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

Google™

YAHOO!®

?

Quo Vadis ?

+ ORACLE special projects

+

Quo Vadis ?

+ ORACLE® special projects

+

=

Agenda

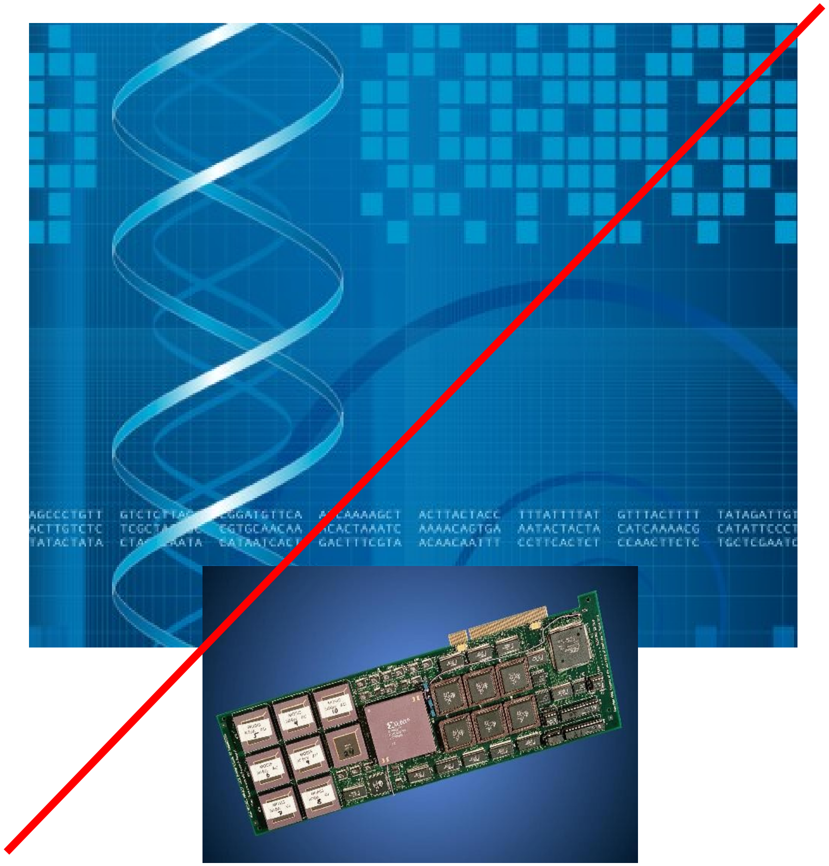
- Introduction
  - GPU & DB search ?

- Porting search to the GPU using CUDA
  - Conventional search on GPU architecture – a mismatch
  - Back to the drawing board:
    - P-ary search – the algorithm
    - Experimental evaluation
    - Why it works

- Conclusions

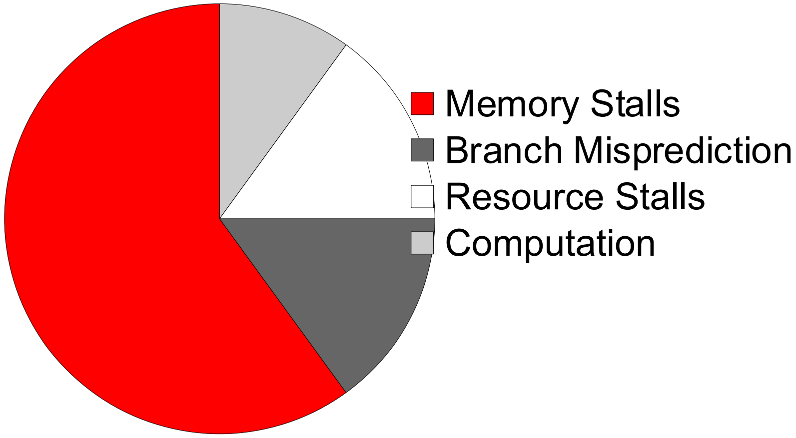
Database Workloads

- 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

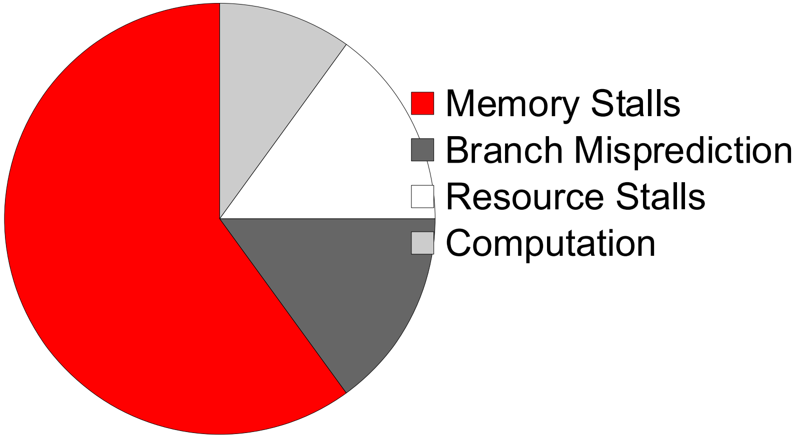


---

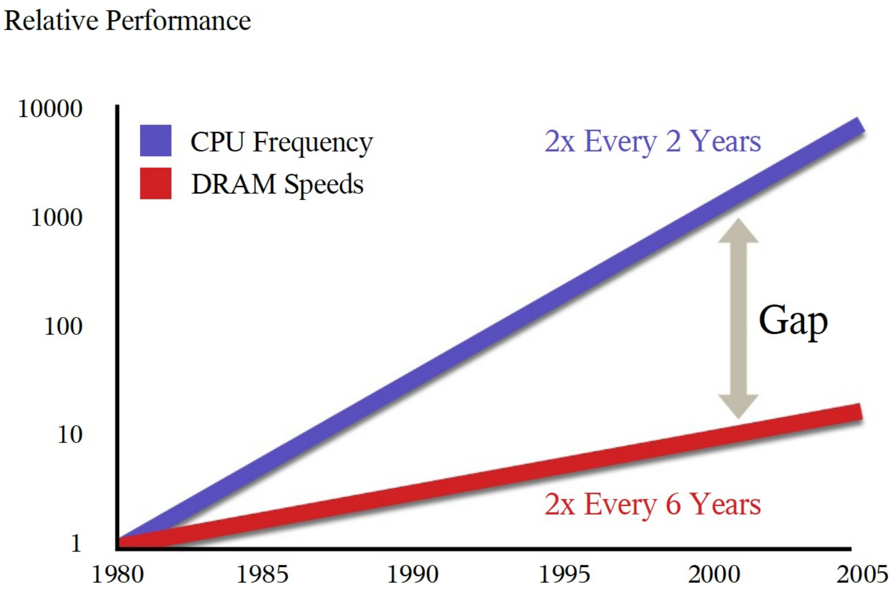
[1] A. Ailamaki, et al. DBMSs on a modern processor: Where does time go? VLDB’99

DB Performance – Where does Time Go

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



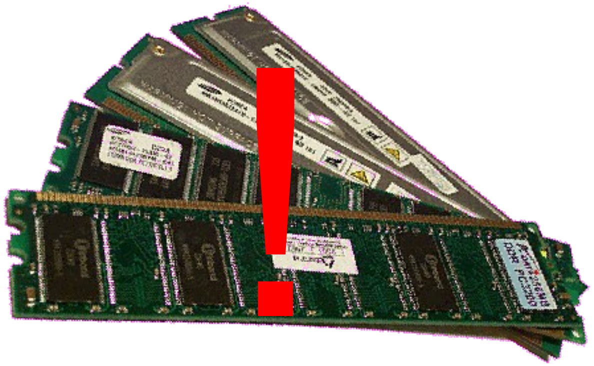
- It's getting worse [2]



---

[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]

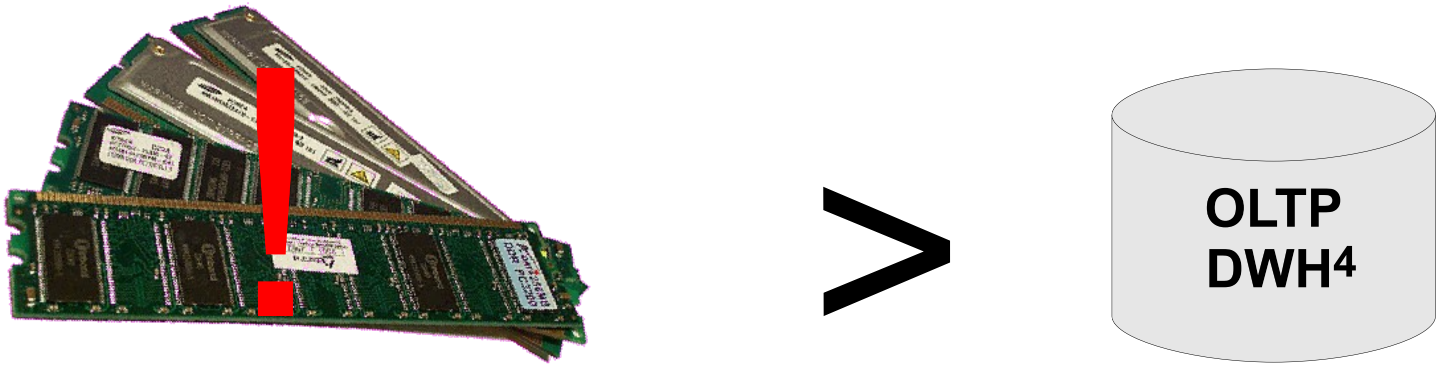


---

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

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

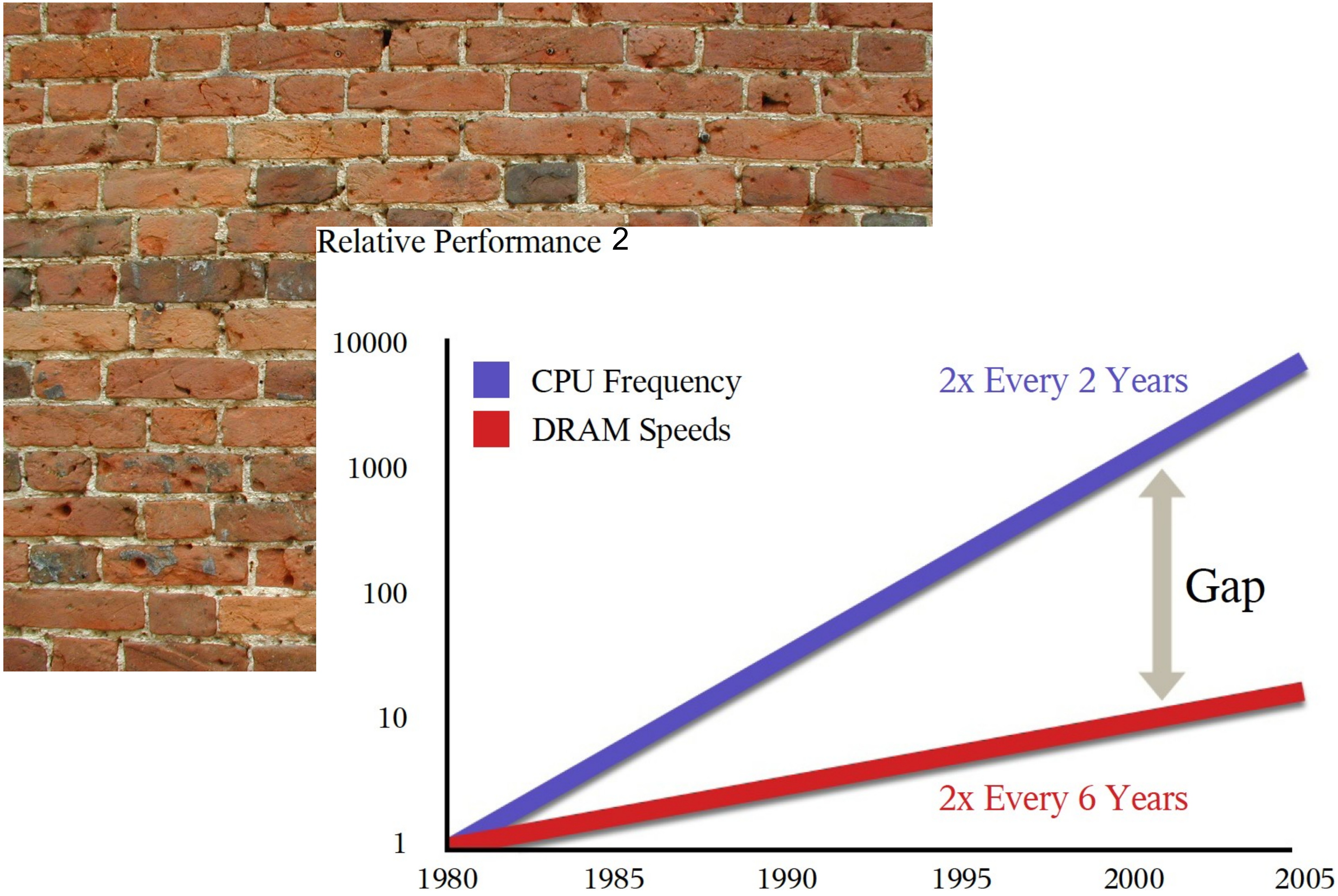
- And worse:
  - Growth rates of main memory size have outstripped the growth rates of structured data in the enterprise[4]
  - Multiple GB main memory DB ...



---

[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]



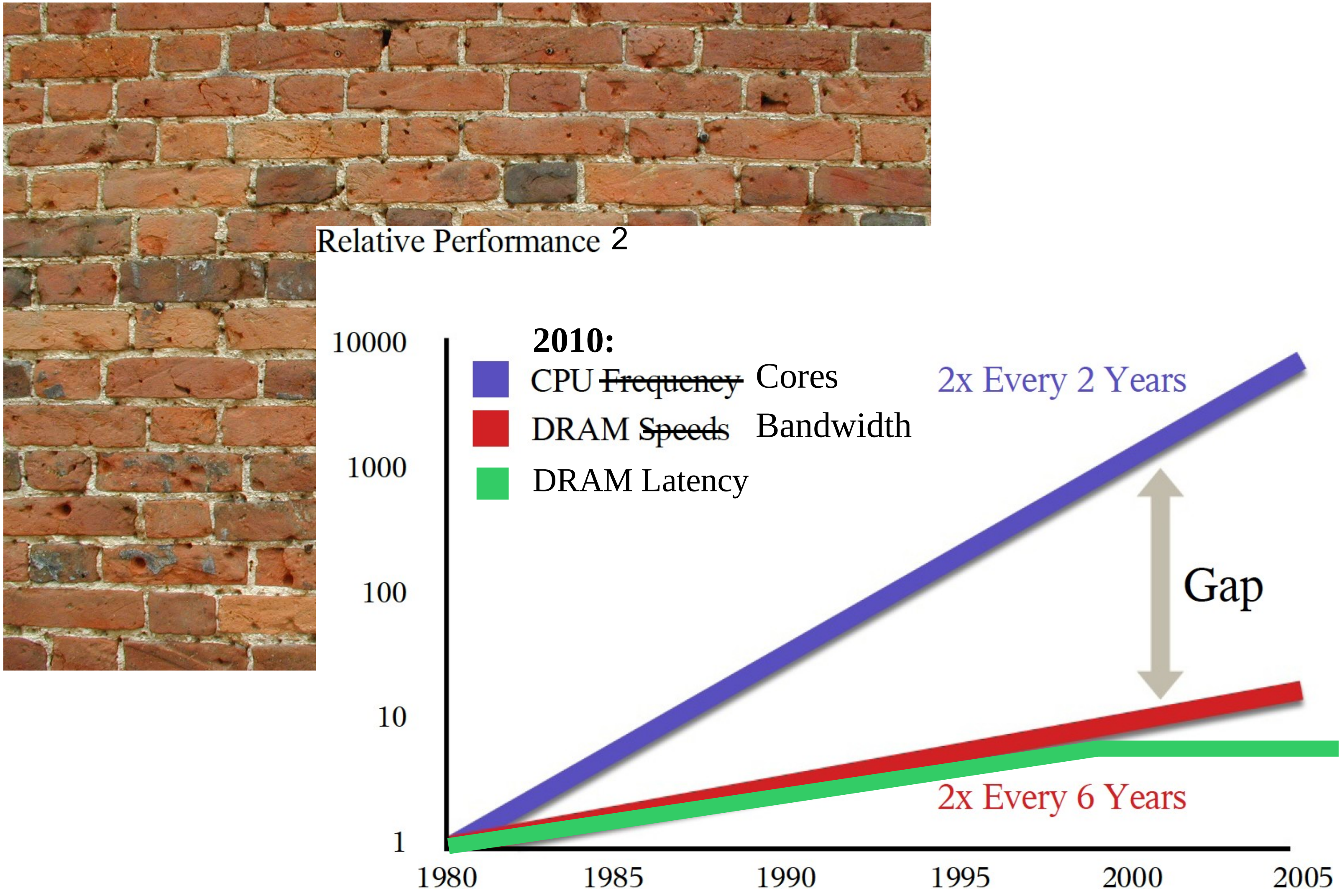
Relative Performance [2]

---

[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]



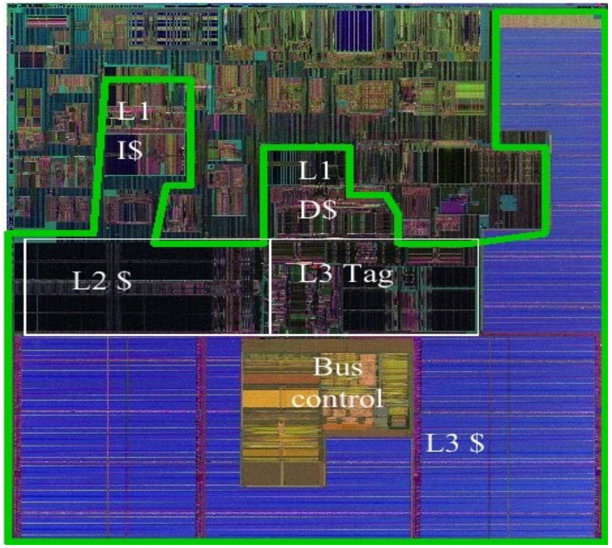
---

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

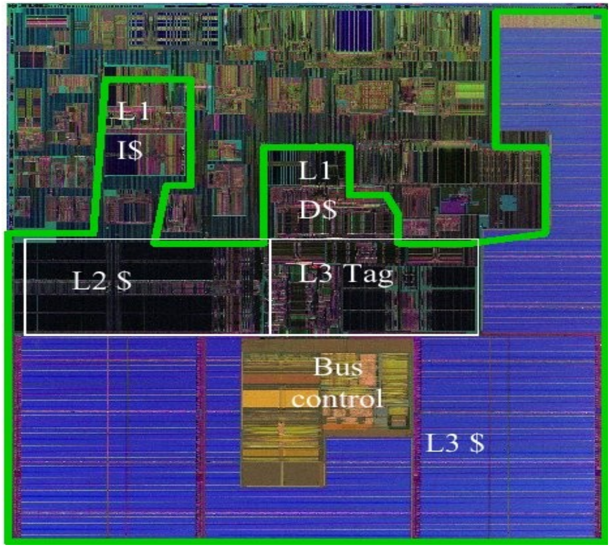
[5] W.A.Wulf et al. Hitting the memory wall: implications of the obvious. SIGARCH - Computer Architecture News'95

Overcoming the Memory Wall

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

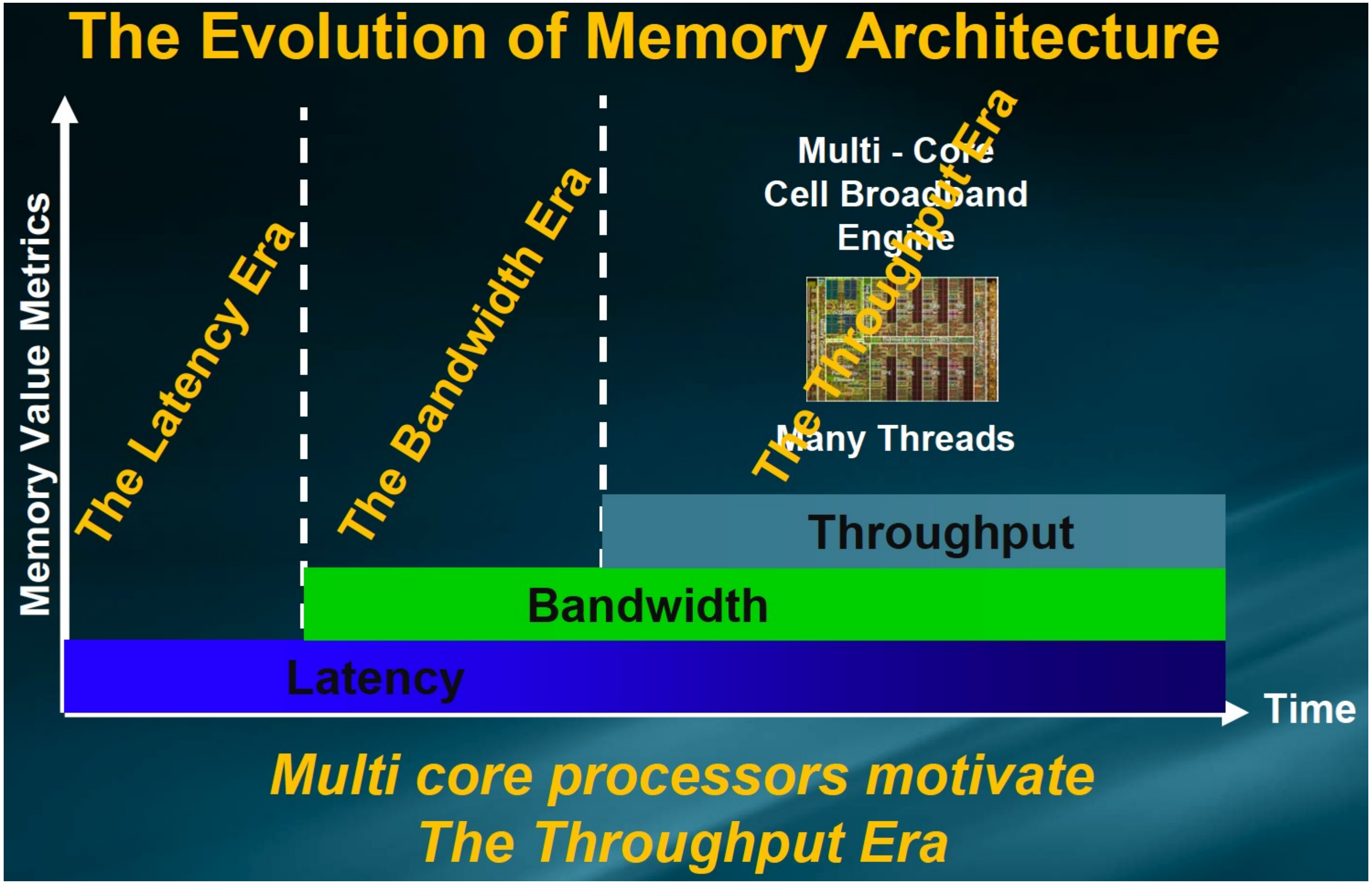


Overcoming the Memory Wall



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

Parallel Memory Accesses ➔ Throughput Computing

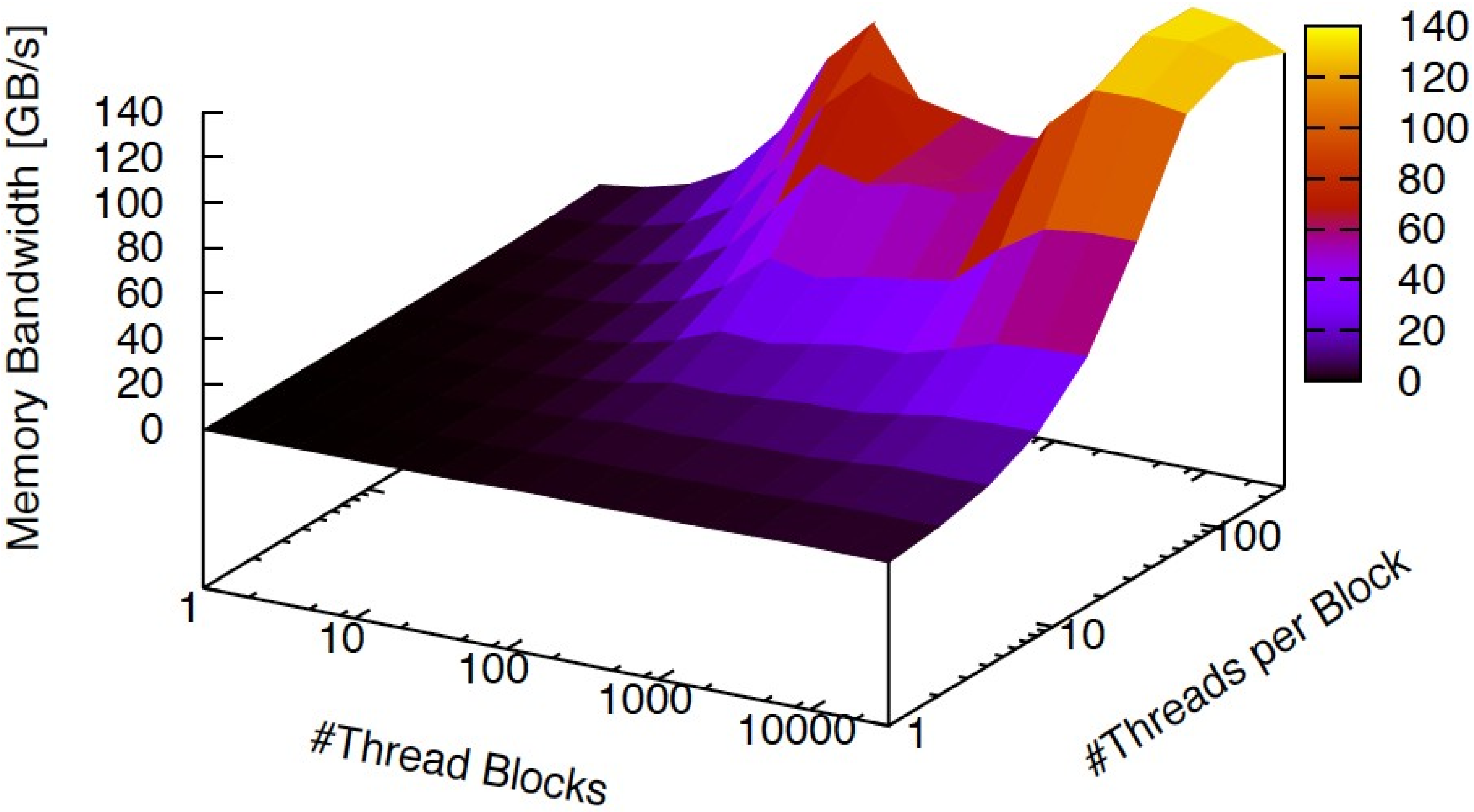


Source: Terabyte Bandwidth Initiative, Craig Hampel - Rambus, HotChips'08

GPUs as an example for highly parallel architectures

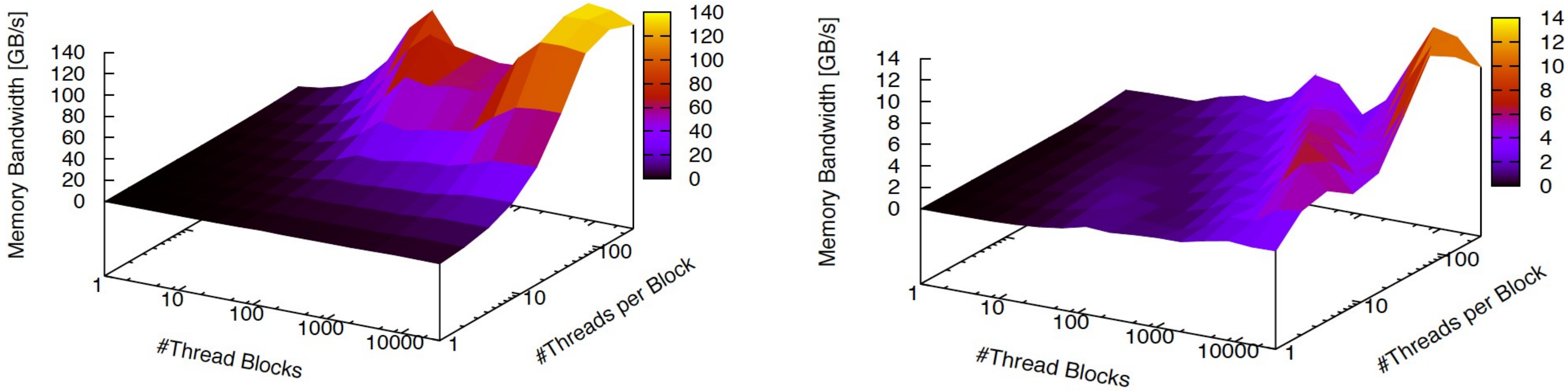
- Besides Teraflop(s) GPU's offer:
  - Massive Parallelism
  - 100+ GB/s memory bandwidth/throughput
  - Better performance per watt and per sqft.

GPU Memory bandwidth – ideal access pattern



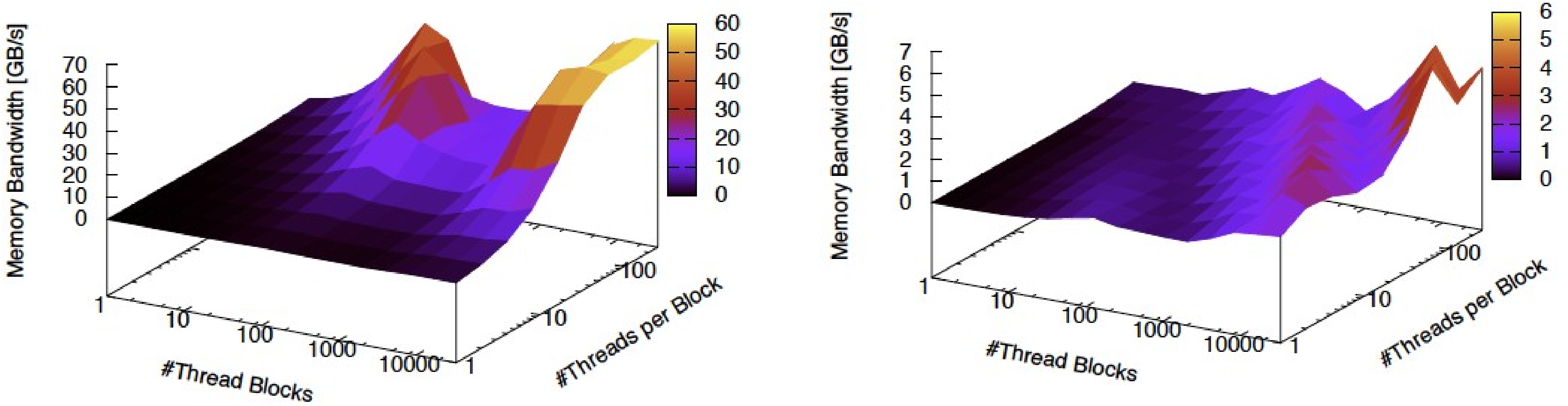
Bandwidth of sequential (coalesced) 32-bit read access for multiple thread configurations.
Results for a nVidia GTX 285 1.5GHz, GDDR3 1.2GHZ.

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) ?
- 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

Conventional Search Algorithms are suboptimal

- “It's the memory stupid!”
  - Binary search means random access =(
  - B-tree search is (partially) sequential
    but not amenable to coalescing

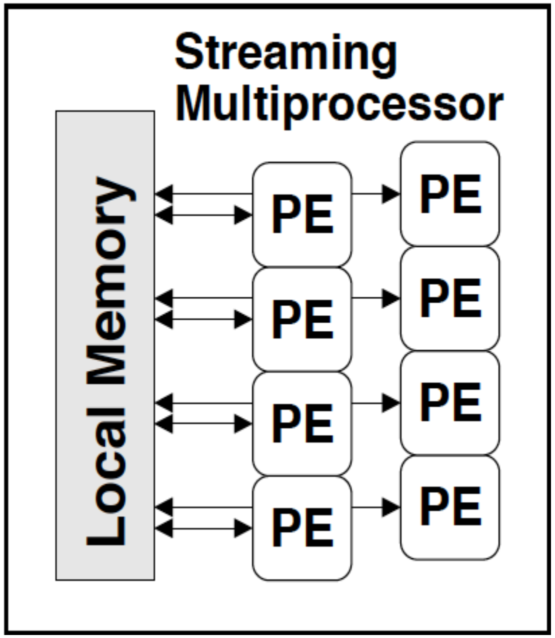
Conventional Search Algorithms are suboptimal

- “It's the memory stupid!”
  - Binary search means random access =(
  - B-tree search is (partially) sequential

    but not amenable to coalescing
- The CPU thread model “1 thread = 1 query” does not map well to the GPU as threads diverge
  - Produces random memory access pattern
  - It's a SIMD machine:
    The larger the # threads the more likely it will take WCET to complete

GPU architecture reminder – SIMD/SIMT

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



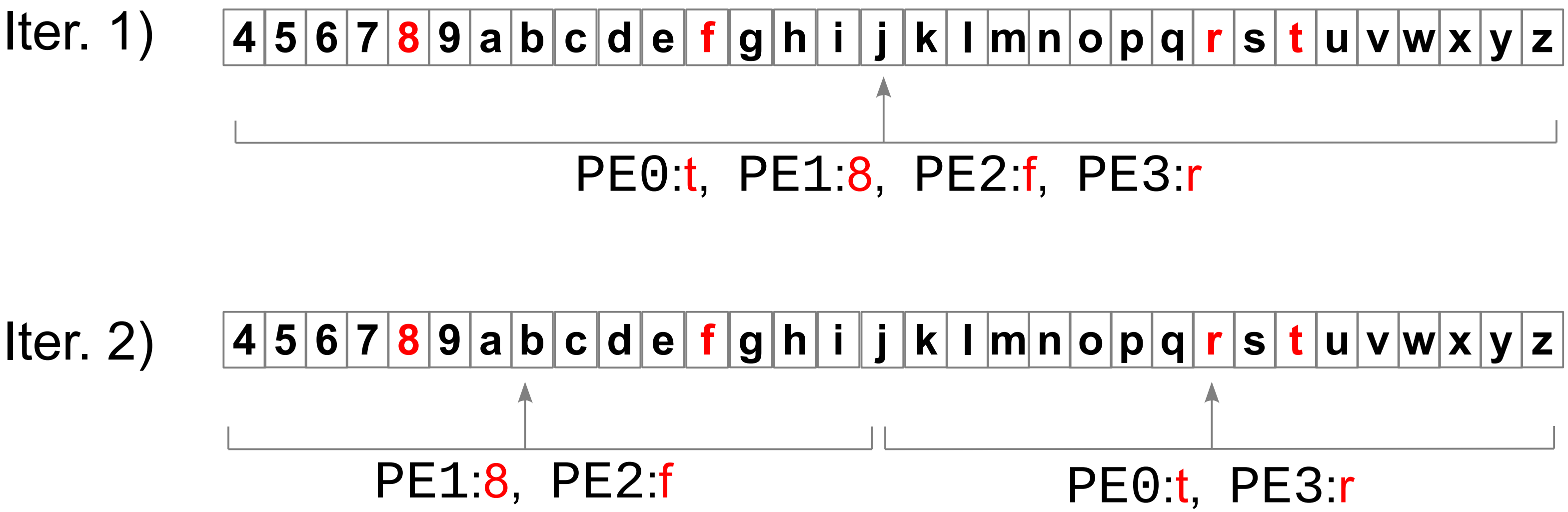
Multi-threaded Binary Search – Example

- 1 Index: a sorted char array 32 entries
- 4 queries: t , 8 , f , r
- 4 processors: PE 1-4
- 1 PE does 1 (binary) search: PE0:t , PE1:8 , PE2:f , PE3:r
- Theoretical worst-case execution time (wcet): $\log_2(32)=5$

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 processors: PE 1 - 4
- 1 PE does 1 (binary) search: PE0:t , PE1:8 , PE2:f , PE3:r
- Theoretical worst-case execution time (wcet): $\log_2(32)=5$



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

PE0:t, PE1:8, PE2:f, PE3:r

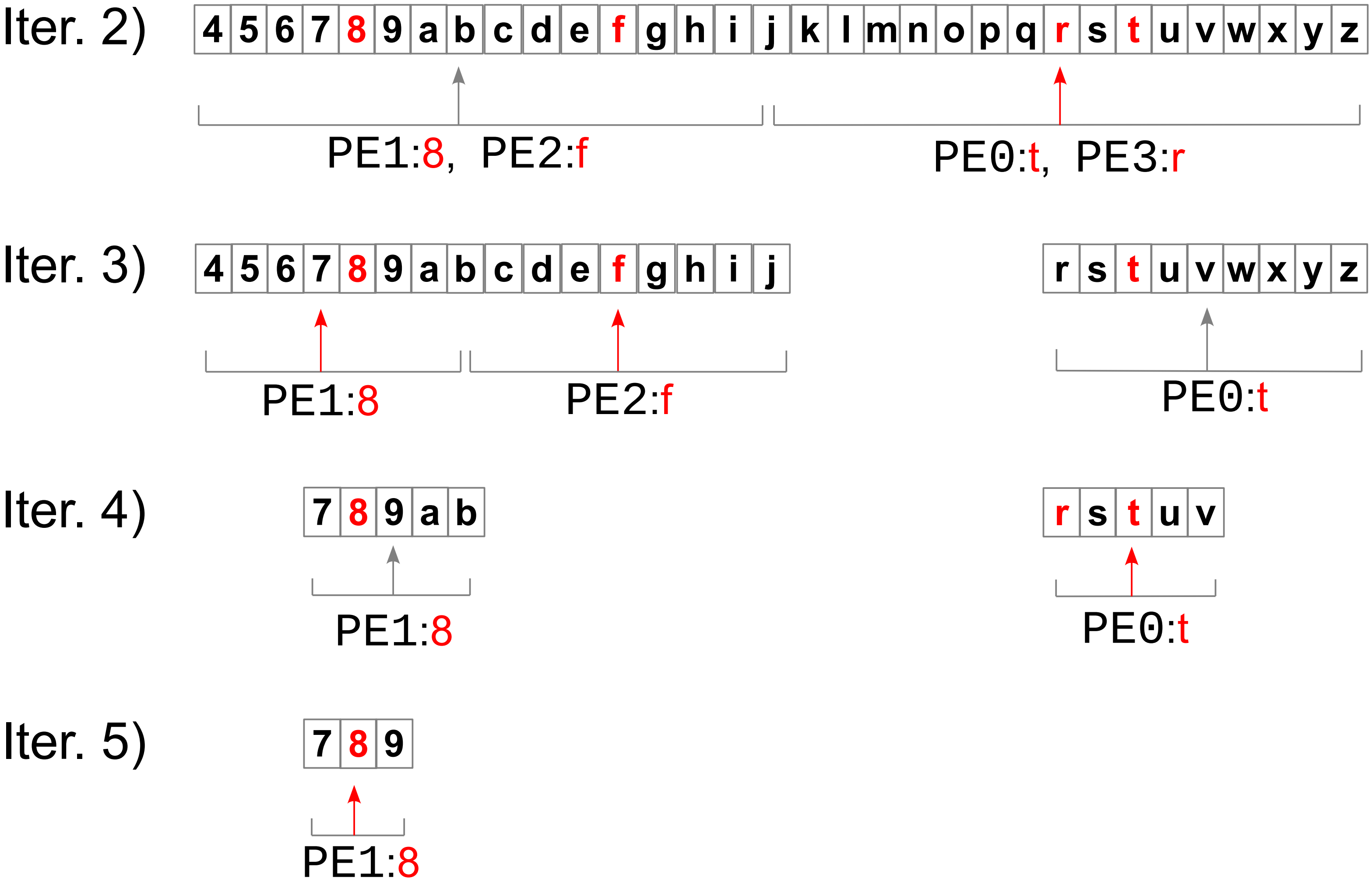
Iter. 2)

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

PE1:8, PE2:f

PE0:t, PE3:r

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

PE1:8, PE2:f

PE0:t, PE3:r

Iter. 3)

4 5 6 7 8 9 a b c d e f g h i j

PE1:8

PE2:f

r s t u v w x y z

PE0:t

Iter. 4)

7 8 9 a b

PE1:8

r s t u v

PE0:t

Iter. 5)

7 8 9

PE1:8

Conventional multi-threading – Analysis

- 100% utilization requires #PEs 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, wcet = $\log_2(n)$

How about improving wcet (latency)?

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

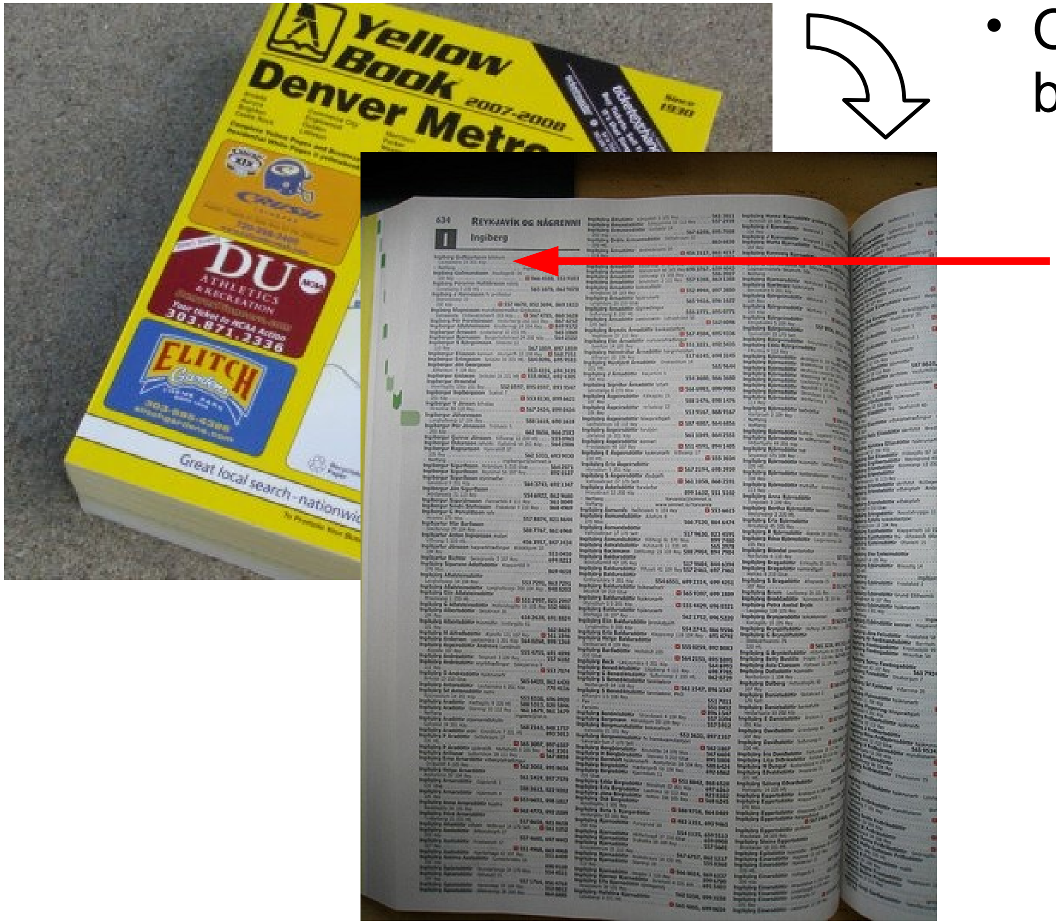
- Improve response time (latency) of core database functions like search in the era of throughput oriented (parallel) computing.

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

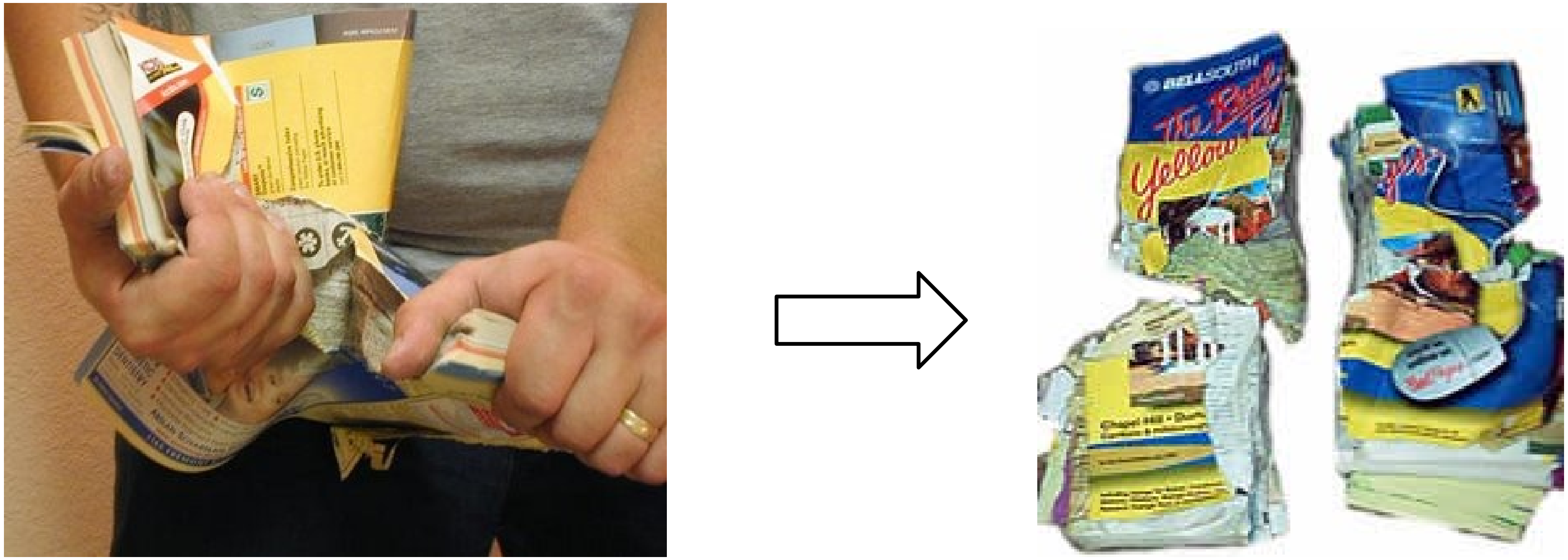
- How Do you (efficiently) search an index?



- 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

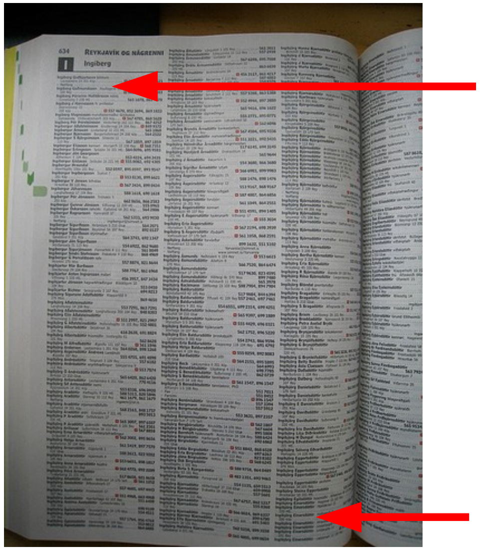
- Divide et impera !!

- 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 ?

II II

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 ?

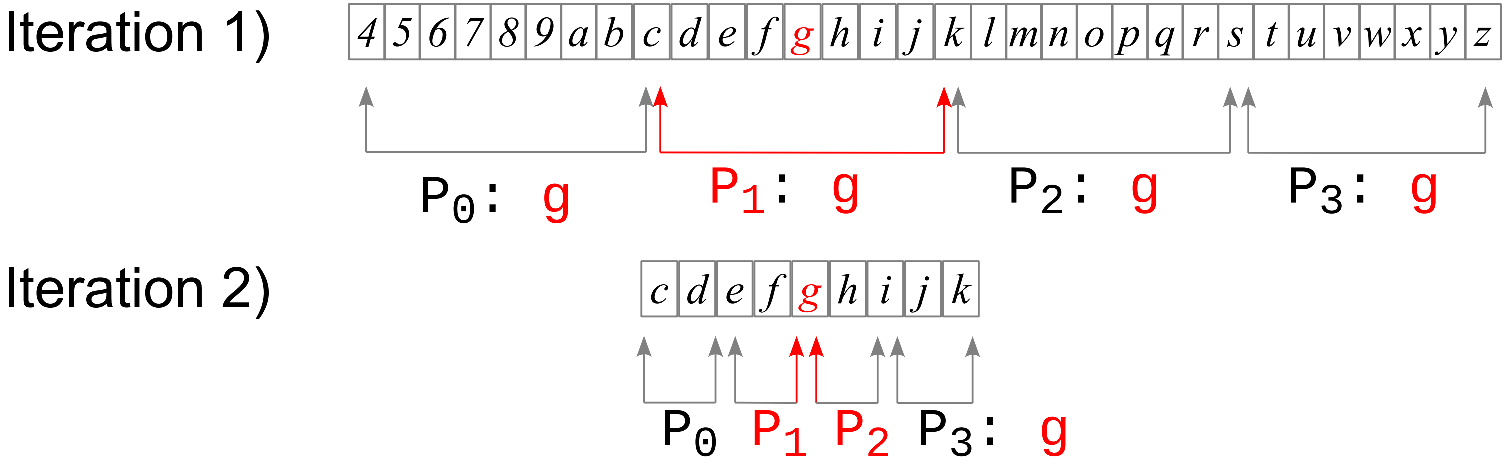
||　　　||

memory access　　　synchronization

- Searching a database index can be implemented the same way
  - Friends = Processors (Threads)
  - Without destroying anything ;-)

P-ary Search - Implementation

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

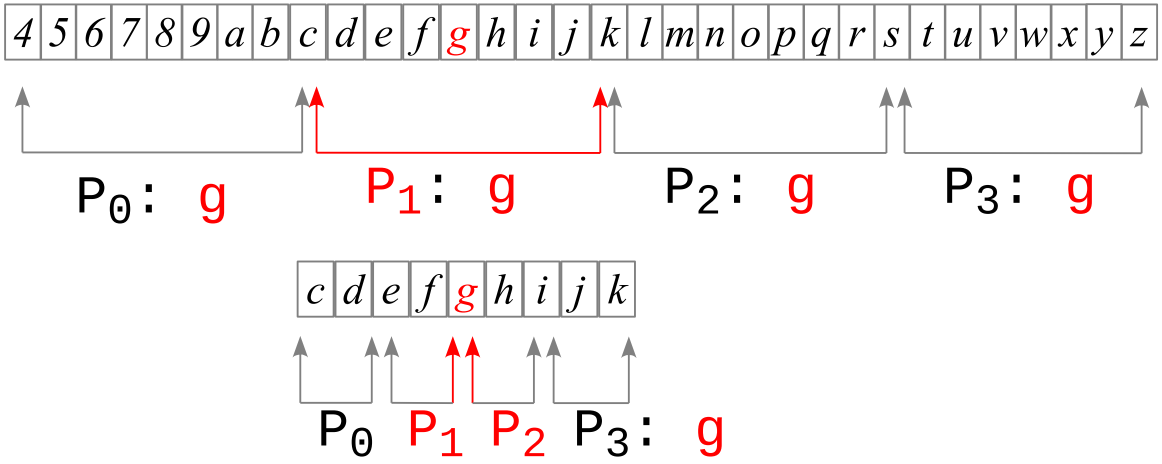


P-ary Search - Implementation

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

Iteration 1)

Iteration 2)



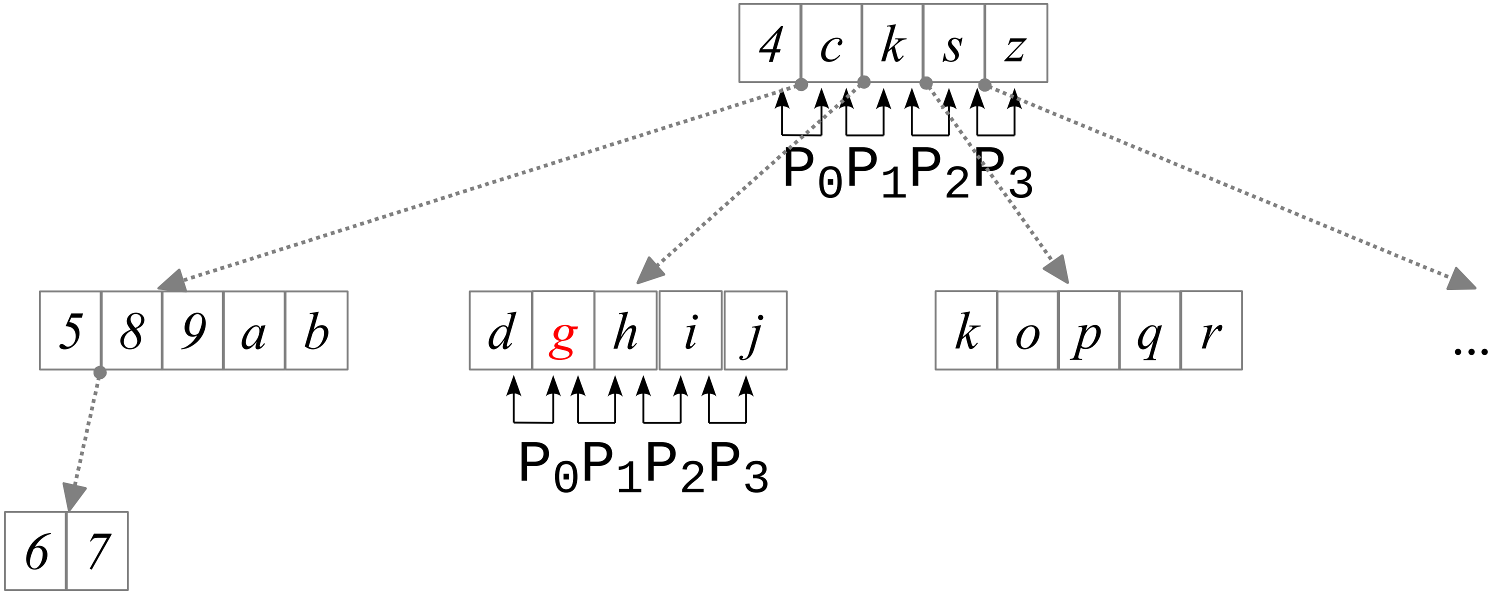
  - Synchronization ~ repartition cost

    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

- Performance depends on data structure
  - B-trees group pivot elements



  - Linear memory accesses are fast
  - 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)

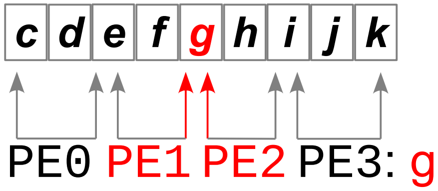
```
    // repeat until found
    while (range_length>BLOCKSIZE){
        // range voodo w/o floats
        range_length = range_length/BLOCKSIZE;
        if (range_length * BLOCKSIZE < old_range_length)
            range_length+=1;
        old_range_length=range_length;

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

P-ary Search – Analysis

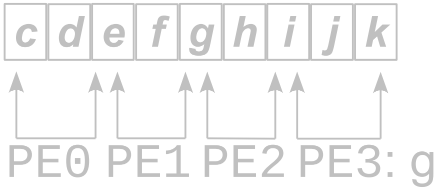
- 100% processor utilization for each query
- Multiple PEs can find a result
  - Does not change correctness

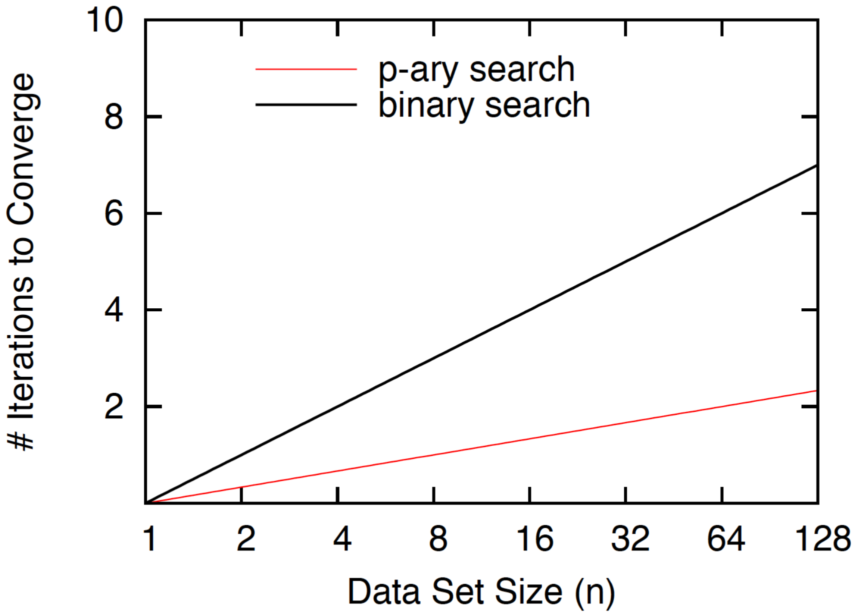


P-ary Search – Analysis

- 100% processor utilization for each query
- Multiple PEs can find a result
  - Does not change correctness
- Convergence depends on #PEs

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



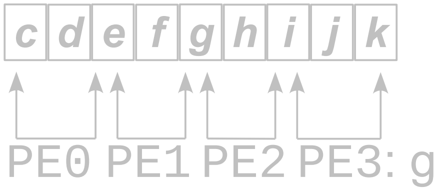


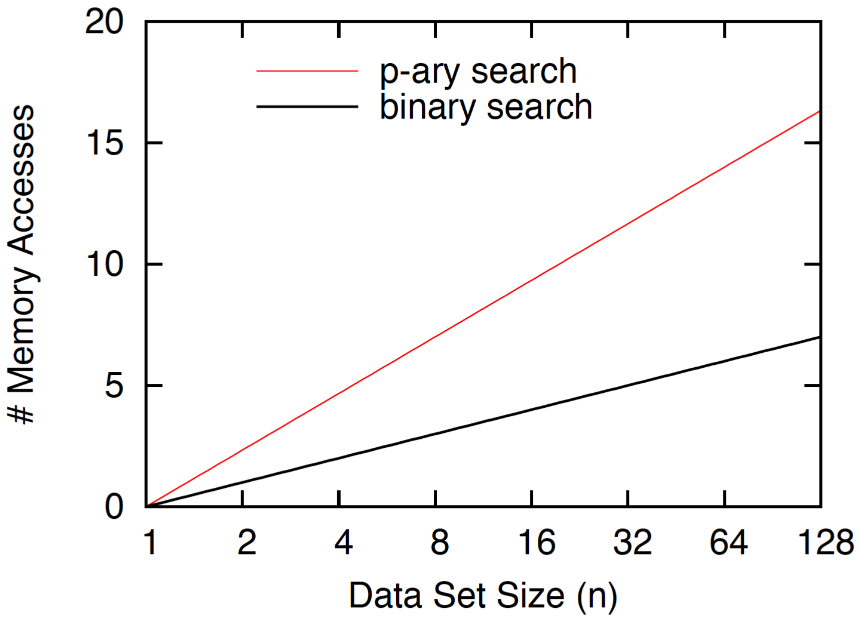
P-ary Search – Analysis

- 100% processor utilization for each query
- Multiple PEs can find a result
  - Does not change correctness
- Convergence depends on #PEs

  GTX285: 1 SM, 8 PEs → 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. $\log_2(n)$



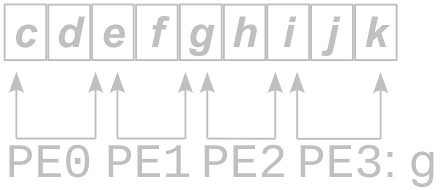


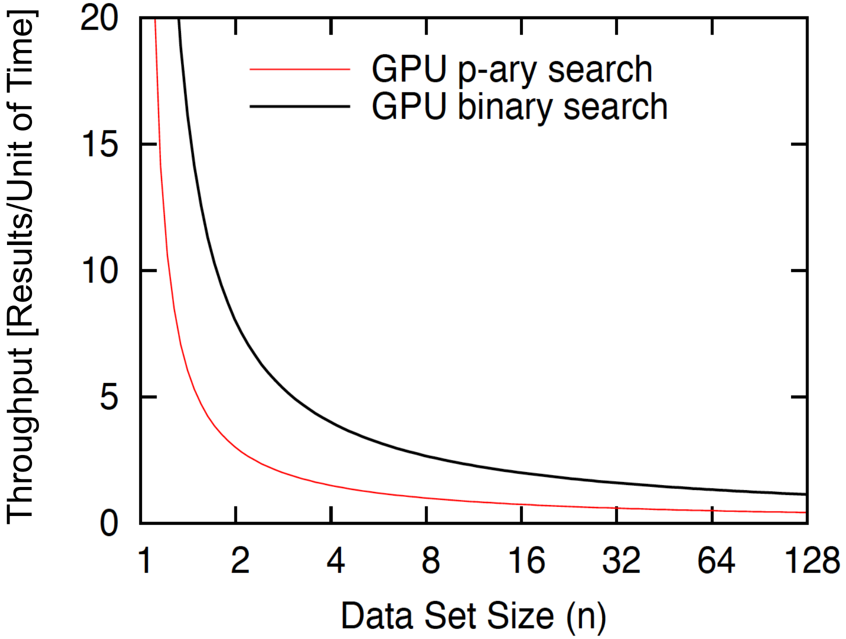
P-ary Search – Analysis

- 100% processor utilization for each query
- Multiple PEs can find a result
  - Does not change correctness
- Convergence depends on #PEs

  GTX285: 1 SM, 8 PEs → p=8
- Better Response time
  - $\log_p(n)$ vs $\log_2(n)$
- More memory access
  - $p*2$ per iteration * $\log_p(n)$
  - Caching

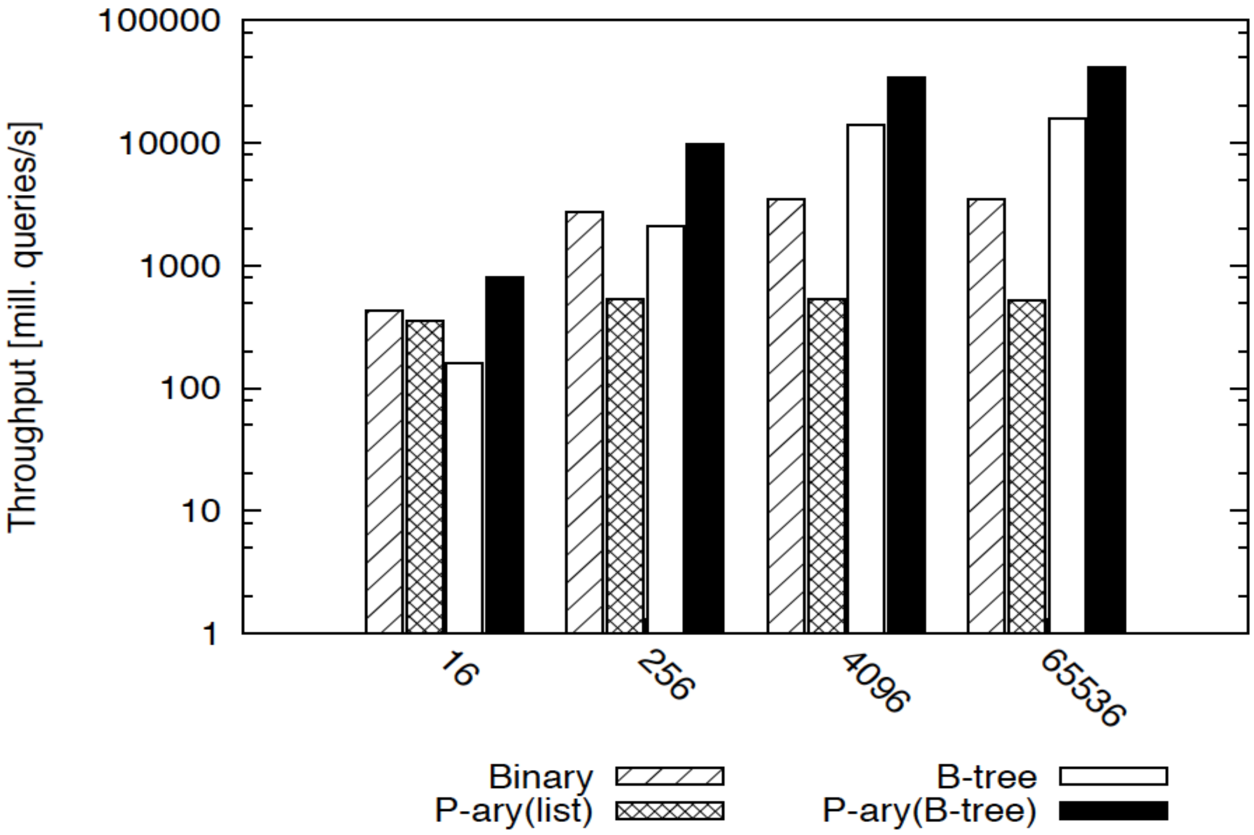
    $(p-1) * \log_p(n)$ vs. $\log_2(n)$
- Lower Throughput
  - $1/\log_p(n)$ vs $p/\log_2(n)$





P-ary Search (GPU) – Throughput

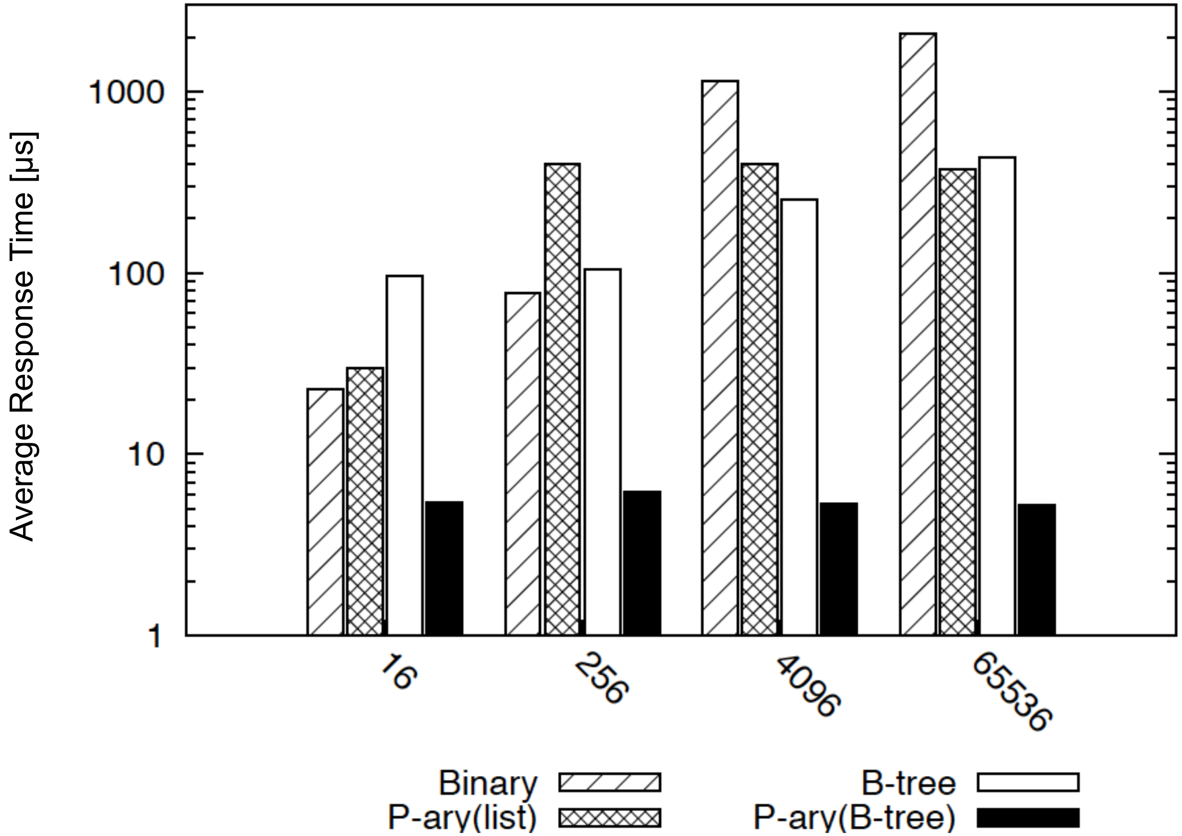
- Superior throughput compared to conventional algorithms



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

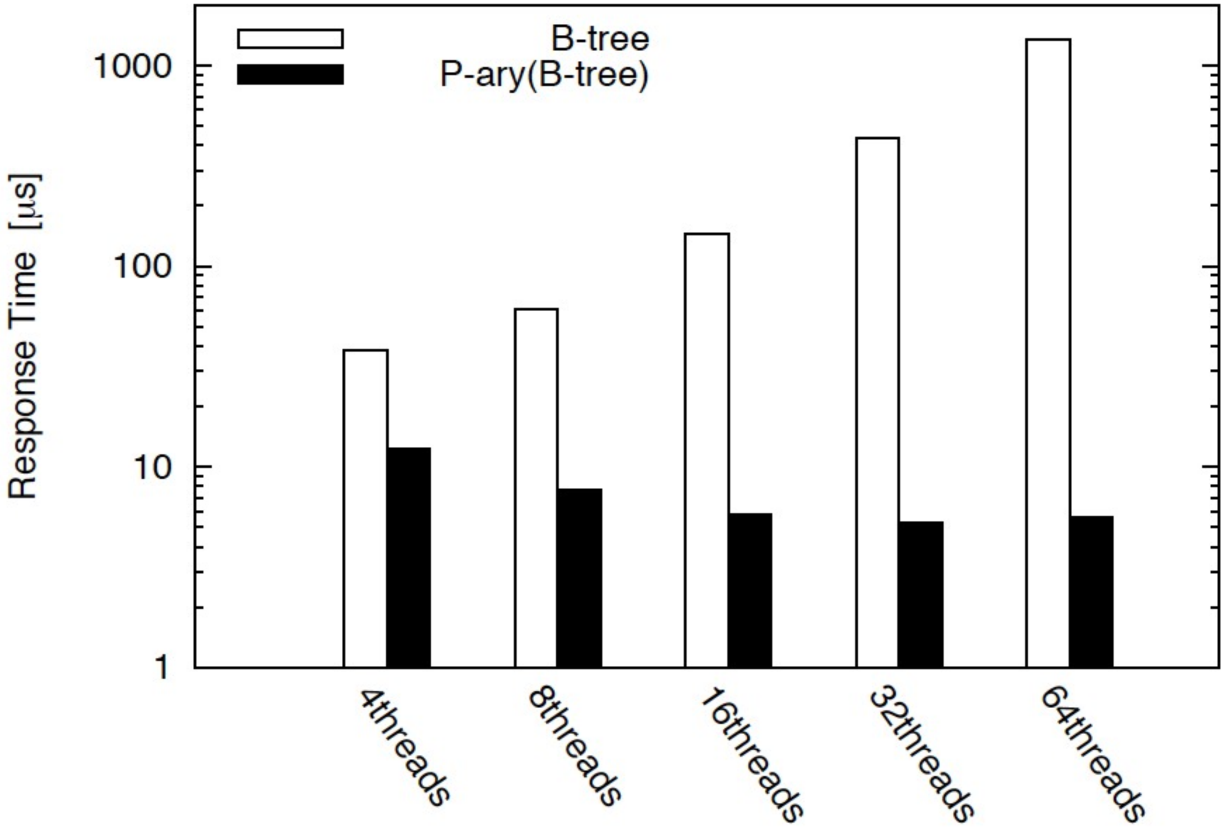
- Response time is workload independent



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

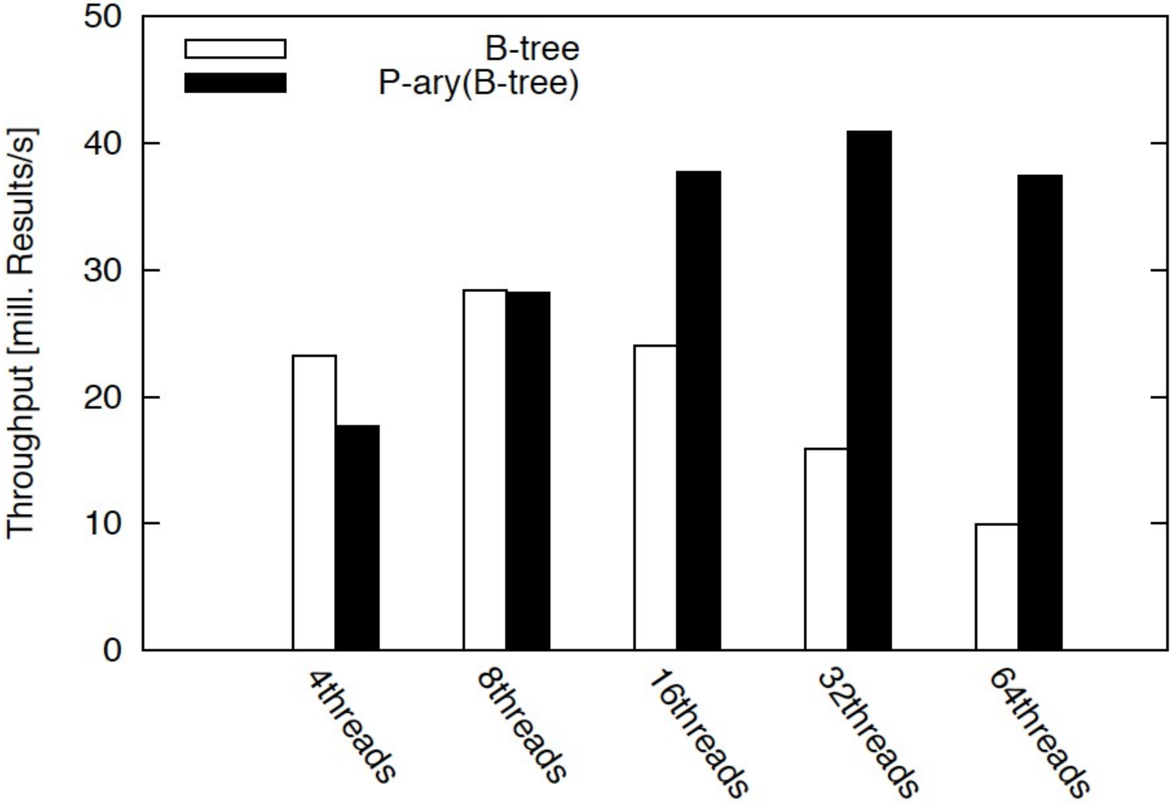
- GPU Implementation using SIMT (SIMD threads)
- Scalability with increasing #threads (P)



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

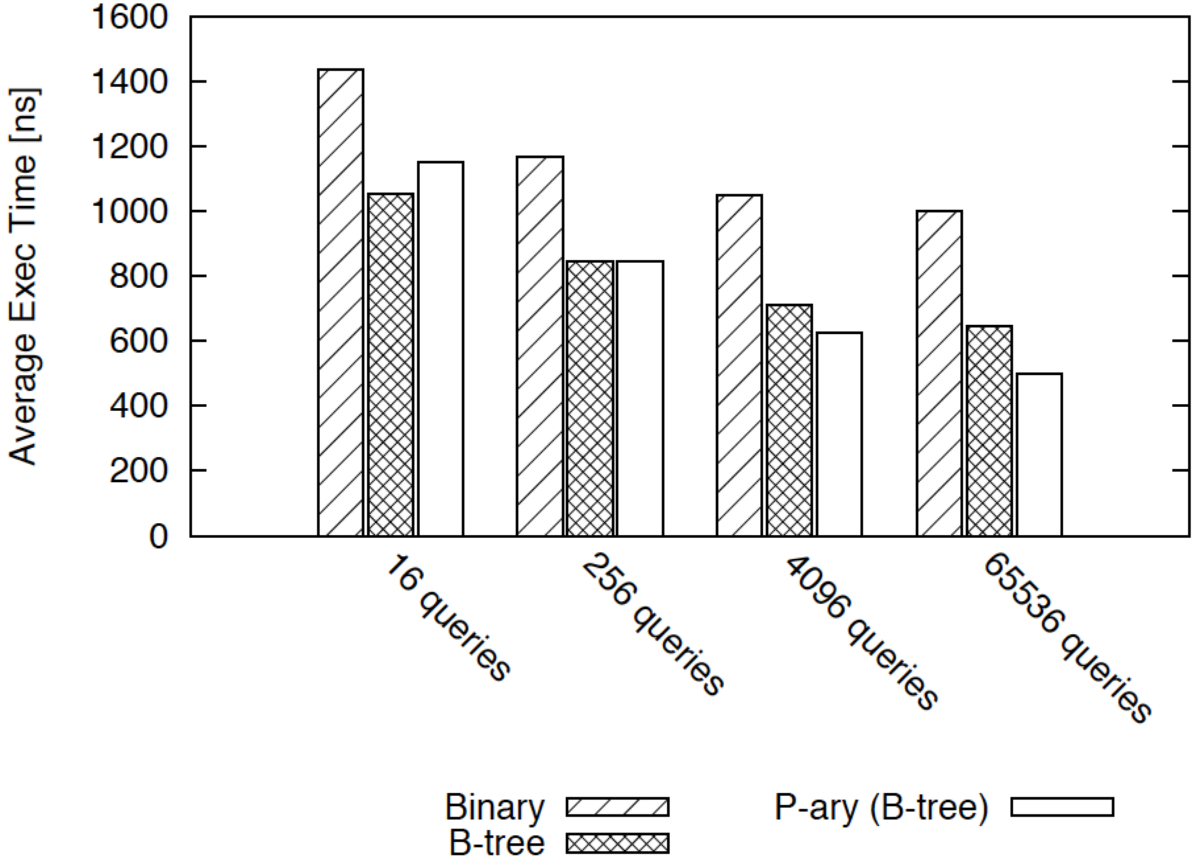
- GPU Implementation using SIMT (SIMD threads)
- Scalability with increasing #threads (P)



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 $\log_4(n)$



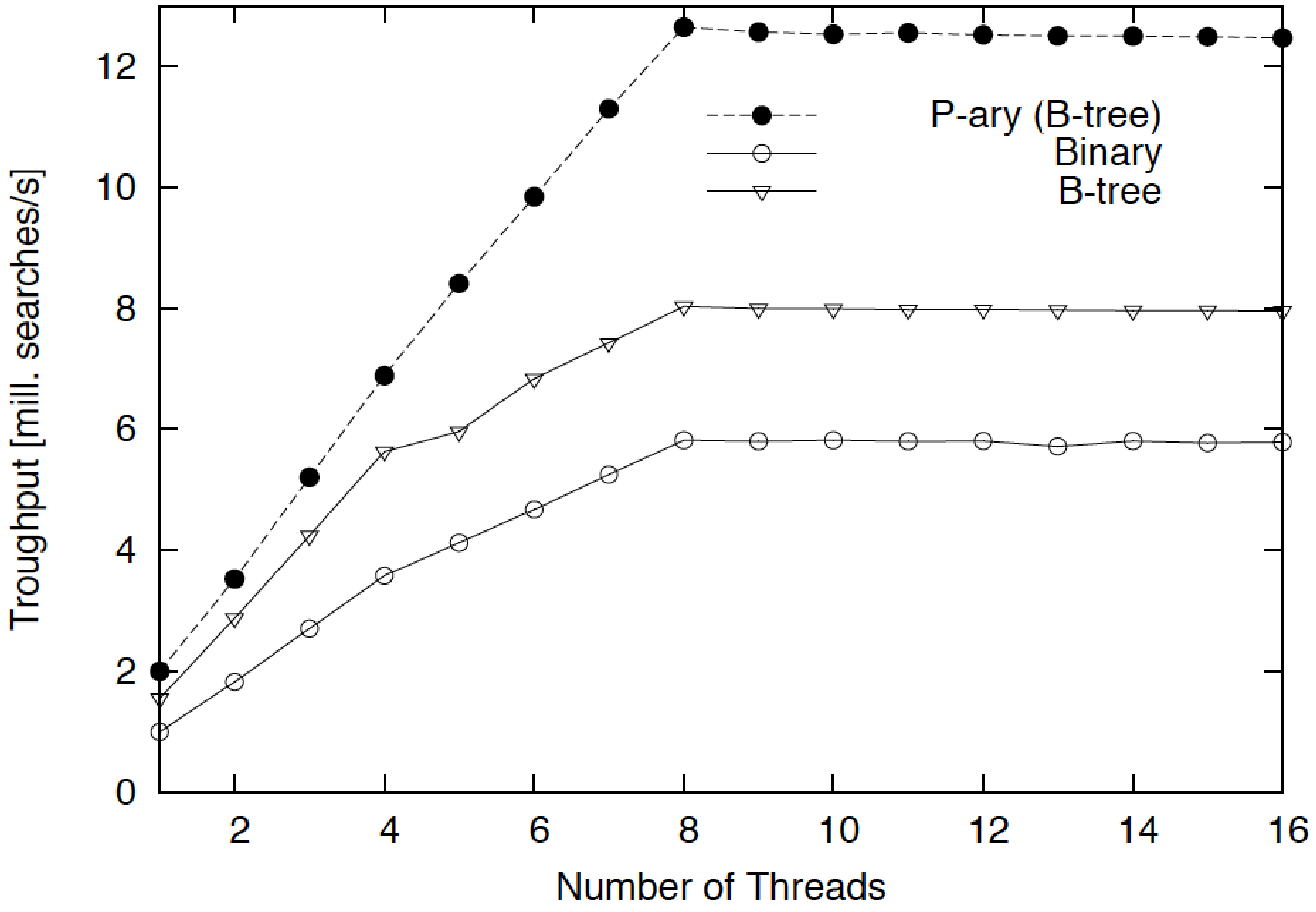
Searching a 512MB data set with 134mill. 4-byte integer entries, Core i7 2.66GHz, DDR3 1666.

---

[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