# Programming GPUs for database applications
## - outsourcing index search operations




Tim Kaldewey

Research Staff Member – Database Technologies
IBM Almaden Research Center
*tkaldew@us.ibm.com*

# Quo Vadis ?

+ ORACLE special projects

# Why Search ?

Honestly, how many times a day do you visit

?

# Quo Vadis ?

\+ ORACLE special projects

\+ 

# Quo Vadis ?

+ ORACLE special projects

+

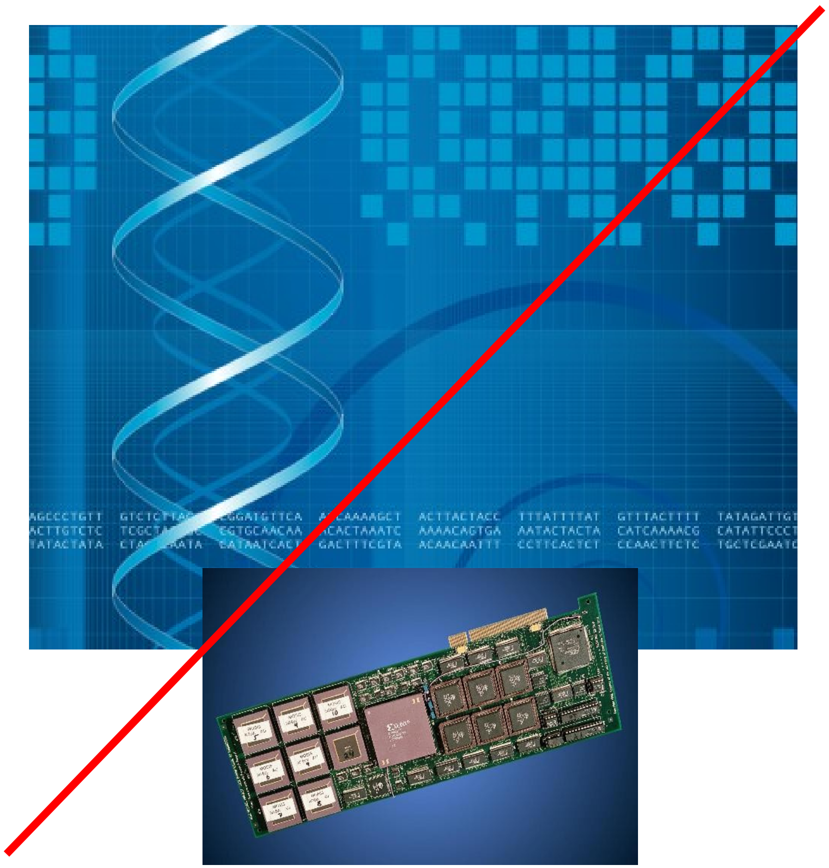
=

# Agenda

- Introduction
  - GPU & DB (search) ?
- GPU search
  - A first implementation – binary search
  - Conventional search algorithms & GPUs – a mismatch
  - Back to the drawing board:
    - P-ary search
    - Experimental evaluation
    - Why it works
- Conclusions

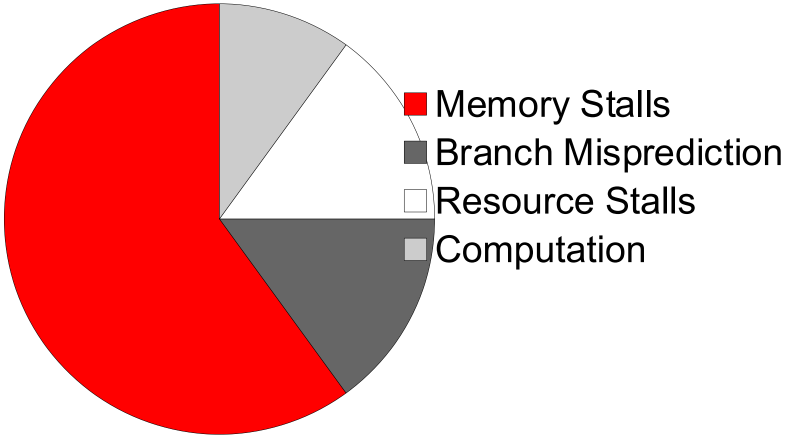
# Database Workloads

- Data-intensive
- Processor performance is not a problem
- Sifting through large quantities of data fast enough is



# DB Performance – Where does Time Go

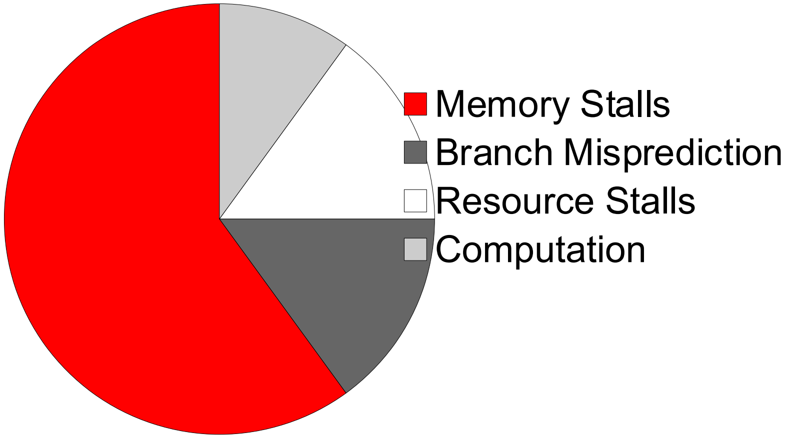
- CPU? I/O? Memory ? [1]
  - 10% indexed range selection



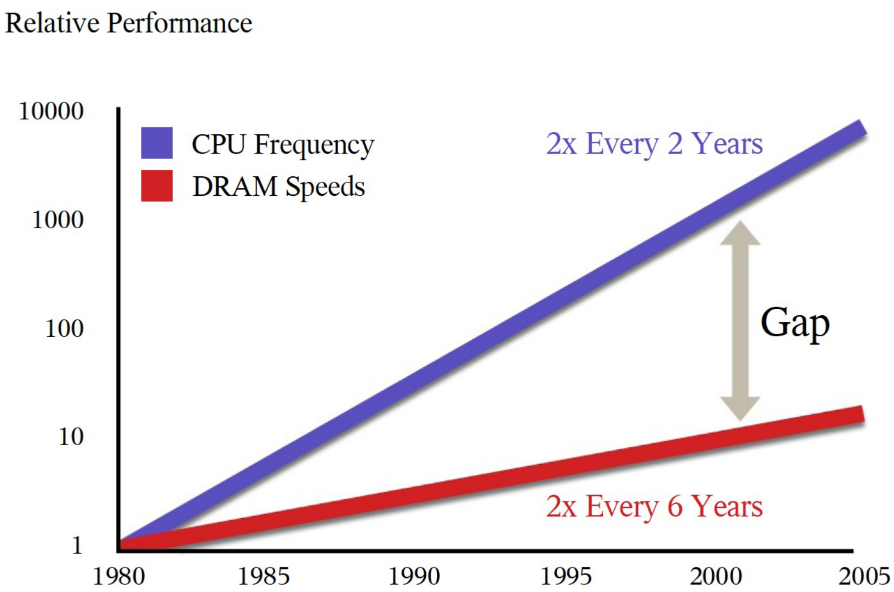

[1] A. Ailamaki, et al. DBMSs on a modern processor: Where does time go? VLDB’99

# DB Performance – Where does Time Go

- CPU? I/O? Memory ? [1]
  - 10% indexed range selection

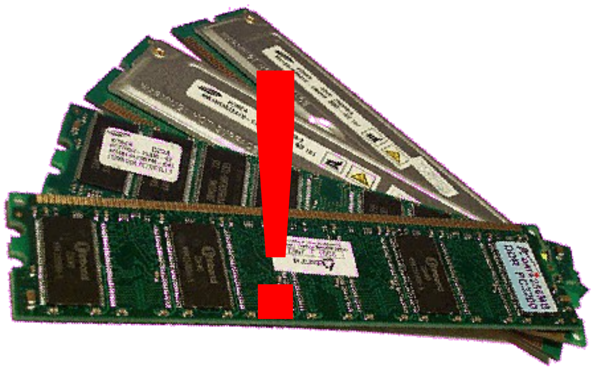



- It's getting worse [2]




[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

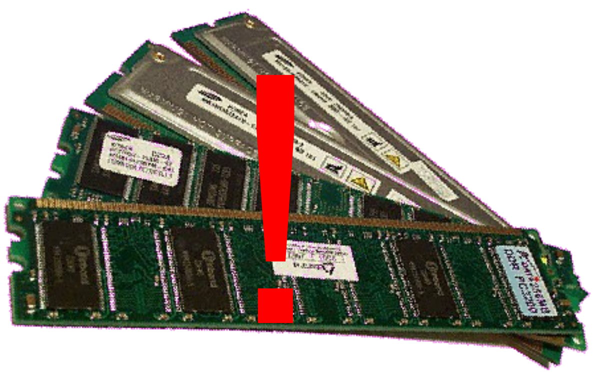
# DB Performance – “It's the memory stupid!” [3]



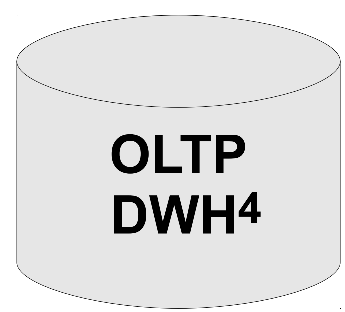
[3] R. Sites. It’s the memory, stupid! MicroprocessorReport, 10(10),1996

# DB Performance – “It's the memory stupid!” [3]

- And worse:
  - Growth rates of main memory size have outstripped the growth rates of structured data in the enterprise [4]
  - Multiple GB main memory DB ...

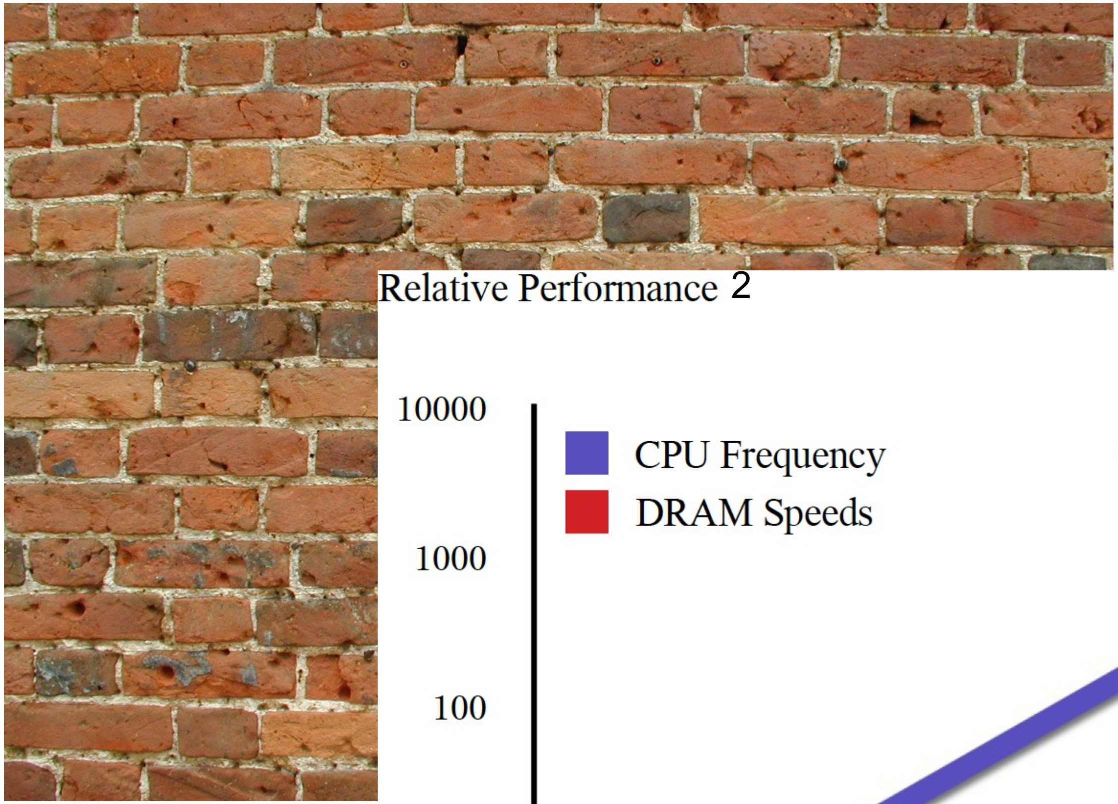


>

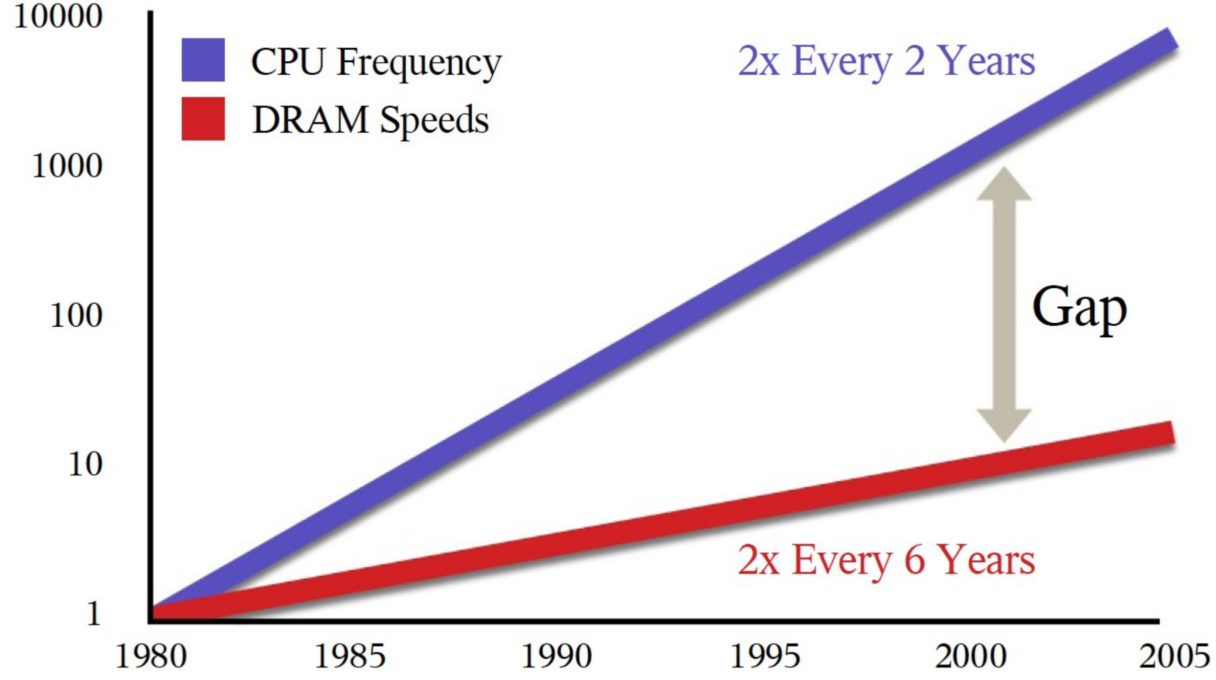



[3] R. Sites. It’s the memory, stupid! MicroprocessorReport, 10(10),1996
[4] K. Schlegel. Emerging Technologies Will Drive Self-Service Business Intelligence. Garter Report 2/08

# The (Memory) Wall [5]

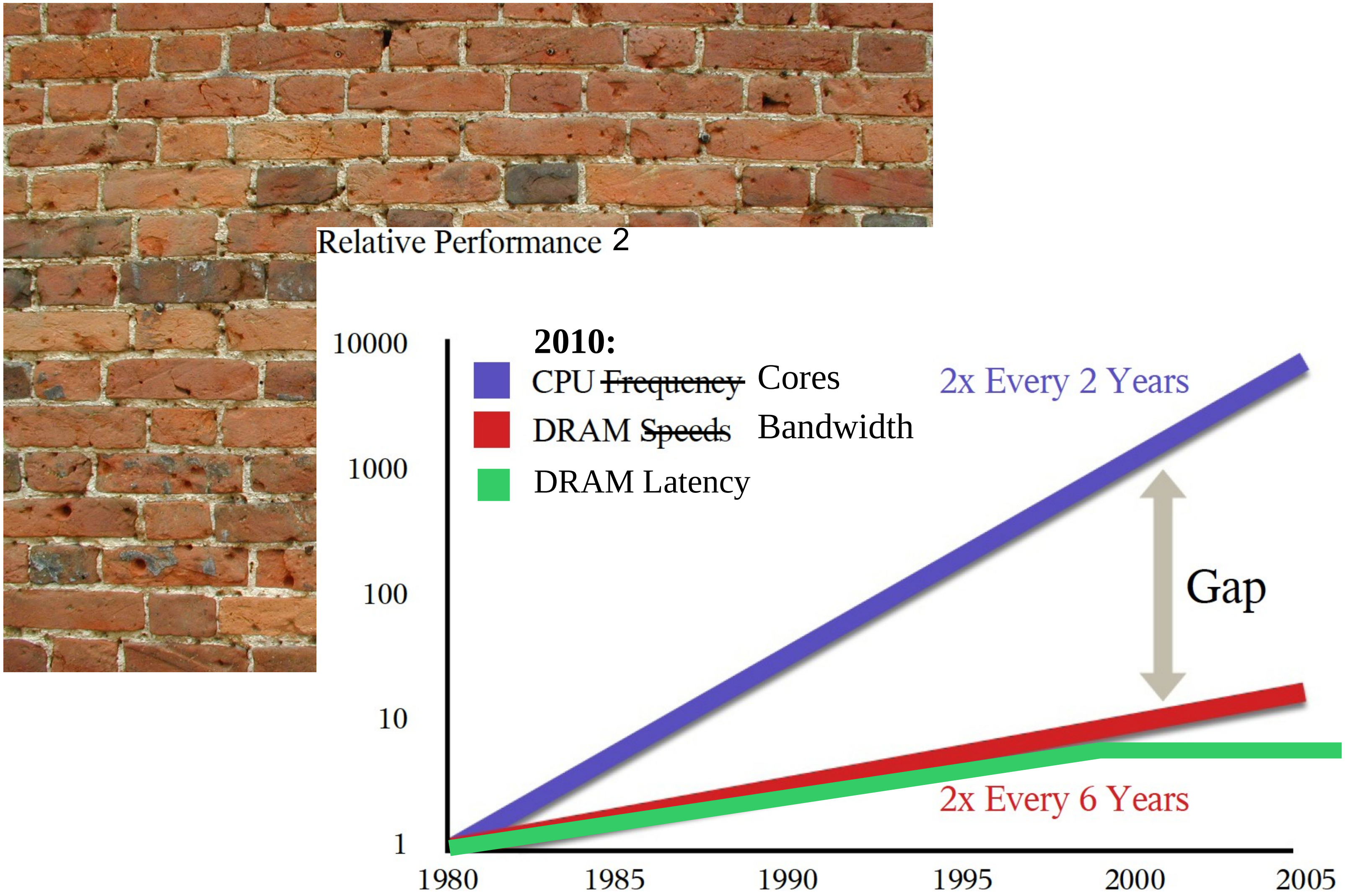


Relative Performance [2]




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

# The (Memory) Wall [5]

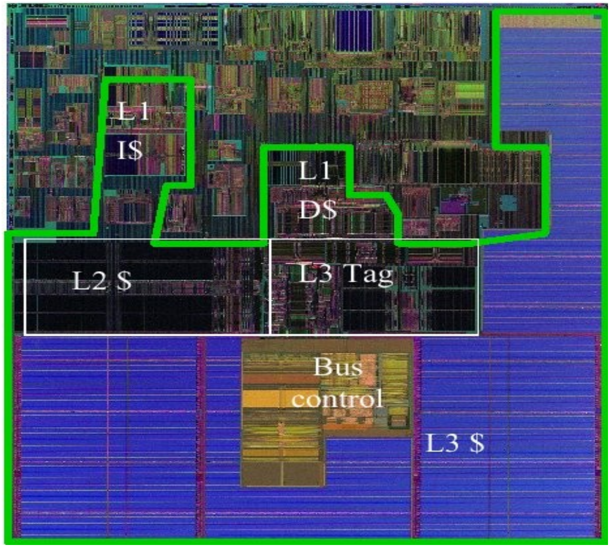



[2] David Yen. Opening Doors to the MultiCore Era. MultiCore Expo 2006

[5] W.A.Wulf et al. Hitting the memory wall: implications of the obvious. SIGARCH - Computer Architecture News'95

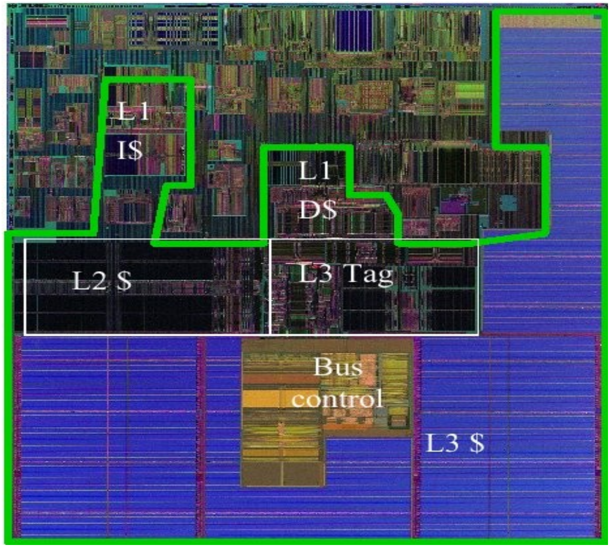
# Overcoming the Memory Wall

- Larger caches
  - Specialized processors
  - Top10 TPC-H – 6/10 use Itanium

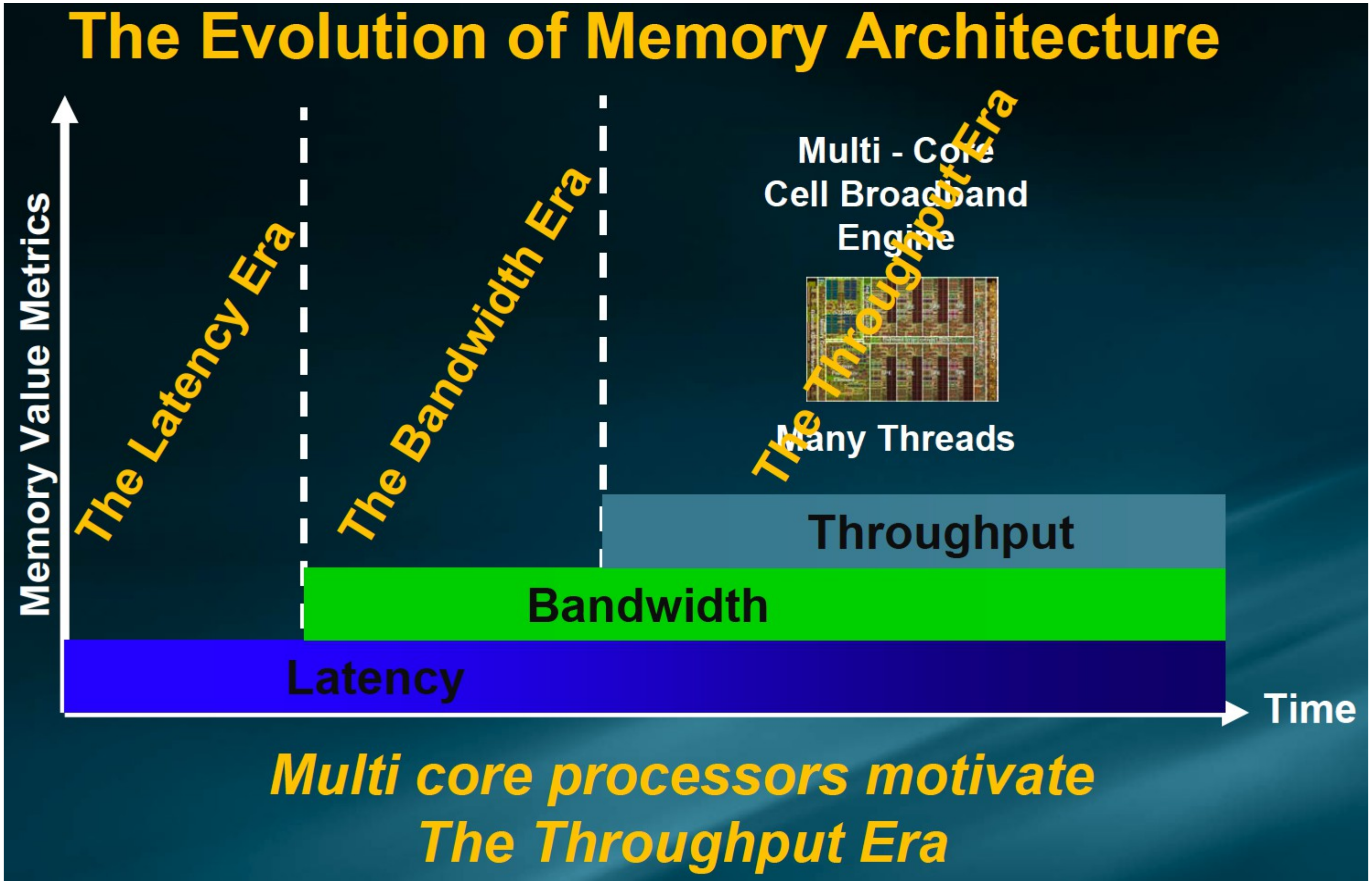


# Overcoming the Memory Wall

- Larger caches
  - Specialized processors
  - Top10 TPC-H – 6/10 use Itanium
- Wait it out?




# Parallel Memory Accesses ➔ Throughput Computing




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

# GPUs as an example for highly parallel architectures

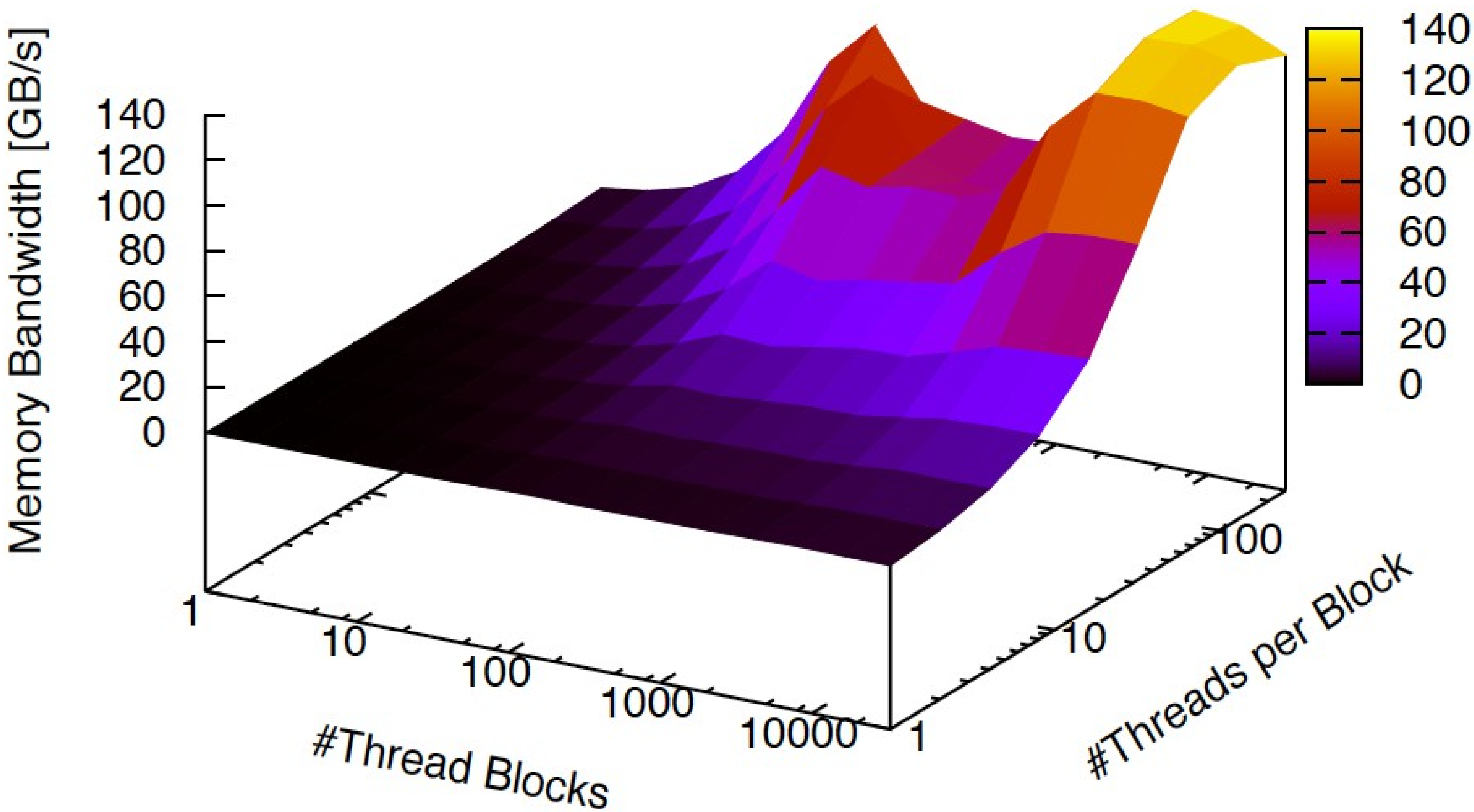
- Besides Teraflop(s) GPU's offer:
  - Massive Parallelism (240 cores)
  - 100+ GB/s memory bandwidth/throughput
  - Better performance per watt and per sqft. than CPUs

# GPU performance specs & measurements



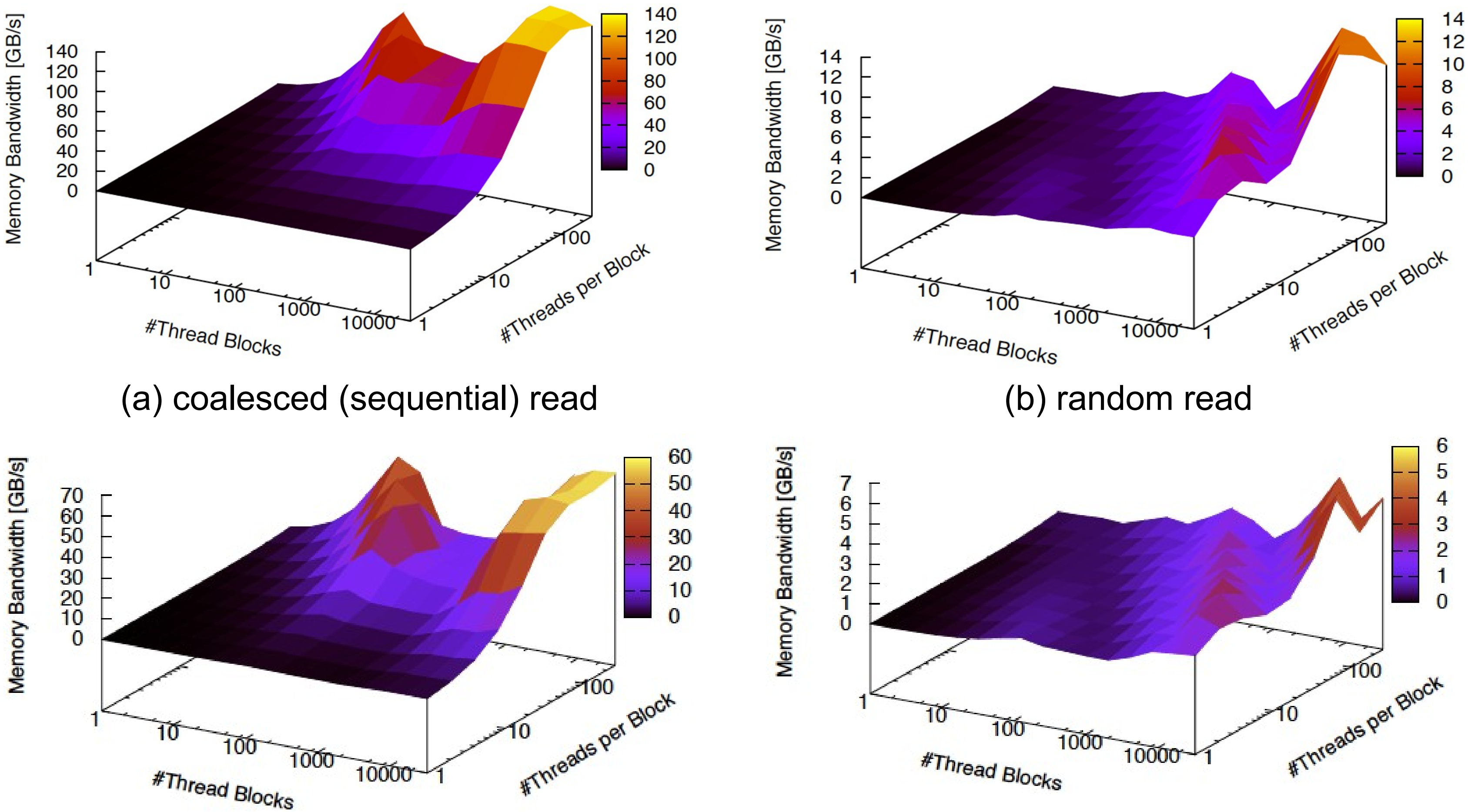
| | GPU (GTX285) | CPU (i7-2600) |
|---|---|---|
| Power consumption | 200 W | 95 W |
| Peak Compute [Spec] | 1063 GFLOP | 109 GFLOP |
| Peak Memory Bandwidth [Spec] | 160 GB/S | 21 GB/s |
| Coalesced/Sequential Read [Measured] | 140 GB/s | 18 GB/s |
| Random Read [Measured] | 8 GB/s | 0.8 GB/s |

# GPU memory bandwidth – it's a throughput machine




Bandwidth of sequential (coalesced) 32-bit read access for multiple thread configurations.
Results for a nVidia GTX 285 1.5GHz, GDDR3 1.2GHZ.

# GPU memory bandwidth




(a) coalesced (sequential) read

(b) random read

(c) coalesced (sequential) write

(d) random write

Parallel memory bandwidth for multiple thread configurations and access patterns. Results for a nVidia GTX 285 1.5GHz, GDDR3 1.2GHZ.

# Agenda

- Introduction
  - GPU & DB (search) ?
- GPU search
  - A first implementation – binary search
  - Conventional search algorithms & GPUs – a mismatch
  - Back to the drawing board:
    - P-ary search
    - Experimental evaluation
    - Why it works
- Conclusions

# A Simple implementation of (index) search

Keyword

| | |
|---|---|
| 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

sorted

| | |
|---|---|
| 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 |

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

sorted

| Keyword |
|---|
| Adam |
| Bethlehem |
| Character |
| Drachenflieger |
| Eva |
| Flughafenbahnhof |
| Grabdenkmal |
| Haubentaucher |

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);
```

- 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, 32 SMs, 8 cores each, 240 cores)
  - Set up an execution configuration & call global function

```
dim3 Dg = dim3(30,0,0);
dim3 Db = dim3(8,0,0);
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

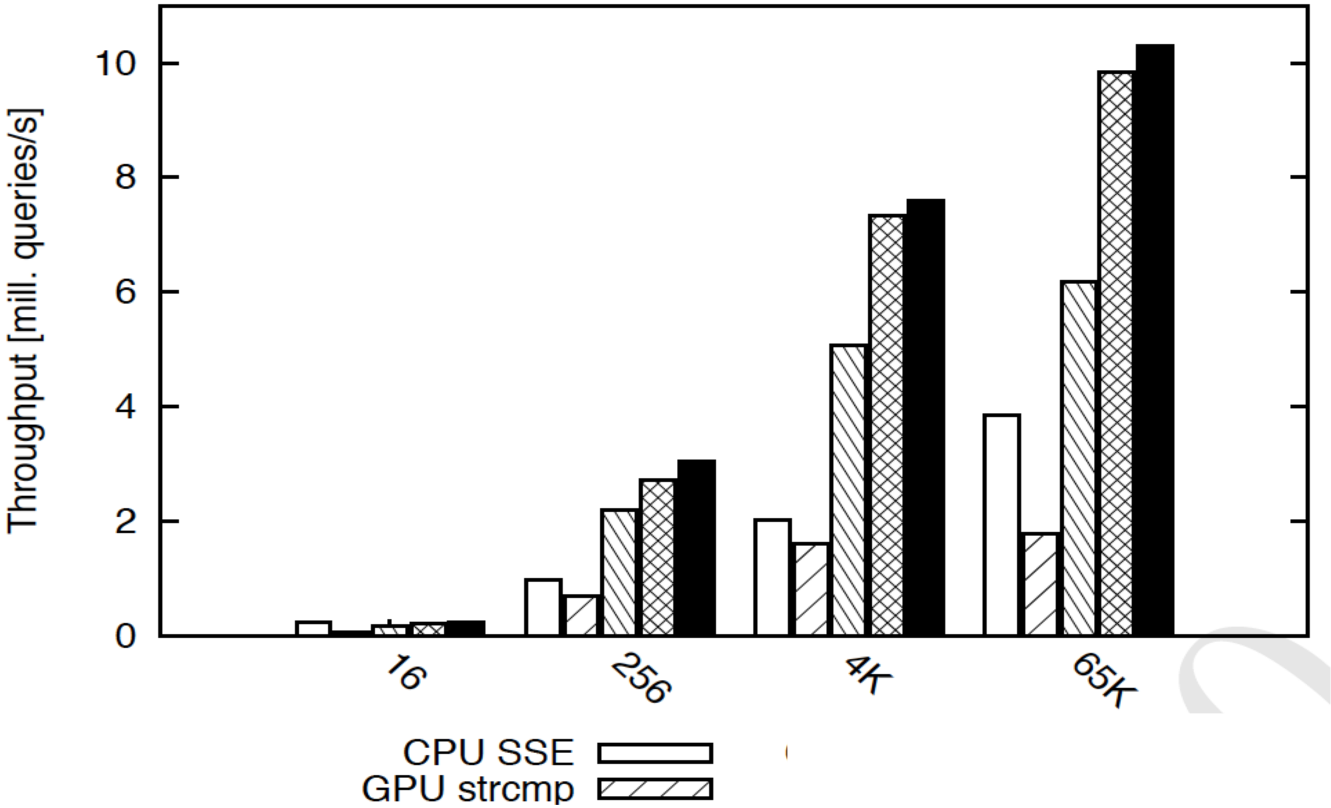
```
__device__ char* bsearchGPU(char *key, char *base, int n, int size){
    char *mid_point;
    int  cmp;

    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));
}
```

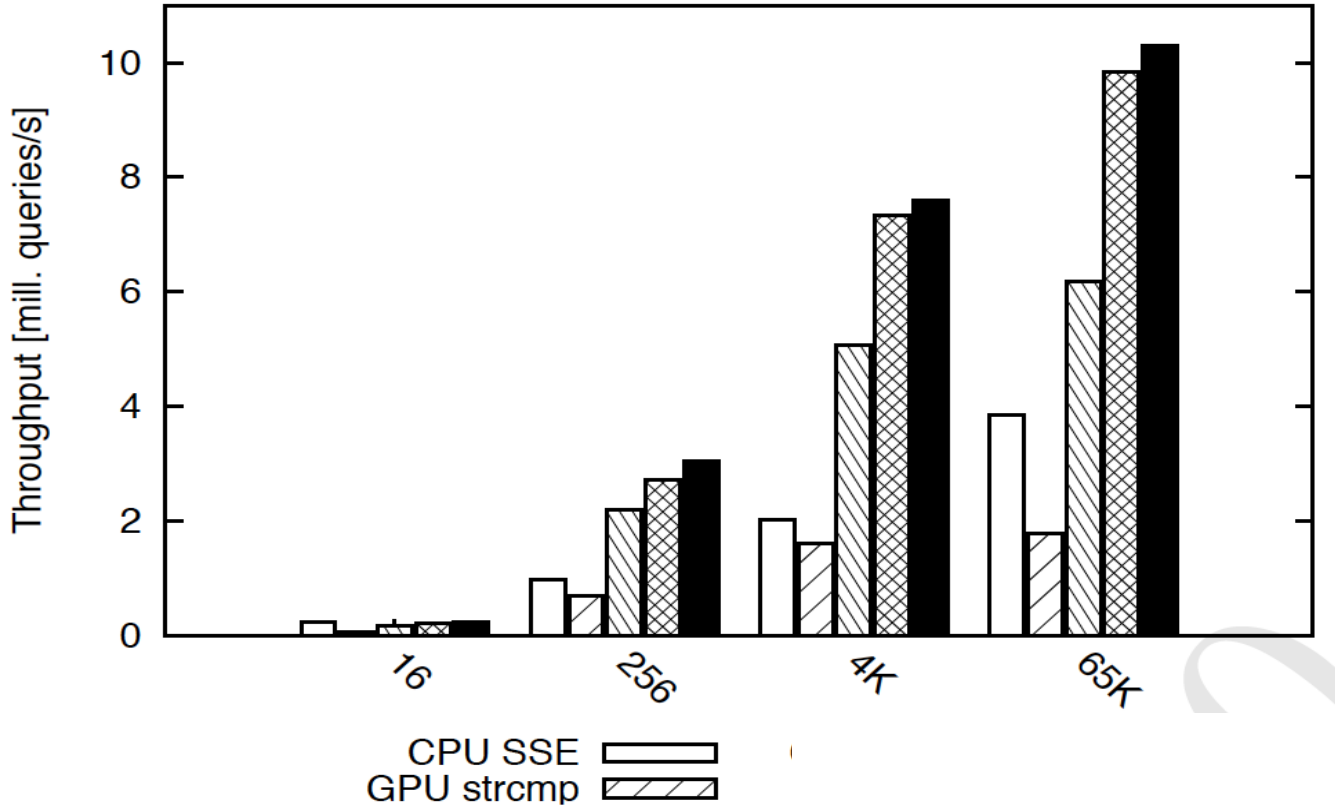
# Binary Search on the GPU

- 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 (225) 16-character strings

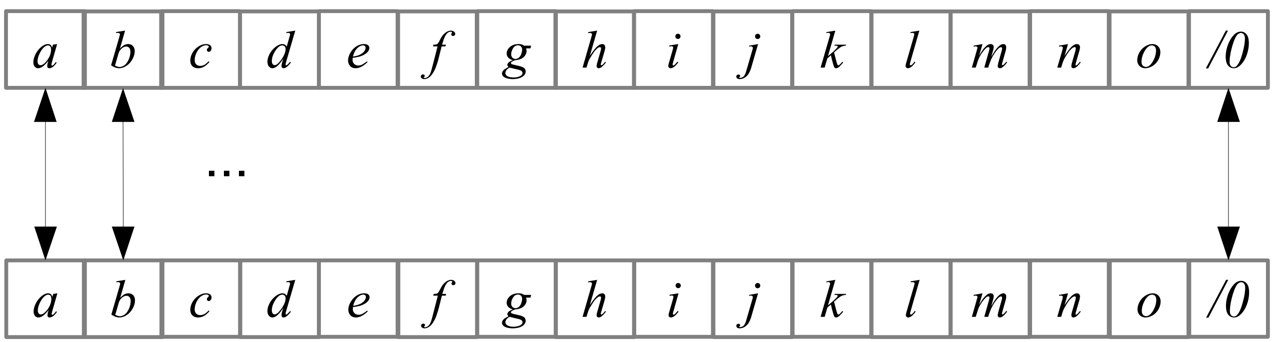



- It's slower than a CPU implementation for all data set sizes!
  - Let's try some optimizations ...

# Search requires to compare

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

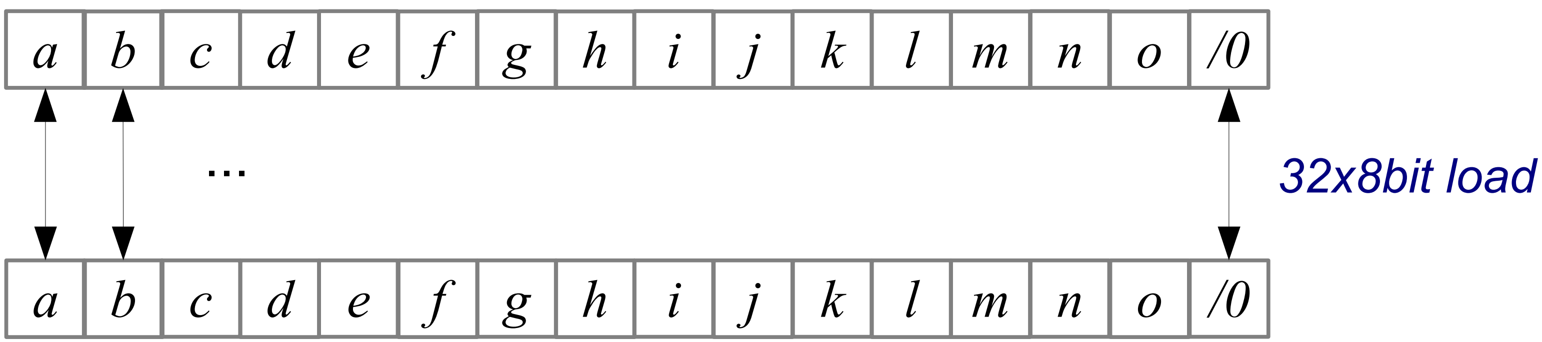
```
int strcmp(const char* s1, const char* s2){
        while (*s1 == *s2++)
                if (*s1++ == 0)return 0;
        return (*s1 - *(s2 - 1));
}
```



# Search requires to compare

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

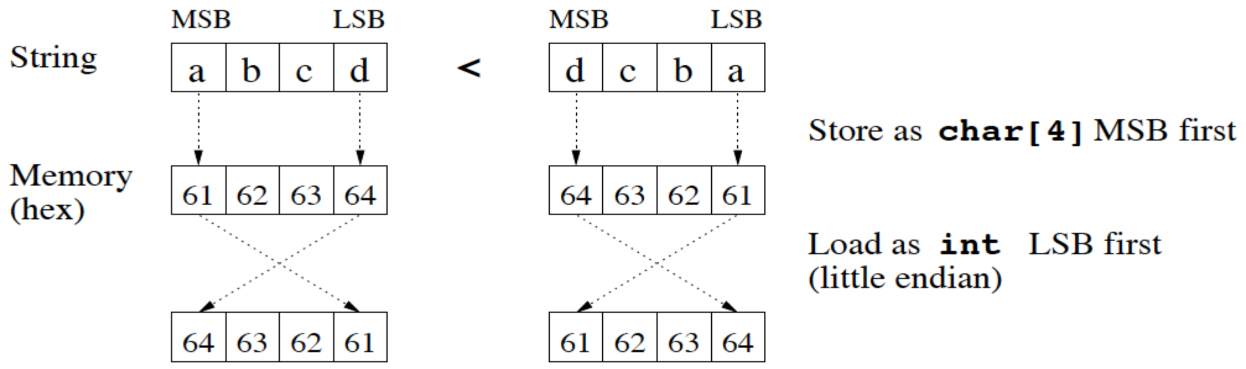
```
int strcmp(const char* s1, const char* s2){
        while (*s1 == *s2++)
                if (*s1++ == 0)return 0;
        return (*s1 - *(s2 - 1));
}
```




- Byte-wise memory access is known to be slow

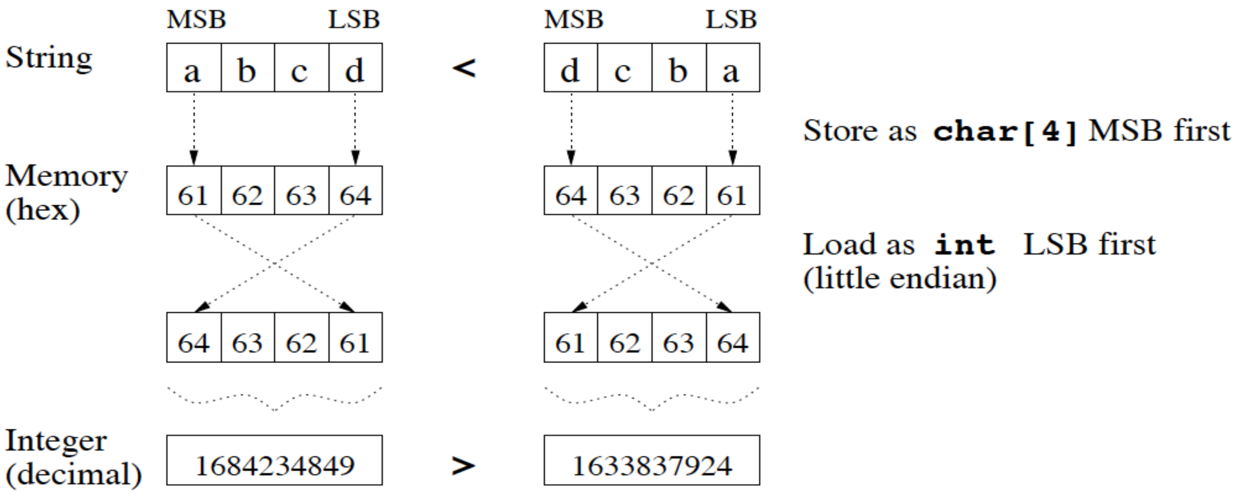
# Optimizing compare operations

- How about vector string comparison, a la SSE?
- No Byte vectors on the GPU … but Integer vectors



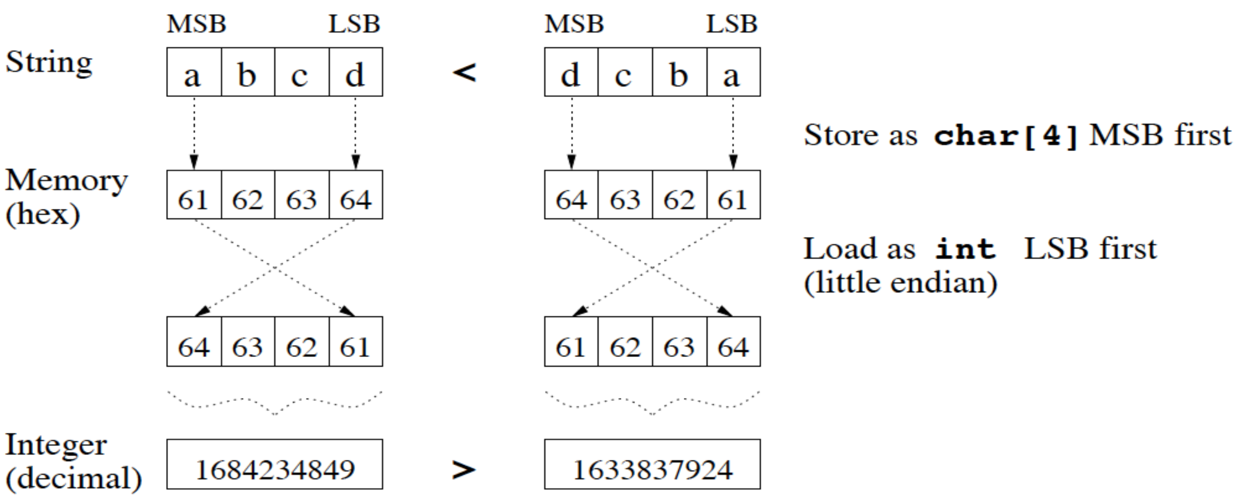
# Optimizing compare operations

- How about vector string comparison, a la SSE?
- No Byte vectors on the GPU … but Integer vectors



# 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

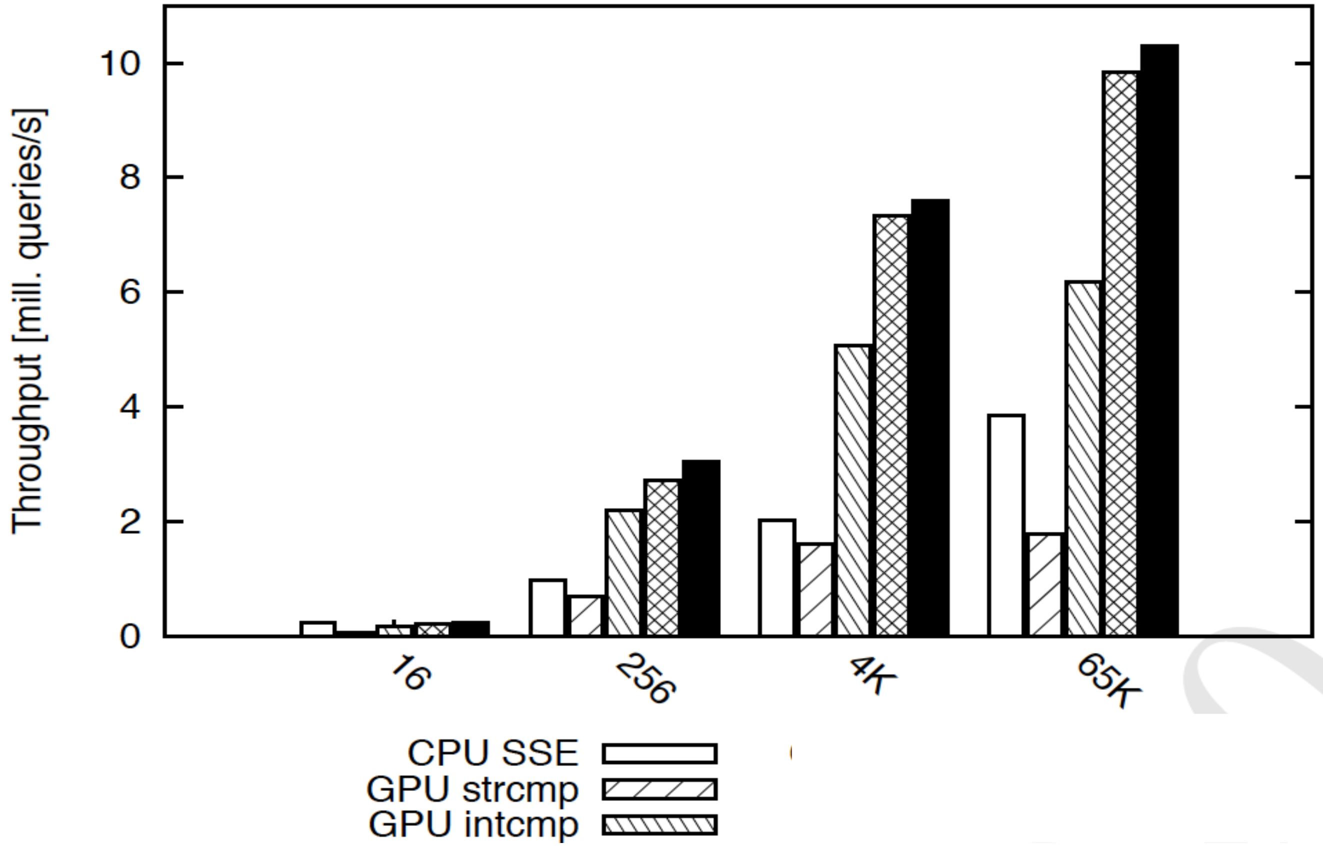
- Comparing integer vectors (bswap for <> skipped for clarity)

```
__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 |

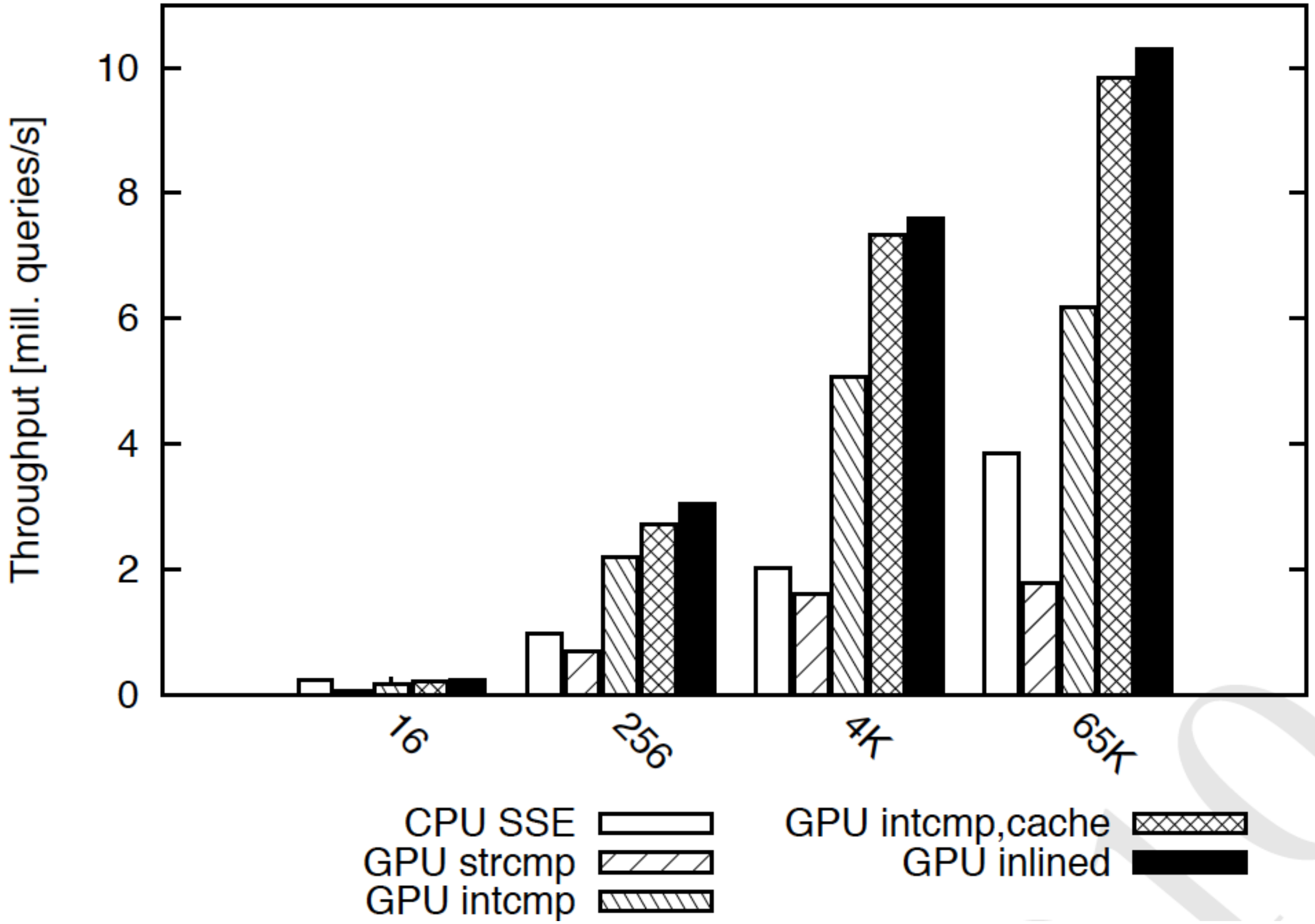
x 16 for each comparison !!!

- 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;
```

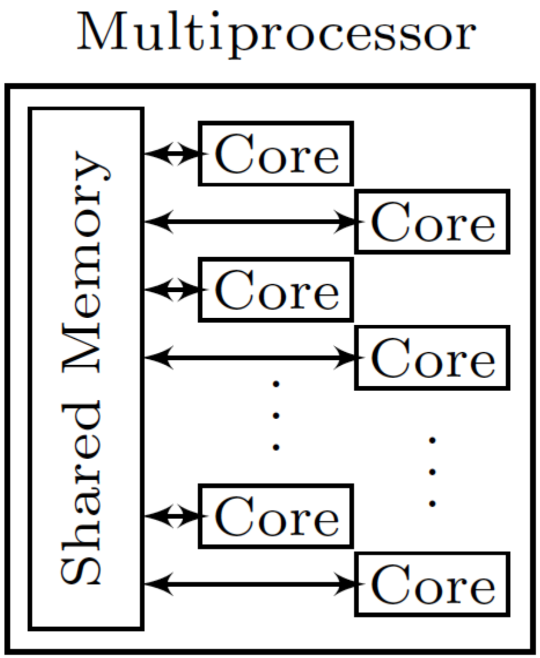
# Binary Search on the GPU – optimized

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



# GPU architecture reminder – SIMD/SIMT

- Inside Streaming Multiprocessor
  - Single Instruction Multiple Threads/Data (SIMT/SIMD)
  - All cores in 1SM execute same instruction or no-op (SIMD threads)
  - Warps of 32 threads (or more, to hide memory latency)




# Multi-threaded Binary Search – Example

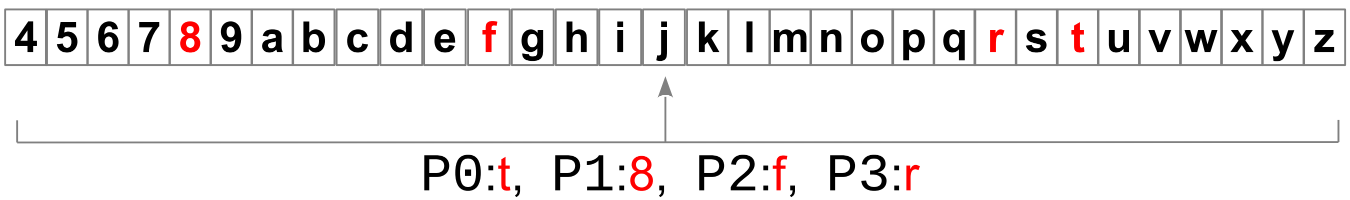
- 1 Index: a sorted char array 32 entries
- 4 queries: t , 8 , f , r
- 4 processor cores: P1-P4
- 1 processor core – 1 search: P0:t , P1:8 , P2:f , P3:r
- Theoretical worst-case execution time: $\log_2(32)=5$

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 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 |

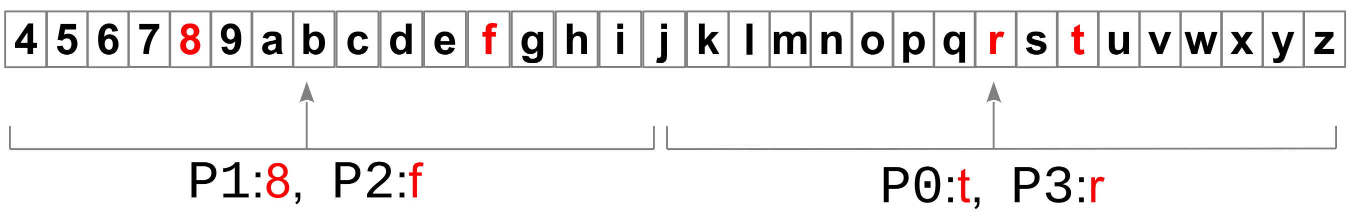
# Multi-threaded Binary Search – Example

- 1 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)




Iter. 2)



# 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?

# Agenda

- Introduction
  - GPU & DB (search) ?
- Porting search to the GPU using CUDA
  - Conventional search and GPU architecture – a mismatch
  - Back to the drawing board:
    - P-ary search – the algorithm
    - Experimental evaluation
    - Why it works
- Conclusions

## Our Goal

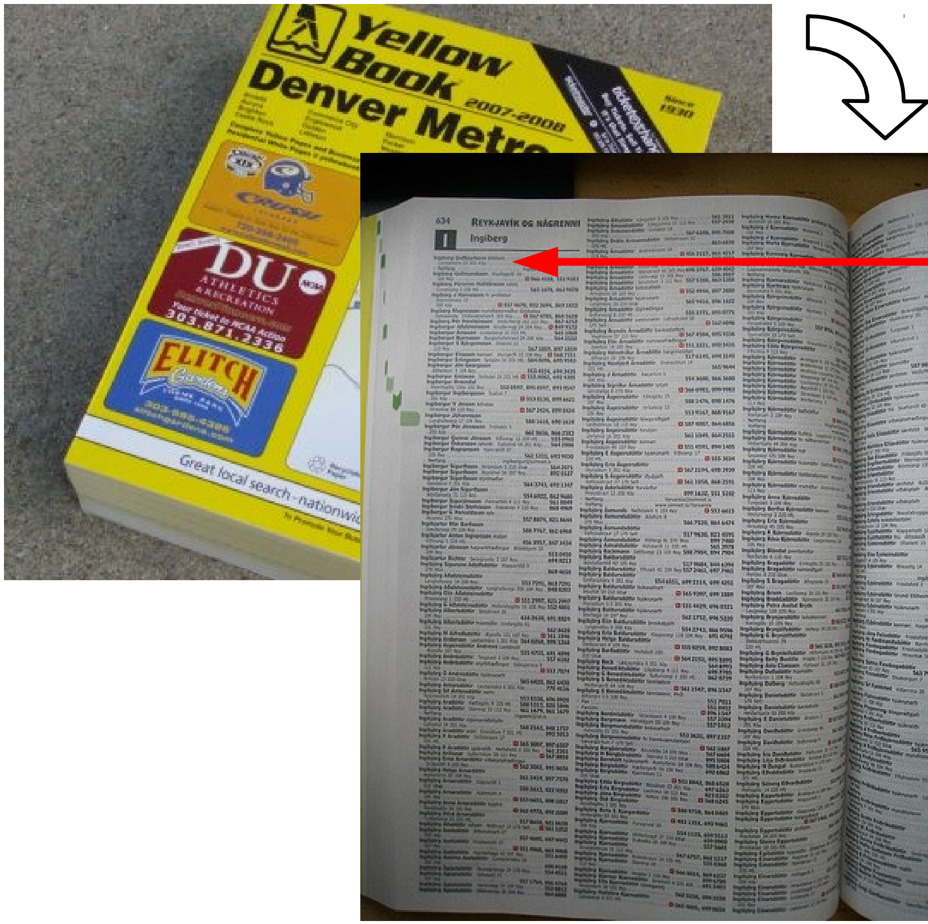
- Improve response time (latency) of core database functions like search in the era of throughput oriented (parallel) computing.

## Research Question

- How can we (algorithmically) exploit parallelism to improve response time (of search)?
  - Can we trade-off throughput for latency?
  - Do we have to trade?

# 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: $\log_2(n/4) < \log_2(n)$

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

# 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 !!

…

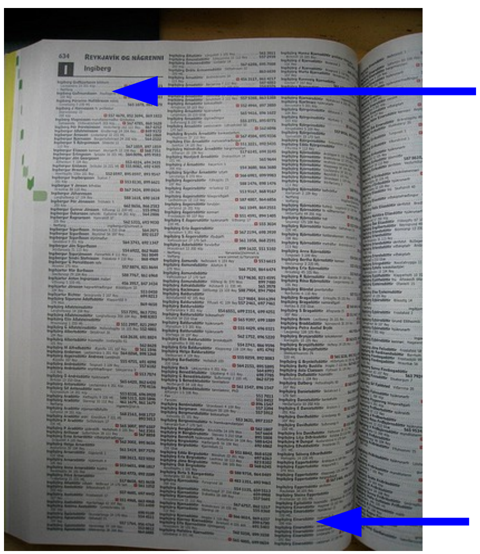
- How do we know who has the right piece ?

# 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 access synchronization

# 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 ?

  || ||

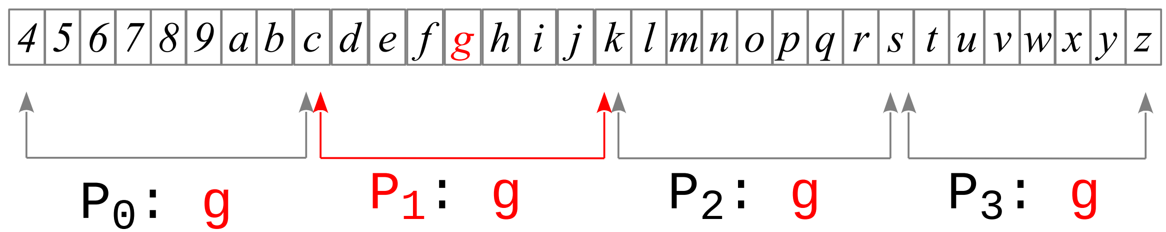
  memory access synchronization

- 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 / processor cores / vector elements

Iteration 1)

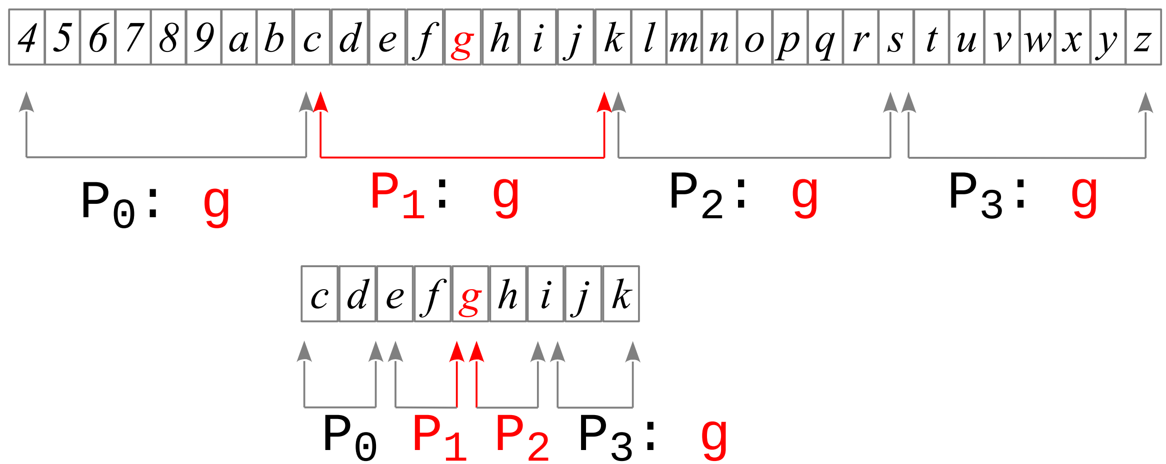


# P-ary Search - Implementation

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

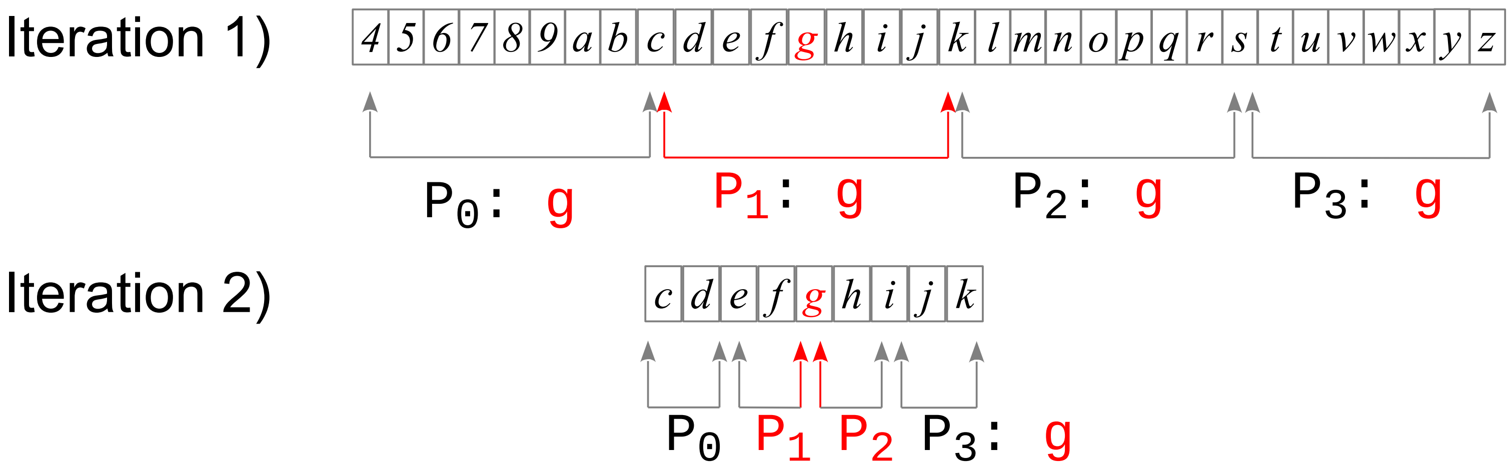
Iteration 1)

Iteration 2)



# P-ary Search - Implementation

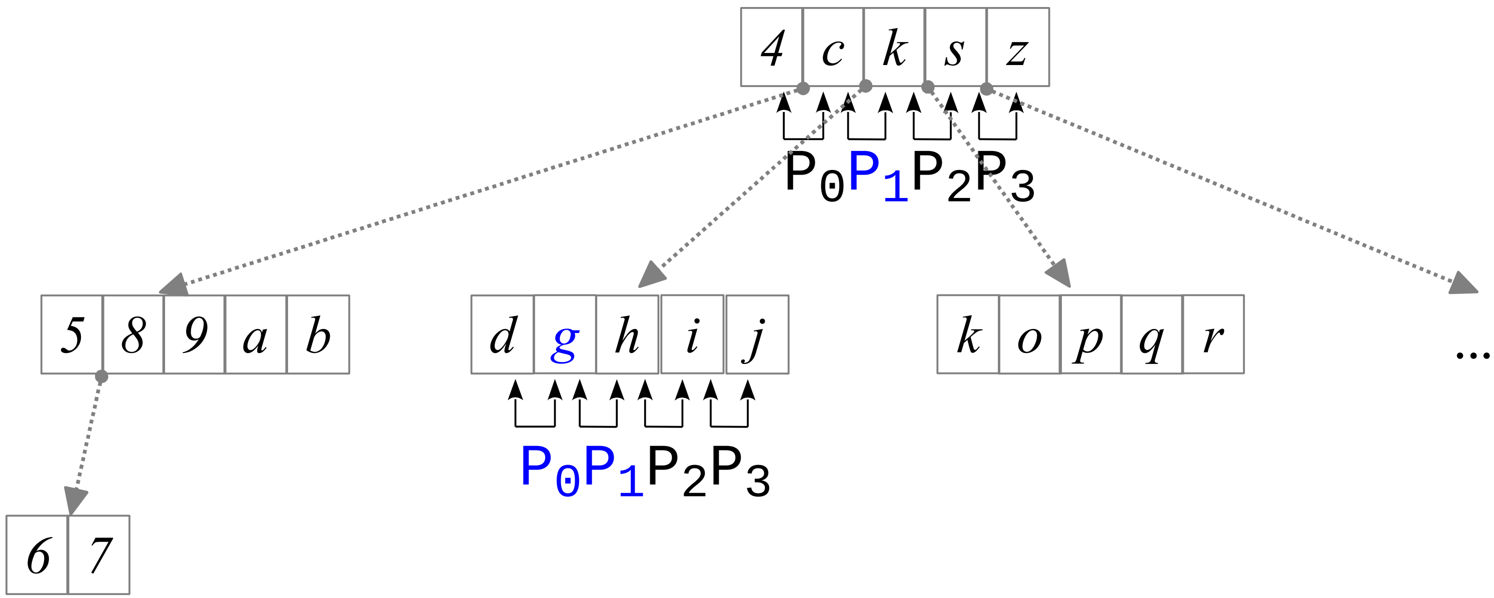
- Strongly relies on fast synchronization
  - friends = threads / processor cores / vector elements




  - 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




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

# P-ary Search on a sorted list – Implementation (1)

```
__global__ void parySearchGPU(int* data , int range_length , int*
                                     search keys , int* results)

   int sk , old_range_length=range_length, range start ;
   // initialize search range starting with the whole data set
   // this is done by one thread
   if (threadIdx.x==0) {
      range_offset=0;
      // cache search key and upper bound in shared memory
      cache[BLOCKSIZE]=0x7FFFFFFF;
      cache[BLOCKSIZE+1]=searchkeys[blockIdx.x];
   }
   // require a sync, since each thread is going to
   // read the above now
   Syncthreads();
   sk = cache[BLOCKSIZE+1];
```

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

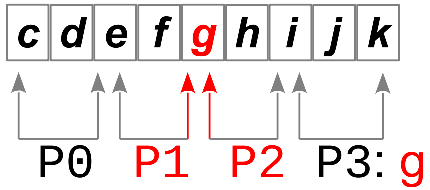
```
    // repeat until the #keys in the search range is
    // smaller than the number of threads
    while (range_length>BLOCKSIZE){
        // calculate search range for this thread
        // avoiding floating point operations
        range_length = range_length/BLOCKSIZE;
        if (range_length * BLOCKSIZE < old_range_length)
            range_length+=1;
        old_range_length=range_length;
        range_start = range_offset + threadIdx.x * range_length;

        // cache the boundary keys
        cache[threadIdx.x]=data[range_start];
        __syncthreads();
        // if the seached 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]){
            range_offset = range_start;
        }
        // all threads need to start next iteration
        // with the new subset
        __syncthreads();
    }
```

# P-ary Search on a sorted list – Implementation (3)

```
    // last iteration
    range_start = range_offset + threadIdx.x;
    if (sk==data[range_start])
        results[blockIdx.x]=range_start;
    }
```

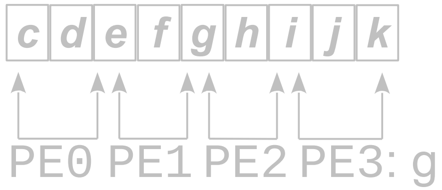
# P-ary Search – Analysis

- 100% processor utilization for each query
- Multiple threads can find a result
  - Does not change correctness



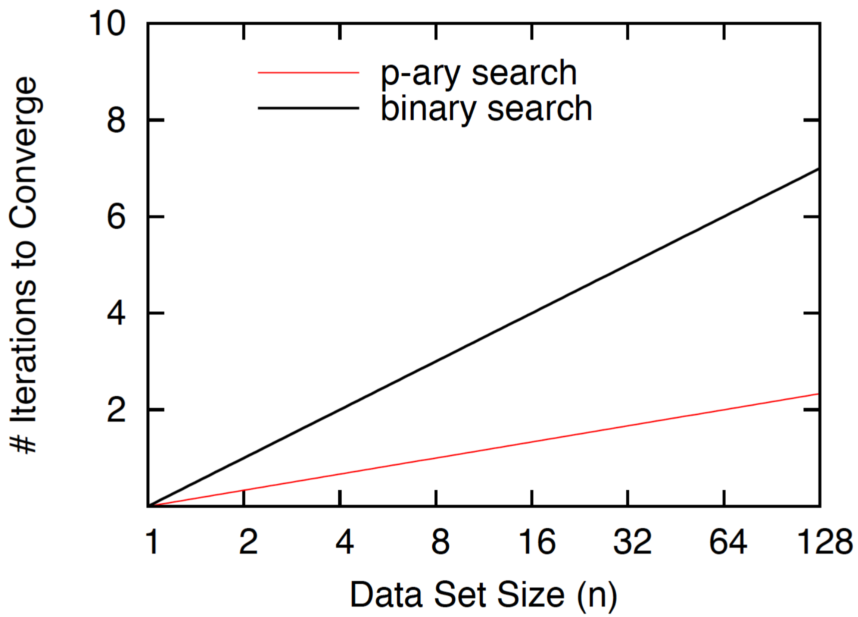
# P-ary Search – Analysis

- 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) → p=8
- Better Response time
  - $\log_p(n)$ vs $\log_2(n)$

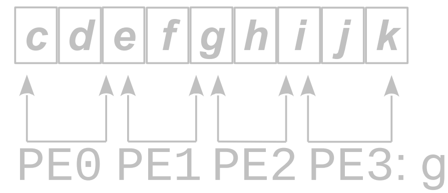


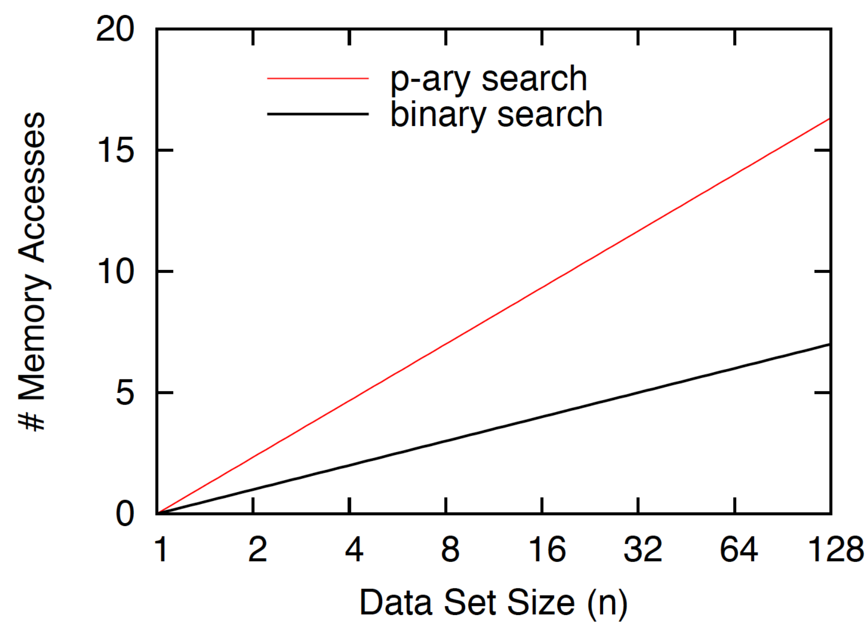
# P-ary Search – Analysis

- 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) → p=8
- Better Response time
  - $\log_p(n)$ vs $\log_2(n)$
- More memory access
  - (p*2 per iteration) * $\log_p(n)$
  - Caching

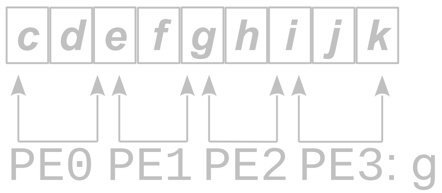
    (p-1) * $\log_p(n)$ vs. $\log_2(n)$

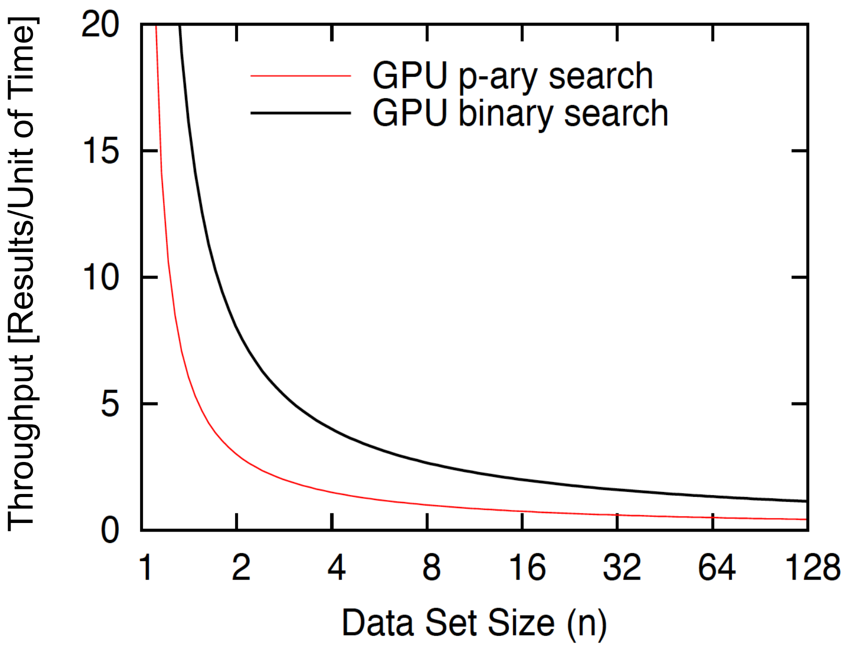


# P-ary Search – Analysis

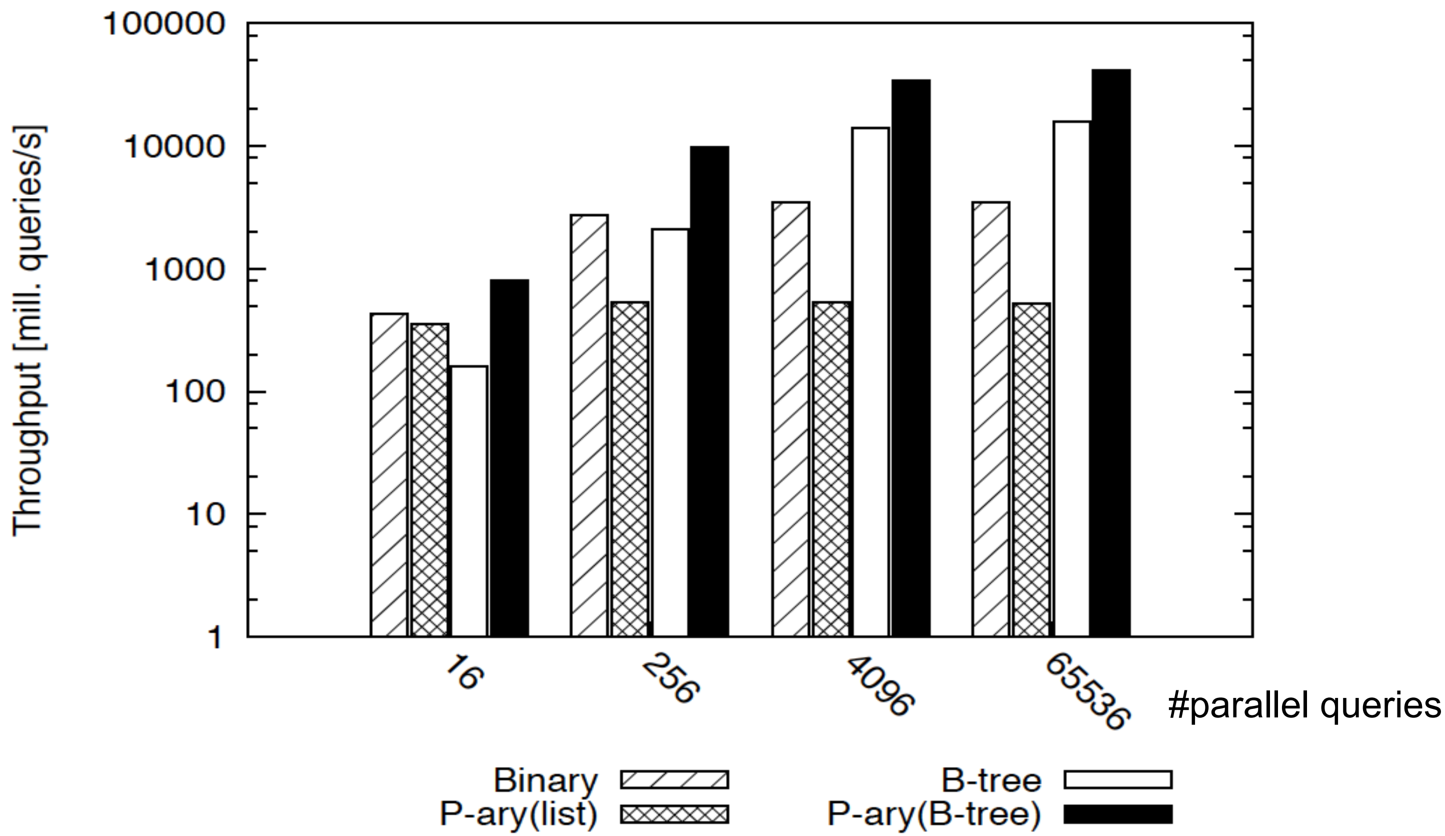
- 100% processor utilization for each query
- Multiple threads can find a result
  - Does not change correctness
- Convergence depends on #threads
  GTX285: 1 SM, 8 cores(threads) → p=8
- Better Response time
  - $\log_p(n)$ vs $\log_2(n)$
- More memory access
  - p*2 per iteration * $\log_p(n)$
  - Caching
    $(p-1) * \log_p(n)$ vs. $\log_2(n)$
- Lower Throughput
  - $1/\log_p(n)$ vs $p/\log_2(n)$



# P-ary Search (GPU) – Throughput

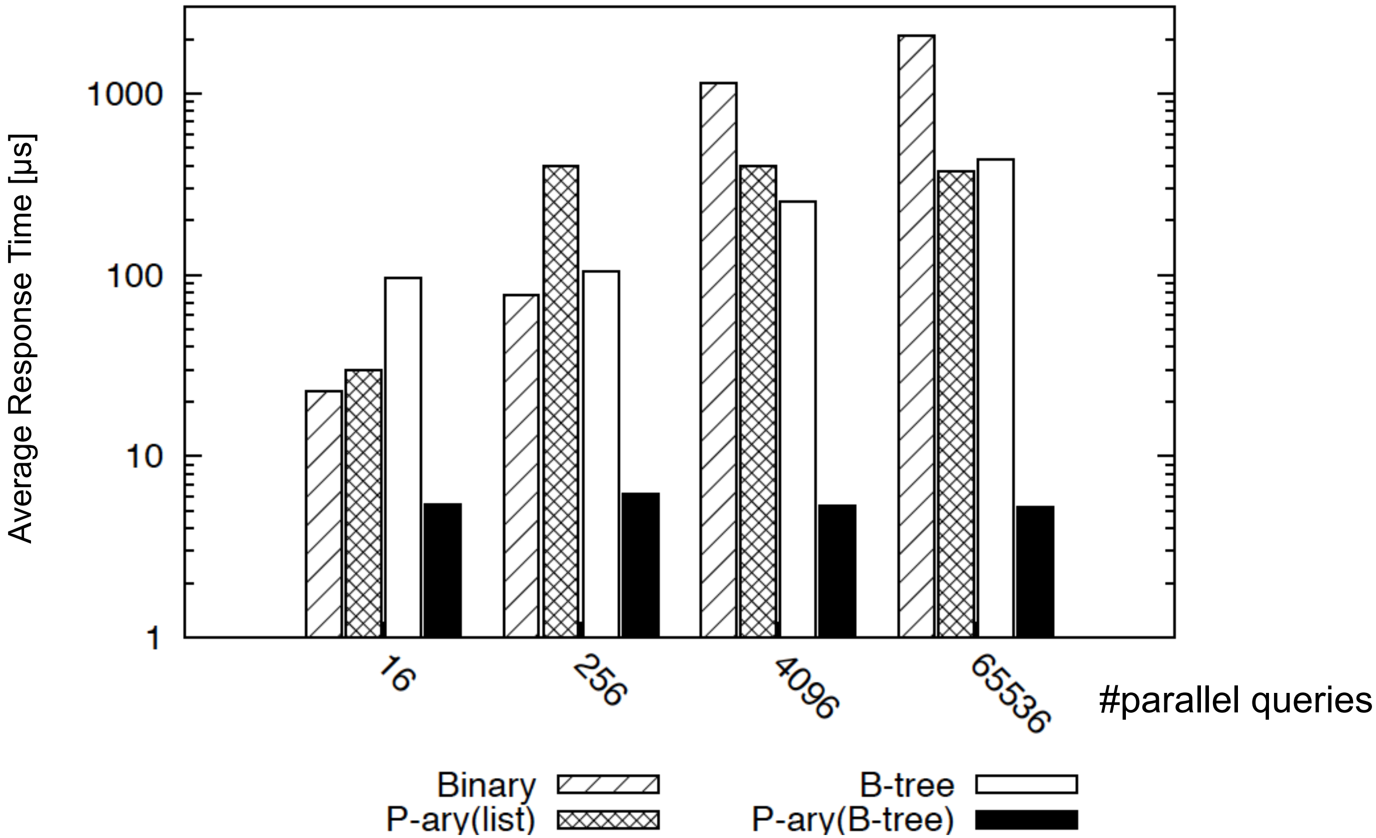
- Superior throughput compared to conventional algorithms




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

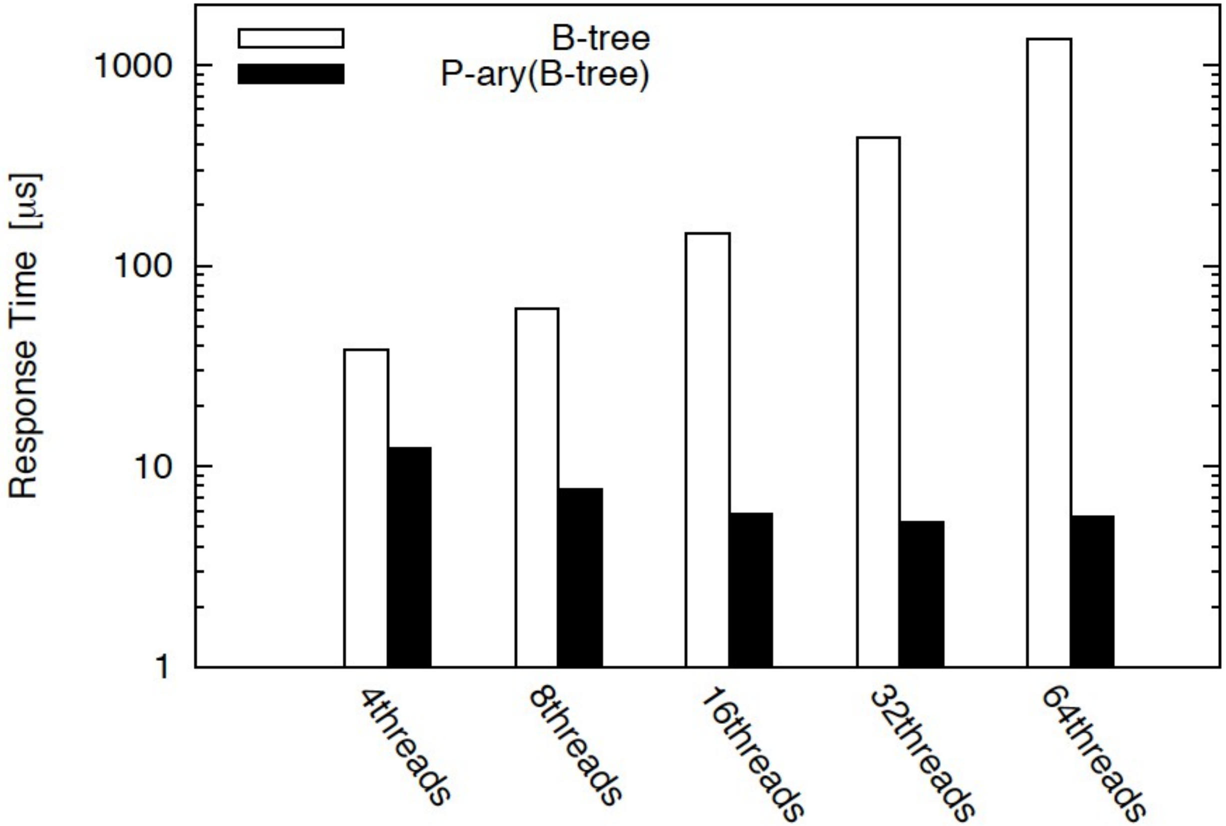
- Response time is workload independent for B-tree implementation




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

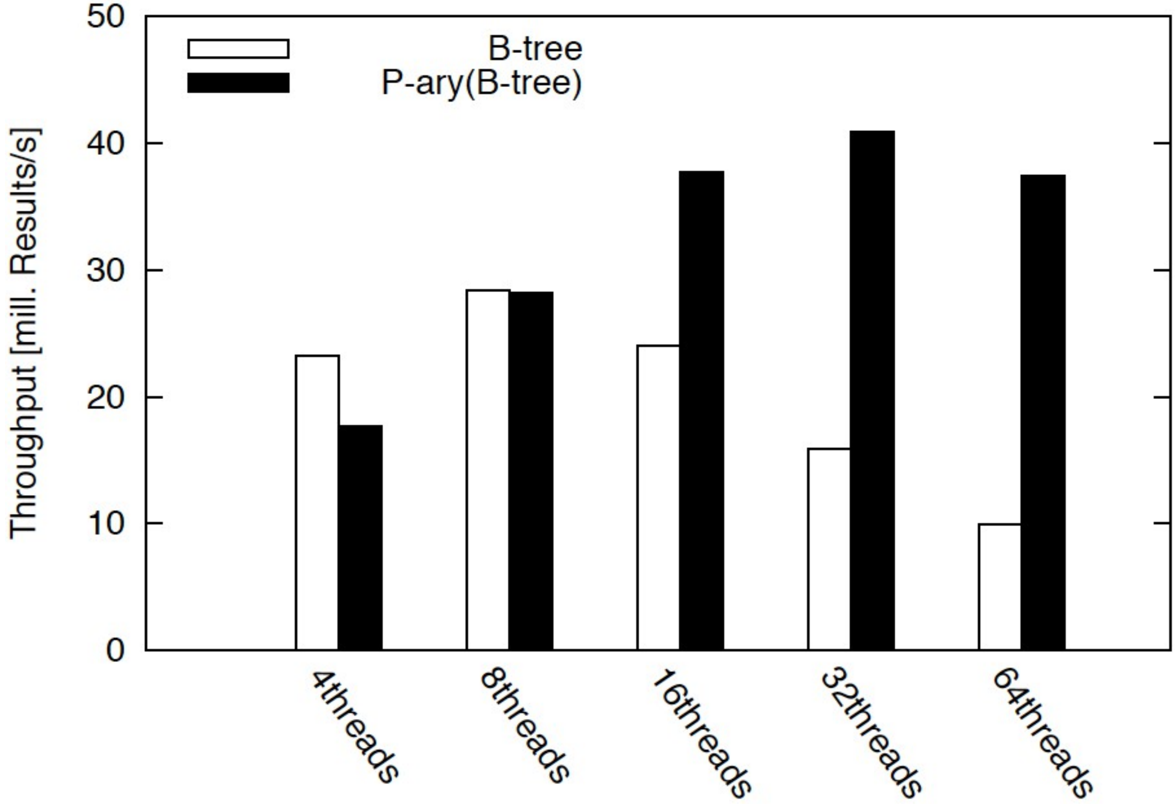
- GPU Implementation using SIMT (SIMD threads)
- Scalability with increasing #threads (P)




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

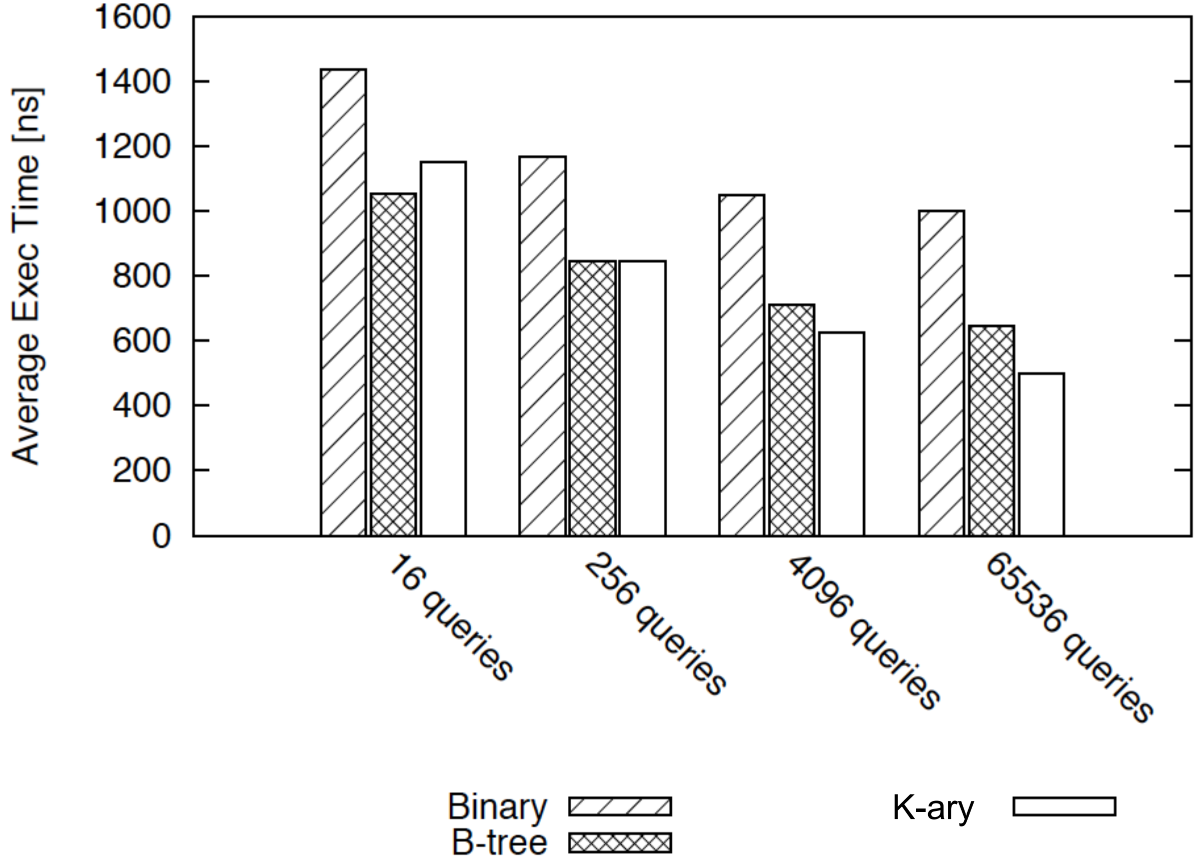
- GPU Implementation using SIMT (SIMD threads)
- Scalability with increasing #threads (P)




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

- K-ary[1] search is the same algorithm ported to the CPU using SSE vectors (int4) → convergence rate $\log_4(n)$

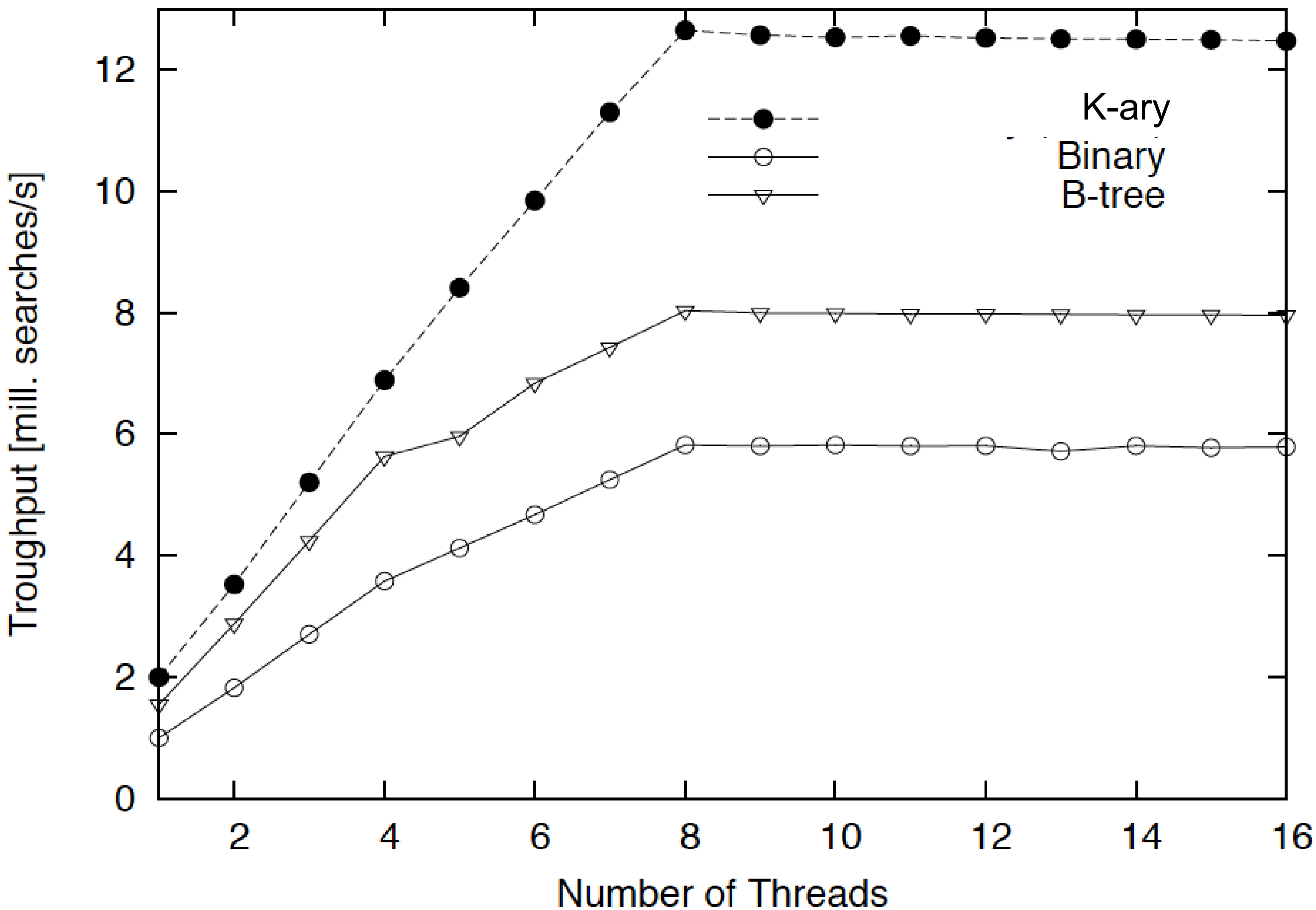



Searching a 512MB data set with 134mill. 4-byte integer entries, Core i7 2.66GHz, DDR3 1666.

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

# P-ary Search(CPU) = K-ary Search

- 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

# P-ary search - an architecture perspective

- Architecture trends
  - Memory latency has bottomed out more than a decade ago
  - Parallel memory bandwidth keeps increasing
    - e.g. Core 2 8GB/s, Core i7 24GB/s (10GB/s per core)
  - Multi-core is just the beginning, many-core is the future
  - Cache per core keeps decreasing (GPU, no caches)
    - Linear (coalesced) memory accesses take its place
  - Core/ thread synchronization costs keep decreasing

➔ Only thing to hope for are increases in parallel memory bandwidth

# P-ary search - an architecture perspective

- Architecture trends
  - Memory latency has bottomed out more than a decade ago
  - Parallel memory bandwidth keeps increasing
    - e.g. Core 2 8GB/s, Core i7 24GB/s (10GB/s per core)
  - Multi-core is just the beginning, many-core is the future
  - Cache per core keeps decreasing (GPU, no caches)
    - Linear (coalesced) memory accesses take its place
  - Core/ thread synchronization costs keep decreasing

➔ Only thing to hope for are increases in parallel memory bandwidth

- P-ary search was designed under this premises and provides
  - Scalable performance – fast thread synchronization
  - Reduced query response time – parallel memory access
  - Increased throughput – coalesced memory access
  - Workload independent constant query execution time

# Questions