary search is the same algorithm ported to the CPU using SSE vectors (int4) $\rightarrow$ convergence rate $\log_4(n)$
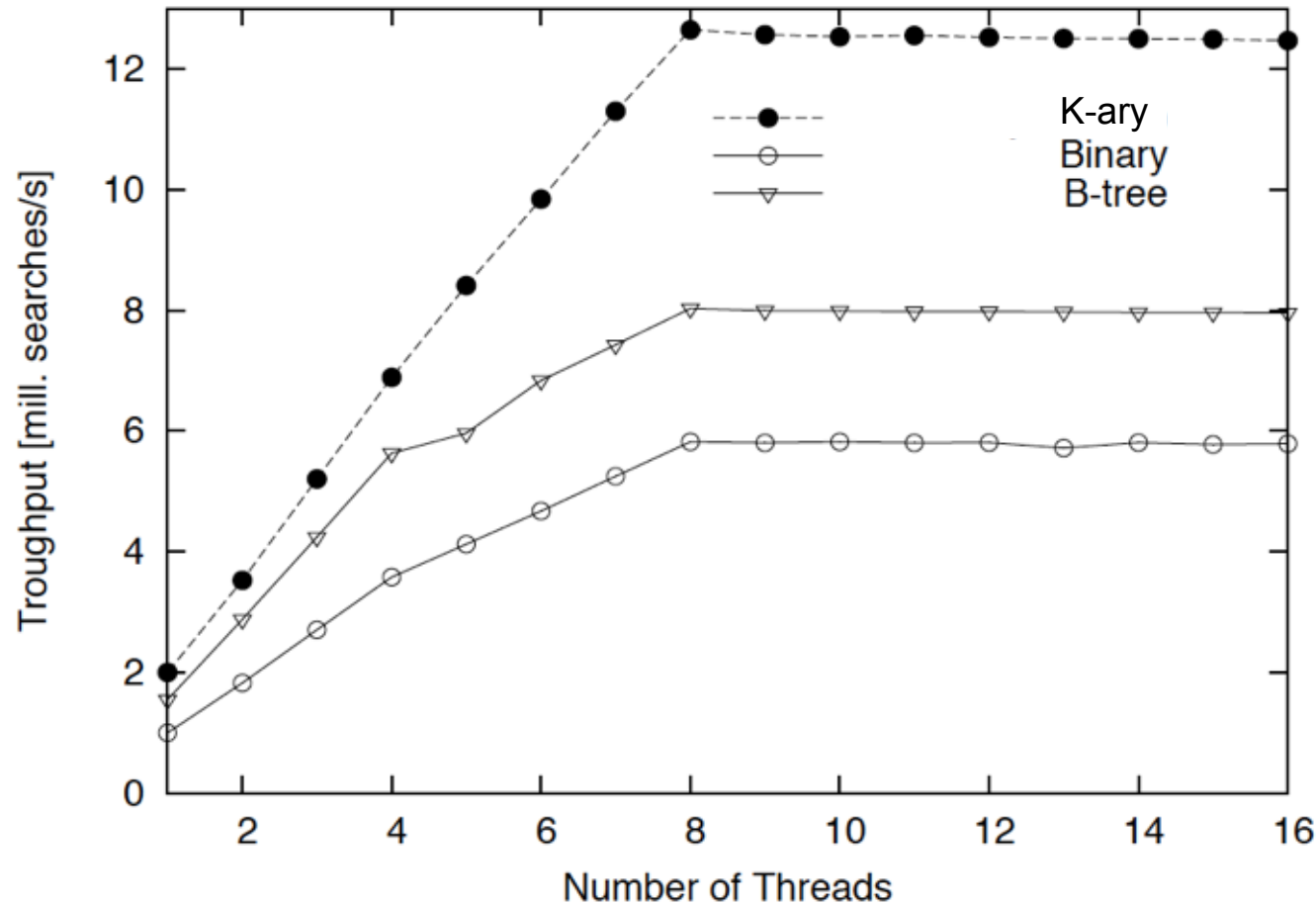

Searching a 512MB data set with 134mill. 4-byte integer entries, Core i7 2.66GHz, DDR3 1666.

[1] B. Schlegel, R. Gemulla, W. Lehner, k-Ary Search on Modern Processors, DaMoN 2000

P-ary Search(CPU) = K-ary Search[1]

- Throughput scales proportional to #threads

64K search queries against a 512MB data set with 134mill. 4-byte integer entries,
Core i7 2.66GHz, DDR3 1666.

[1] B. Schlegel, R. Gemulla, W. Lehner, k-Ary Search on Modern Processors, DaMoN 2000

Questions?