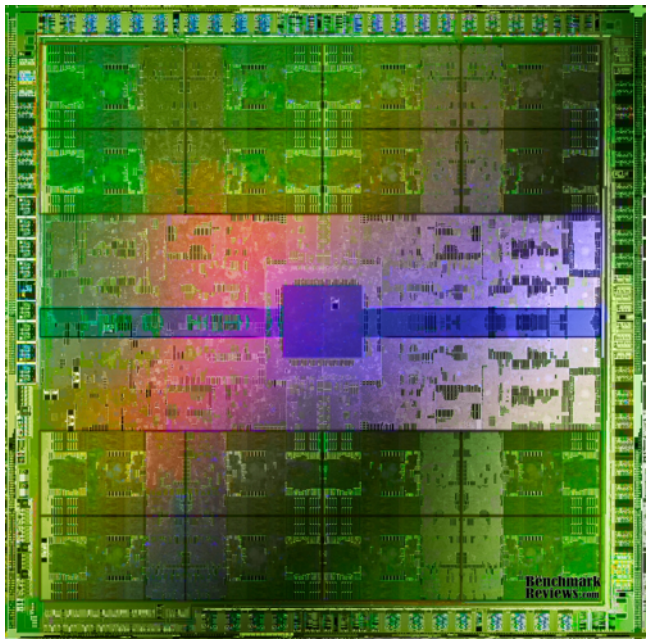
Large Scale Data Management on the GPU



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

Disclaimer

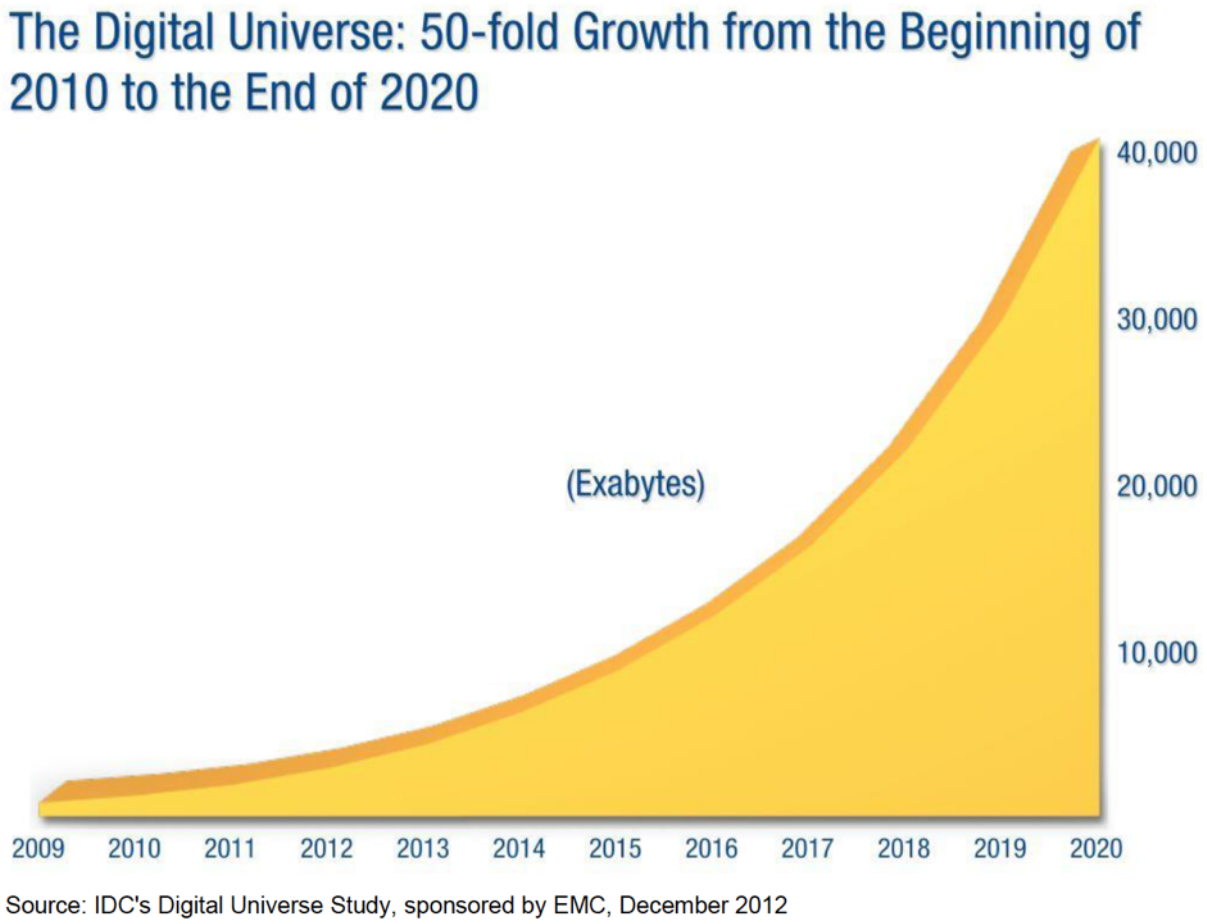
The author's views expressed in this presentation do not necessarily reflect the views of IBM.

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 this class many years in a row.

Managing Big (!) Data



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

- Supercomputing is tackling the Exaflop
- Data Management got past the Exabyte range alreadyc

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

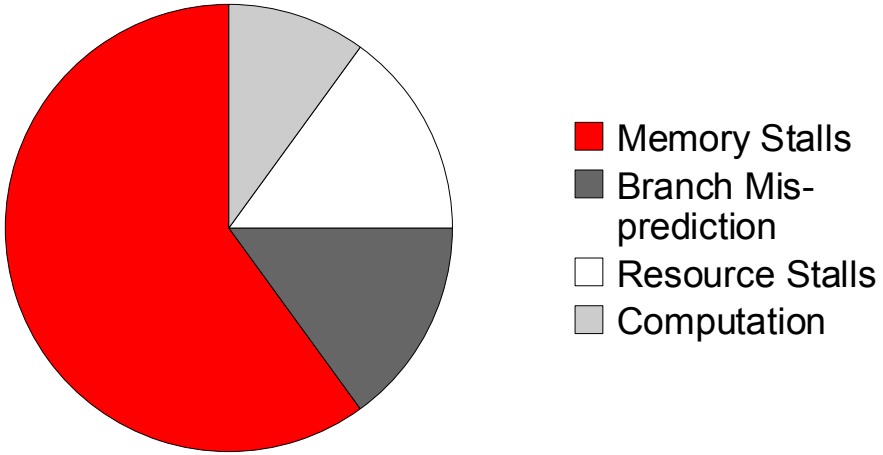
= DATA GROWTH

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

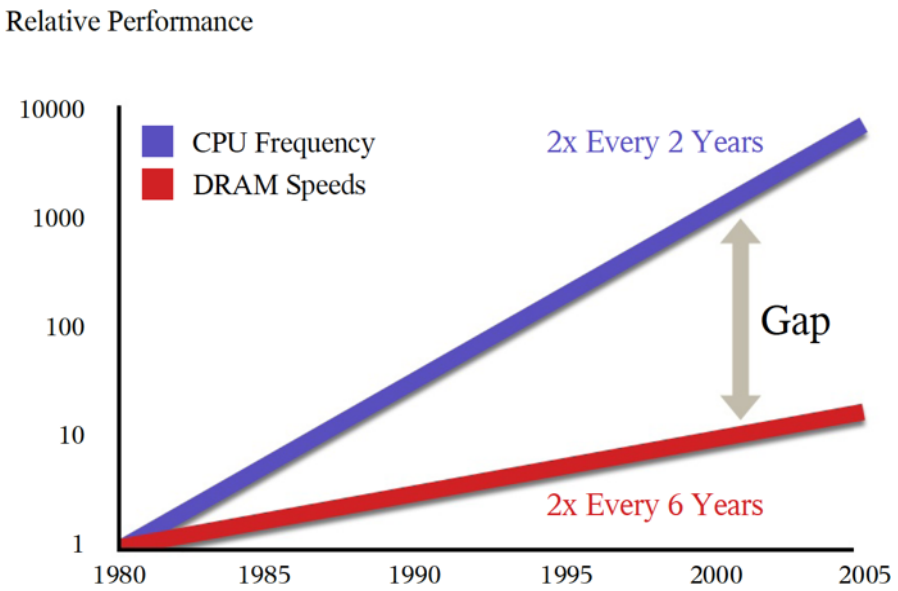
- How to efficiently analyze/process rapidly increasing volumes of (business data) ?
- What is the problem ?
  - Aren't processers getting faster every day ?

DB Performance – Where does time go ?

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

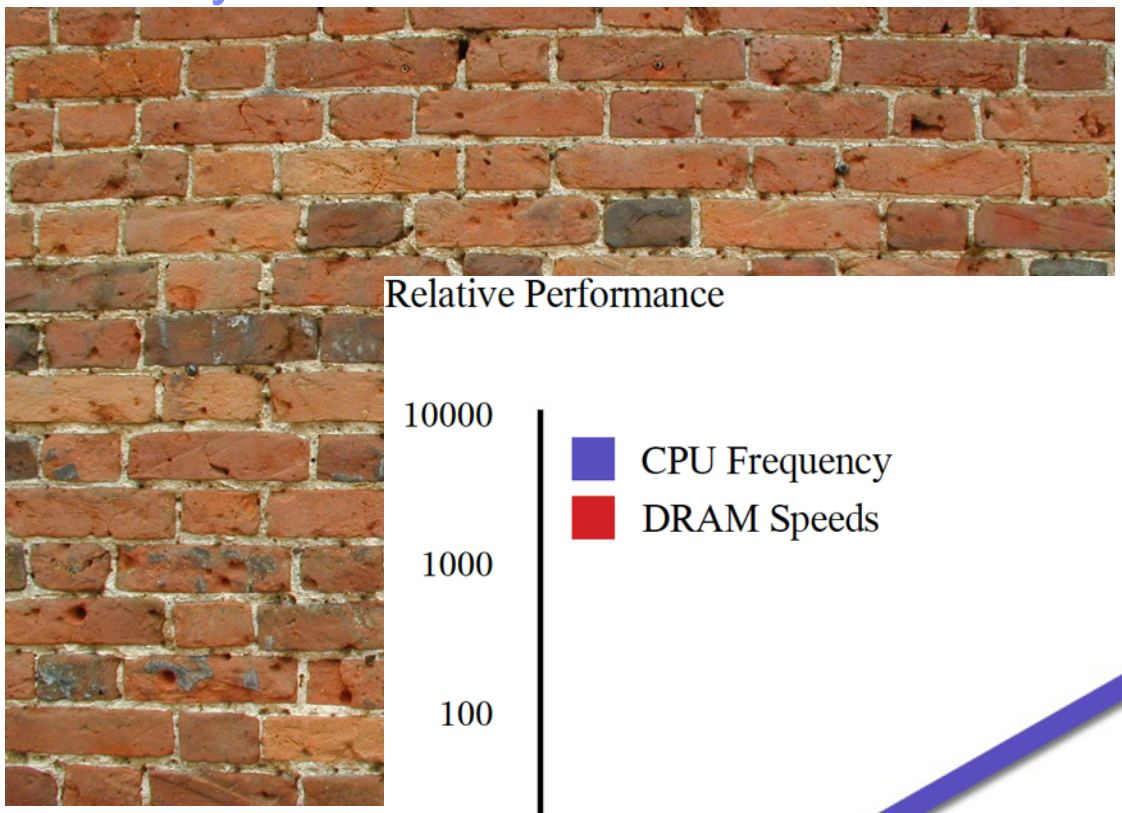


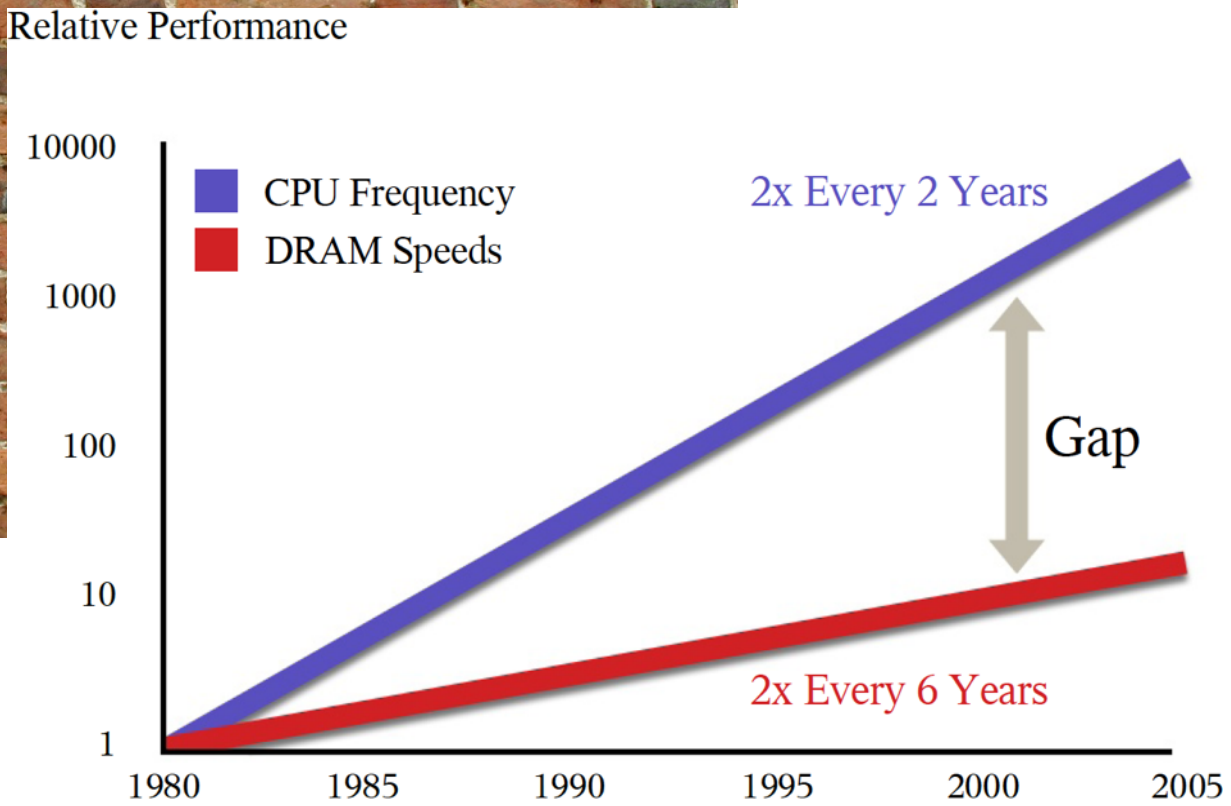
- It's getting worse [2]



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]



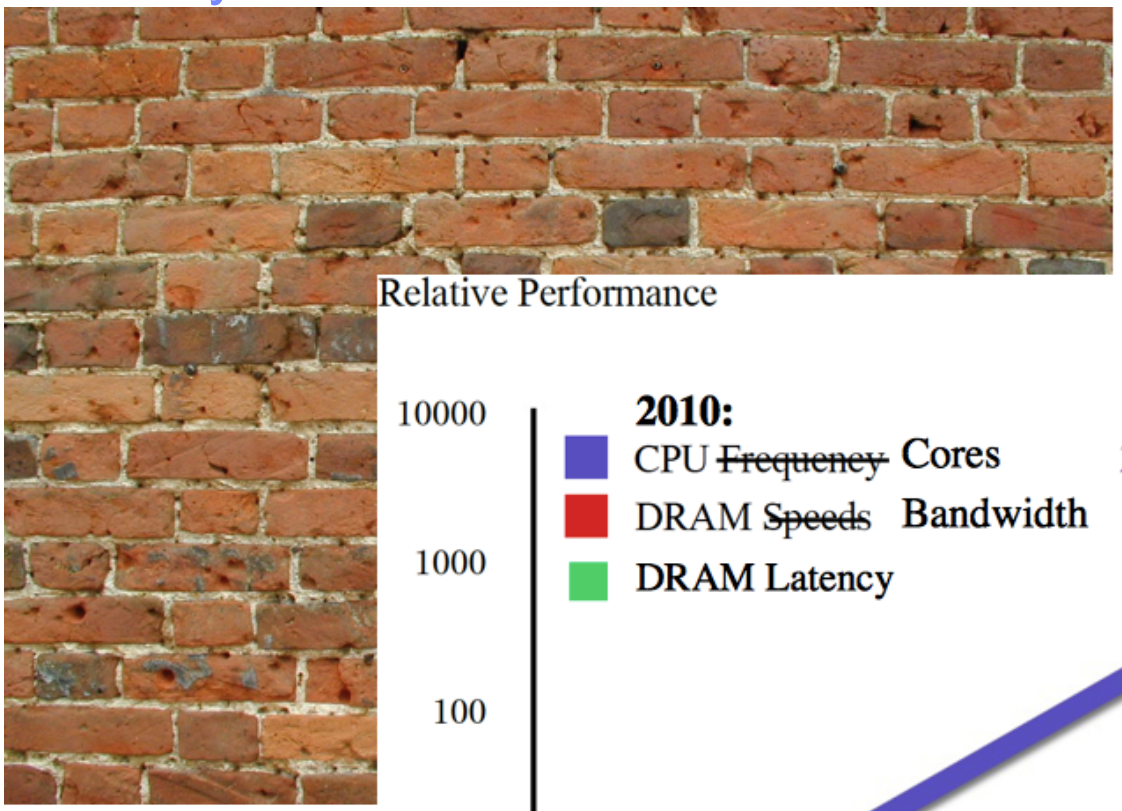


Sources:

[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]

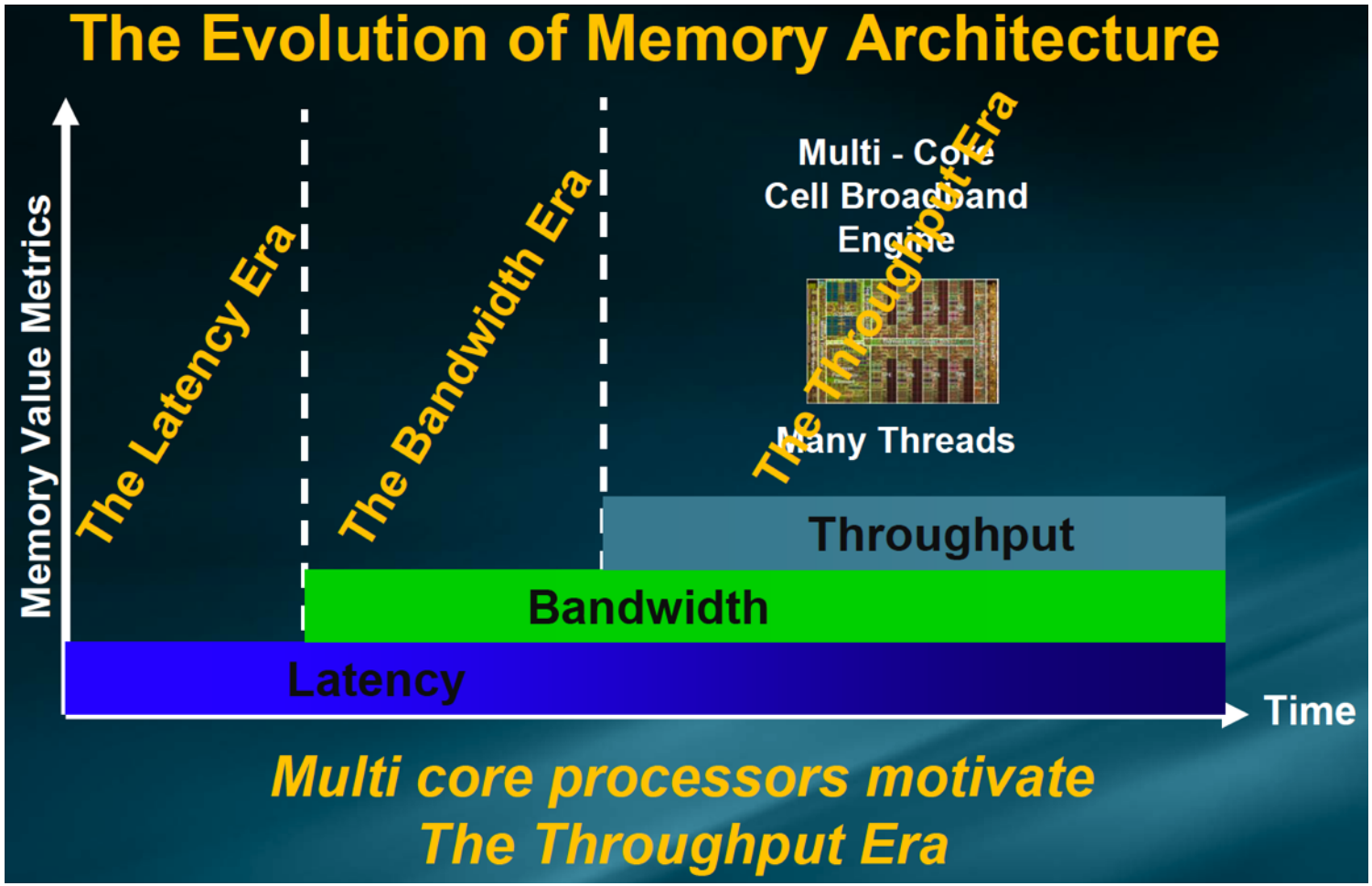


Sources:

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

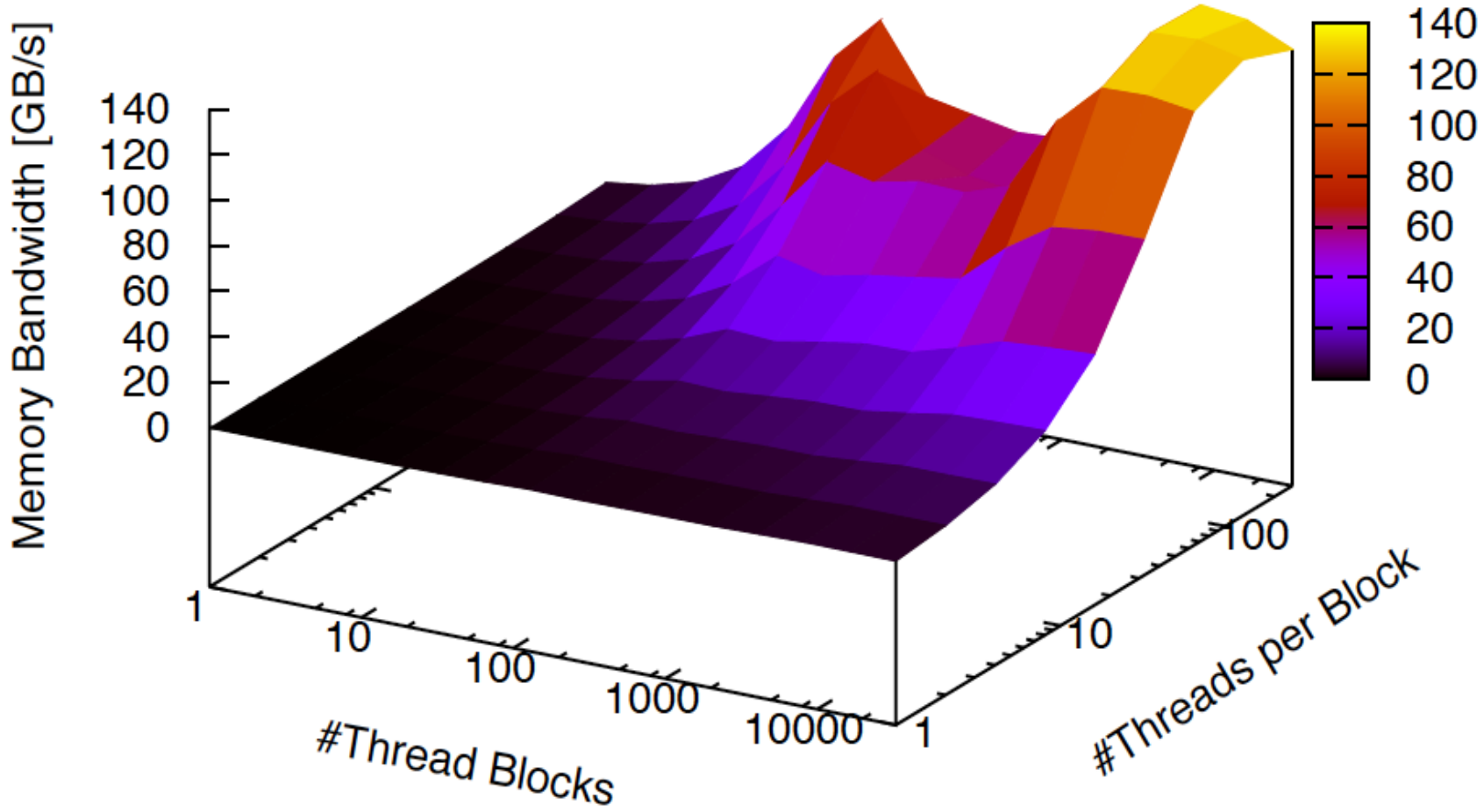
[2] David Yen. Opening Doors to the MultiCore Era. MultiCore Expo 2006

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

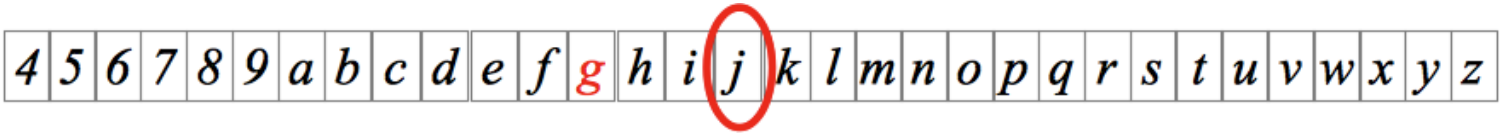
Why search?

- Honestly, how many times a day do you visit

?

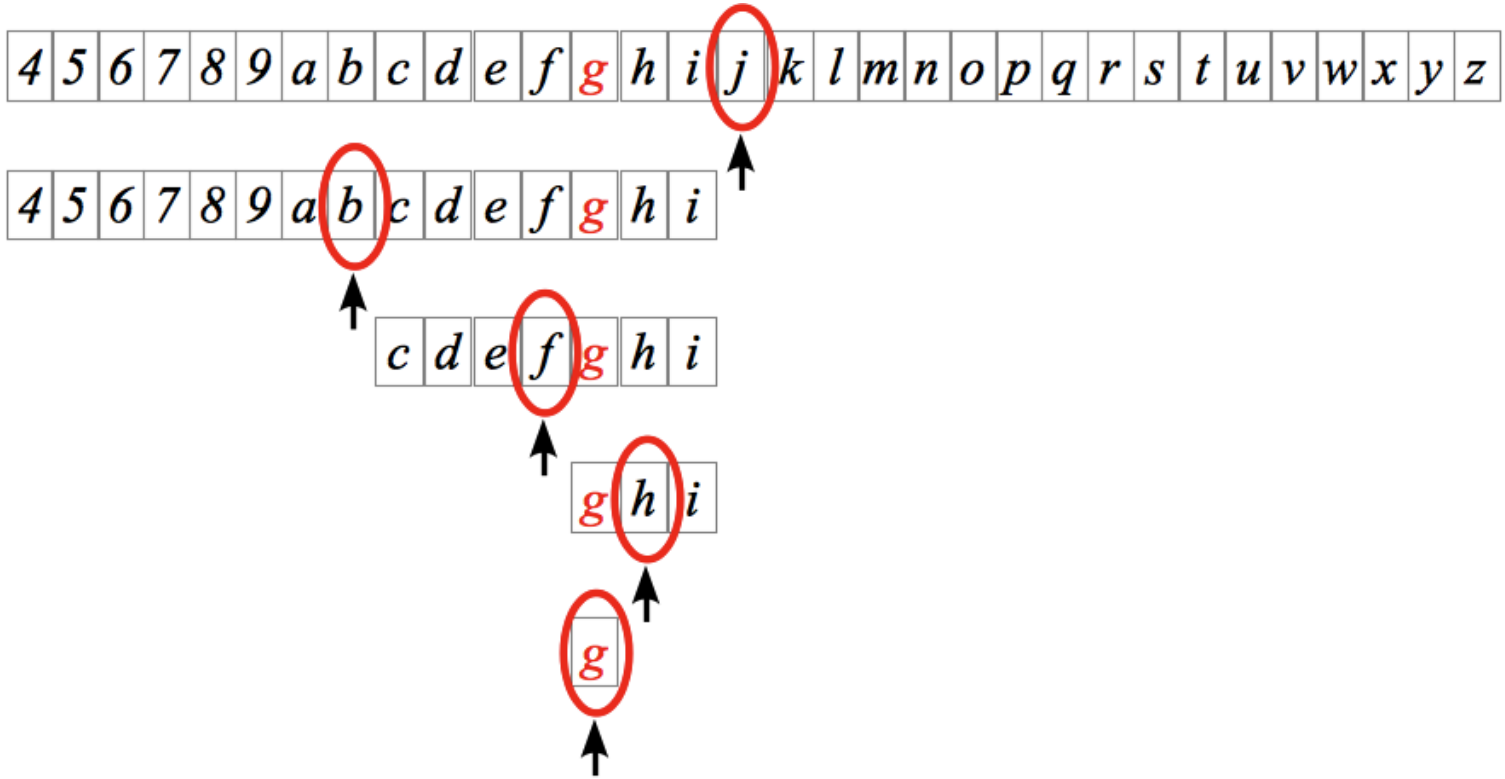
Search – Memory Access Pattern

- 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



Search – Memory Access Pattern

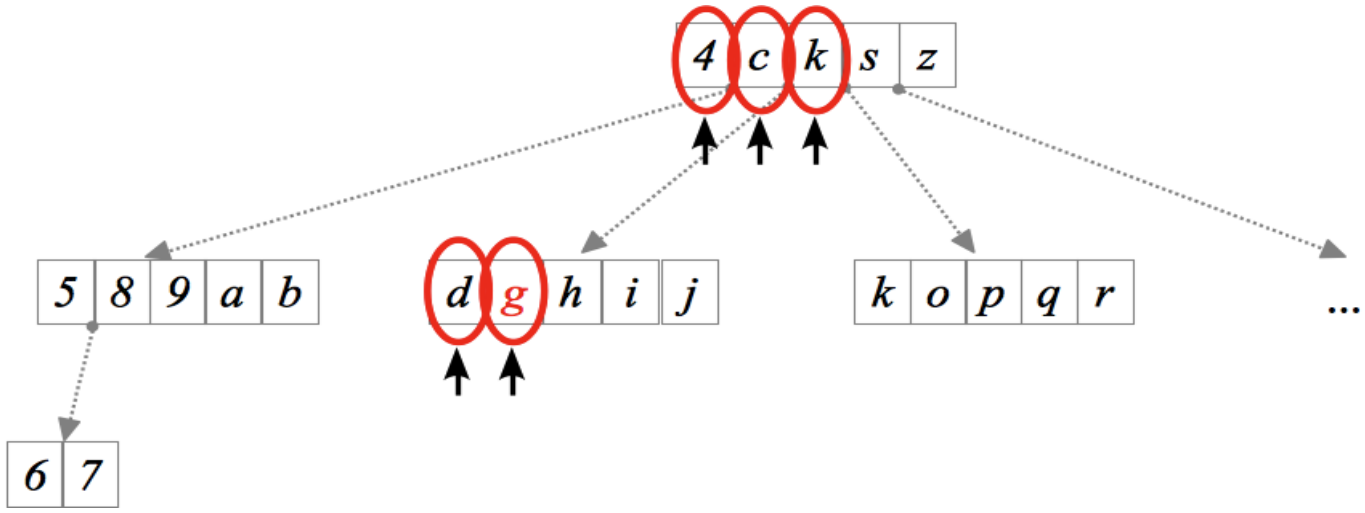
- Binary search (in a sorted list)
  - Choose Pivot Element at position *#elements / 2*
  - Compare if equal, larger, or smaller that search key (g)
    - Equal ➔ Done
    - Smaller ➔ proceed with lower half of the list
    - Larger ➔ proceed with upper half



- 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.

	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

How do you achieve this?

(1) Nvidia: 192.4 × 10^6 B/s ≈ 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

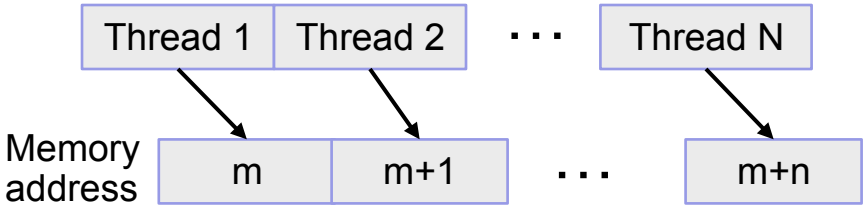
Agenda

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

Coalesced Memory access (reminder)

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

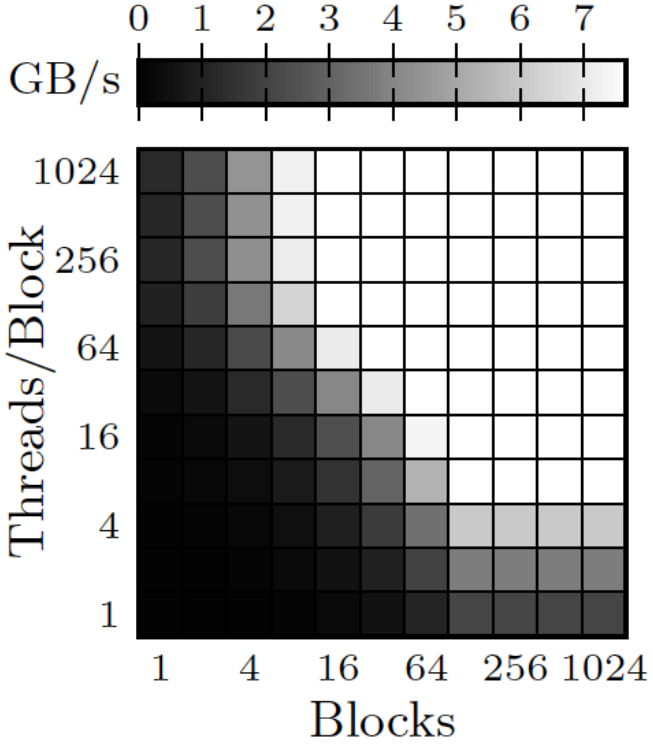



```
for (position = blockIdx.x*blockDim.x+threadIdx.x;
     position < length;
     position += blockDim.x*gridDim.x){
x = list[position];
...
```

- 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

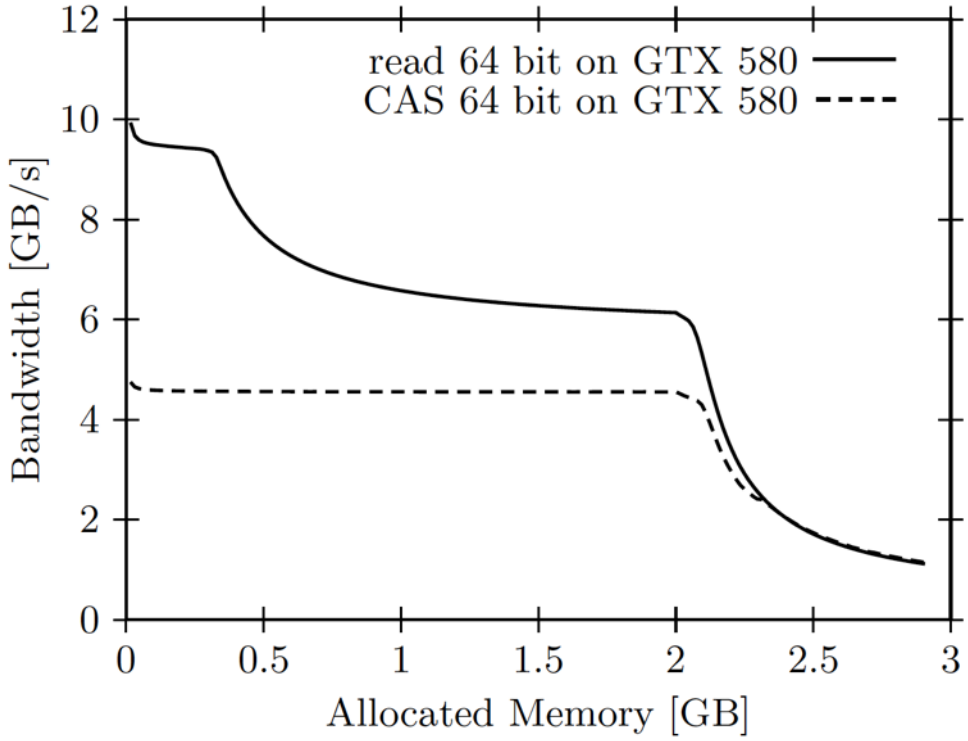


64-bit random access within 512 MB of device memory on GTX 580

- 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?

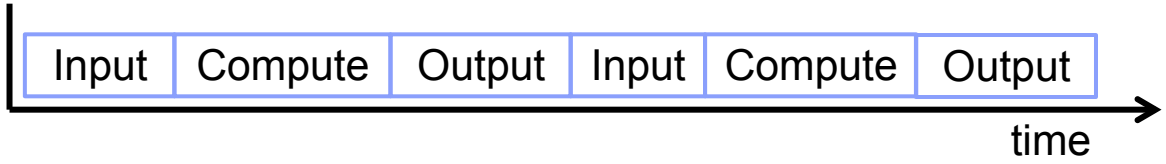
Agenda

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

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

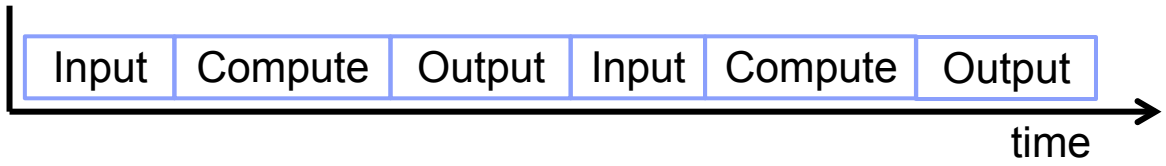
- Synchronous:



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

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

- Synchronous:

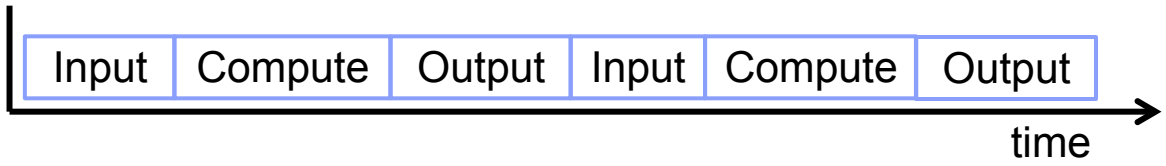


  - Copy data to GPU, compute, copy results back to CPU

```
// 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);
```

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

- Synchronous:



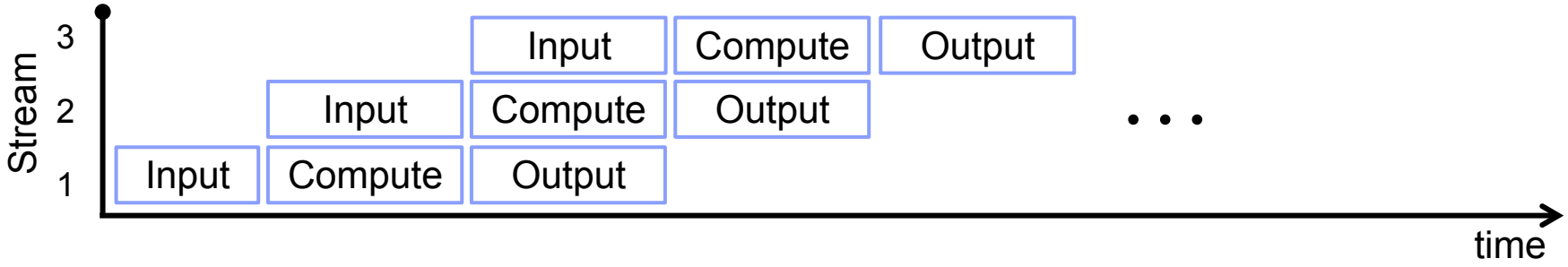
  - Copy data to GPU, compute, copy results back to CPU

```
// 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):

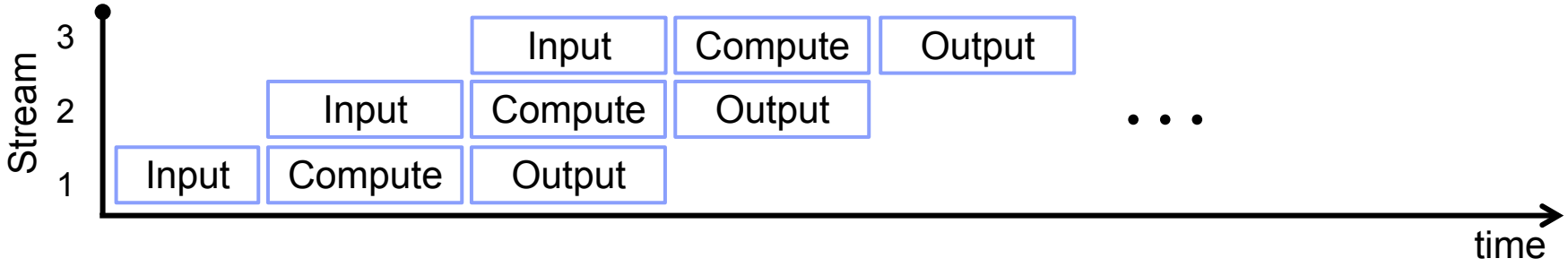
- Asynchronous data transfers, aka CUDA Streams:



  - 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:

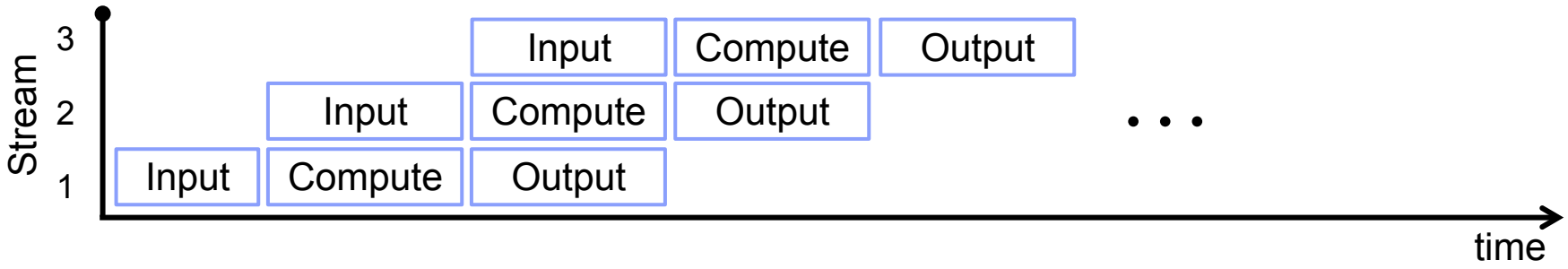


  - While one stream is executing a kernel, others can copy data in (and out)
    - Effectively overlap data copies with compute

```
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:



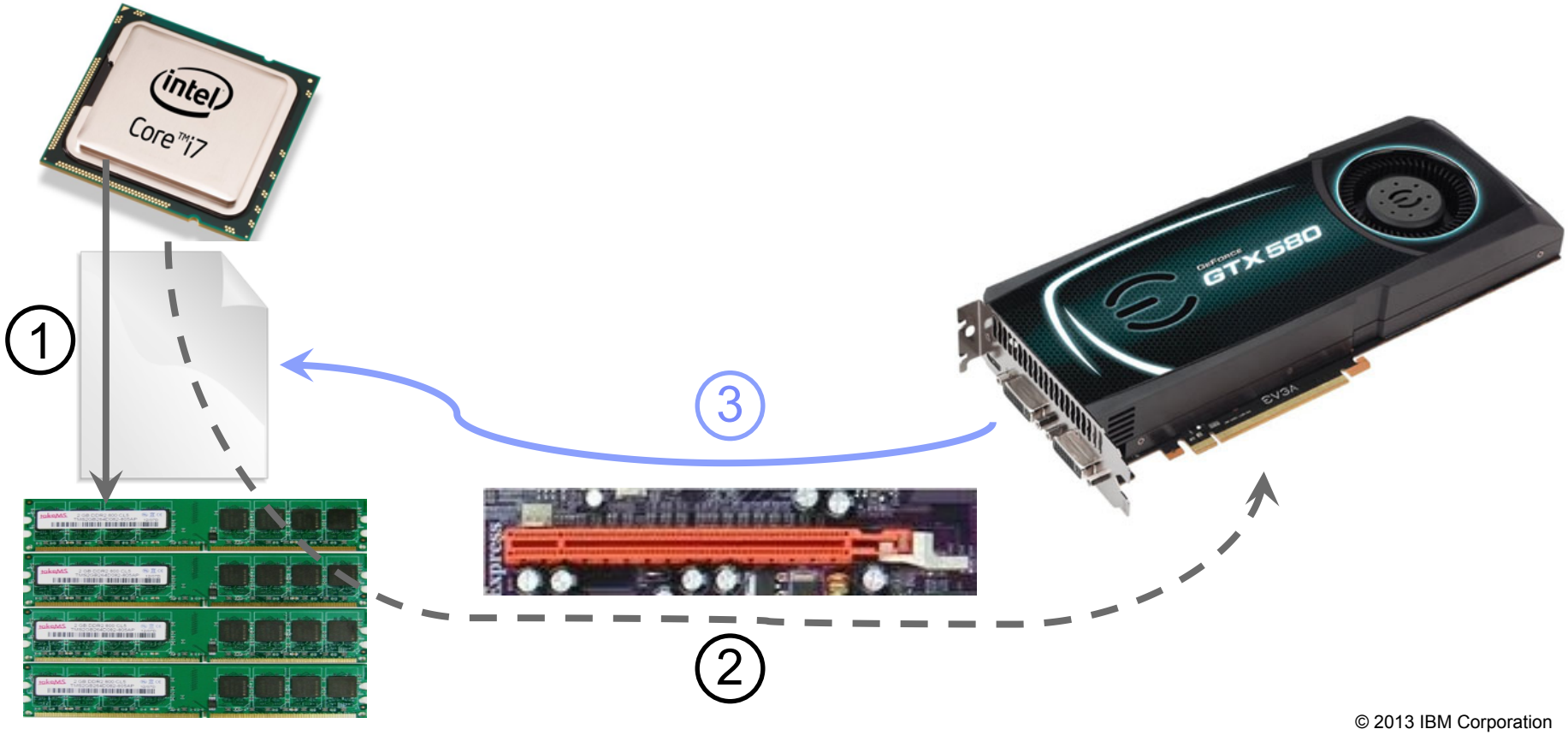
  - While one stream is executing a kernel, others can copy data in (and out)
    - Effectively overlap data copies with compute

```
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

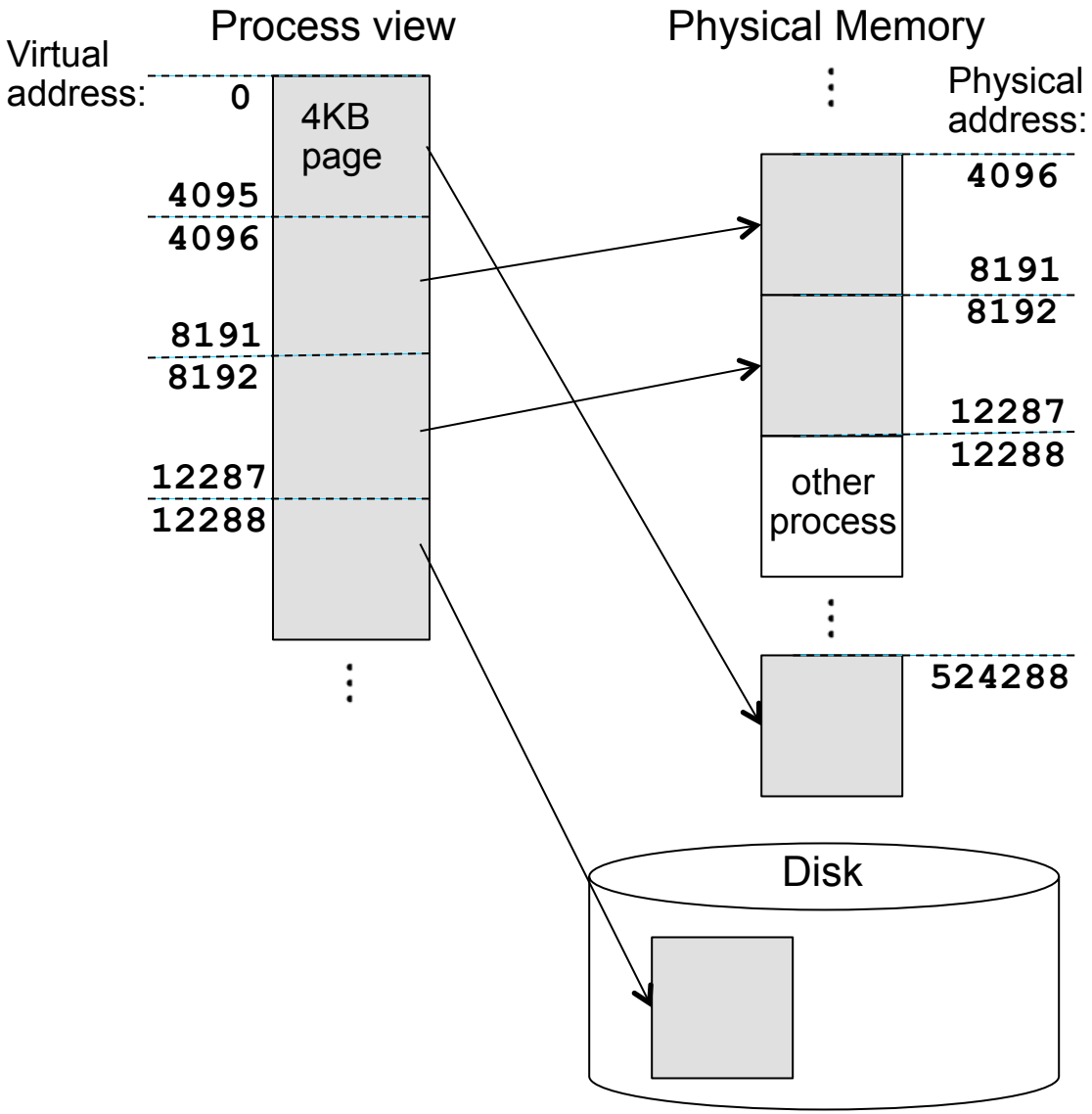


How to get data to the GPU (and back):

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

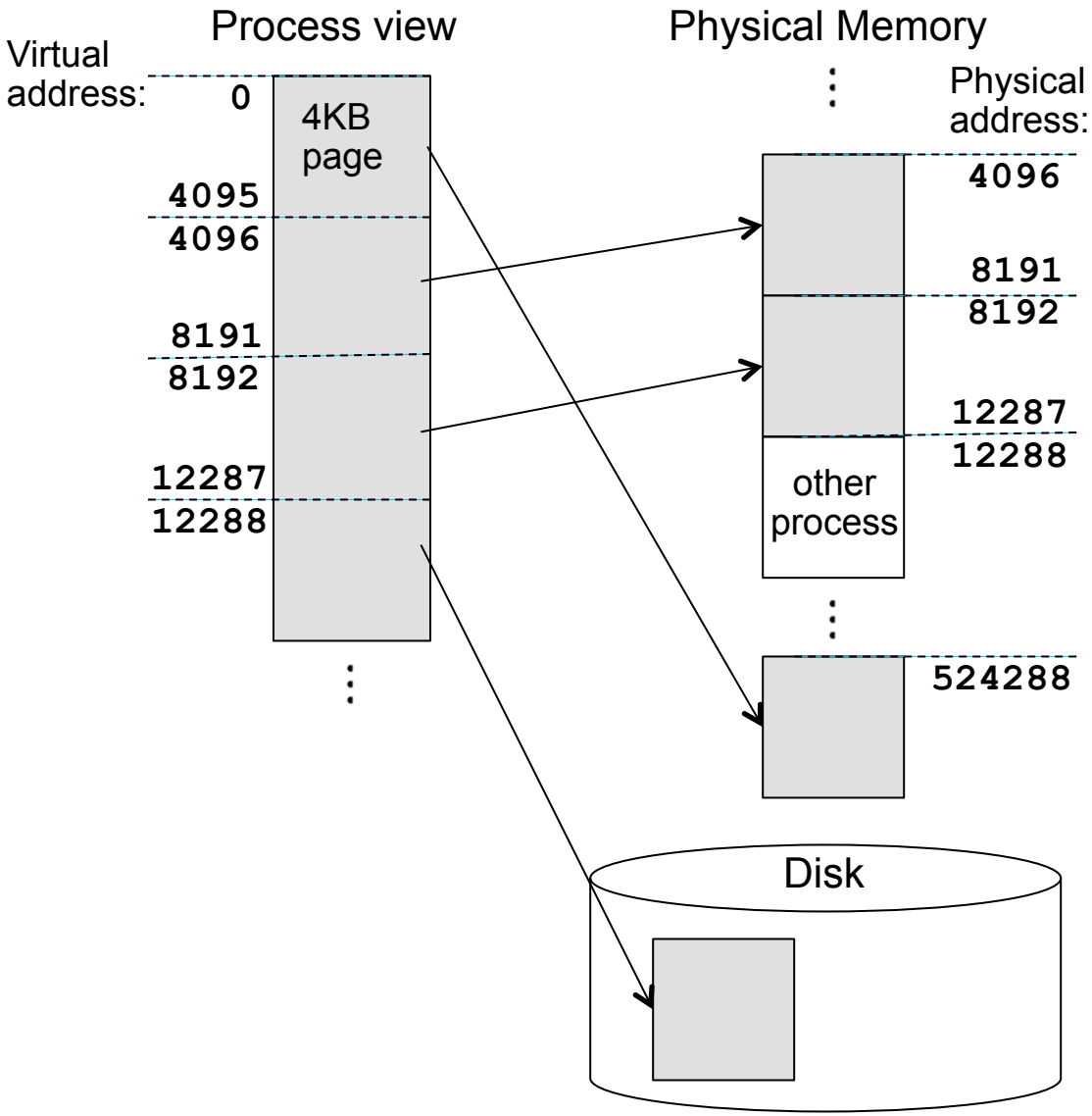
```
cudaHostRegister(h_ipt,ipt_size,cudaHostRegisterMapped);
cudaHostRegister(h_opt,opt_size,cudaHostRegisterMapped);
cudaHostGetDevicePointer(&ipt_ptr,h_ipt,0);
cudaHostGetDevicePointer(&opt_ptr,h_opt,0);
```

Zero Copy Access & Virtual memory management



- 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



- 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;
}
```

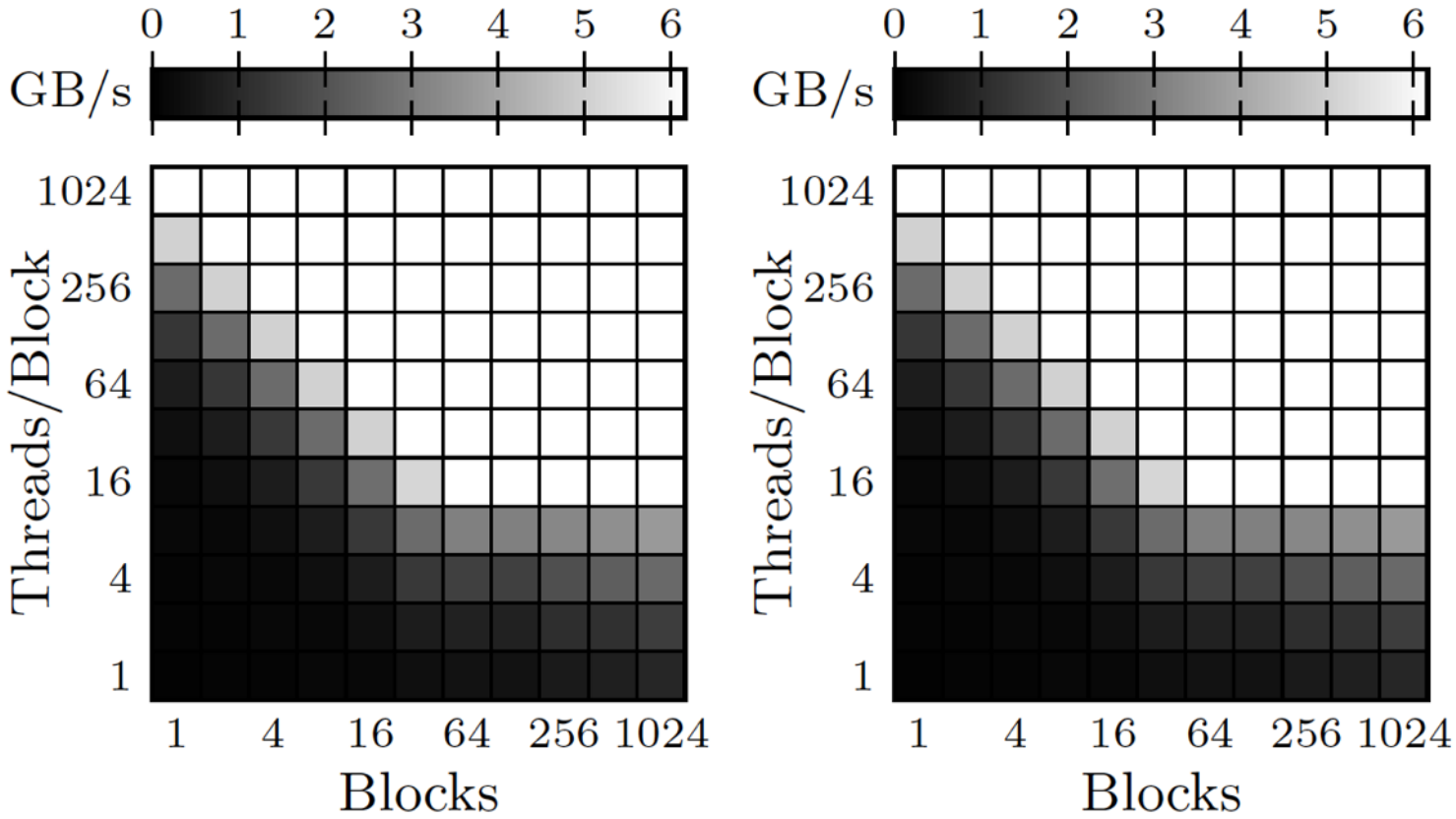
How to get data to the GPU (and back):

- 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

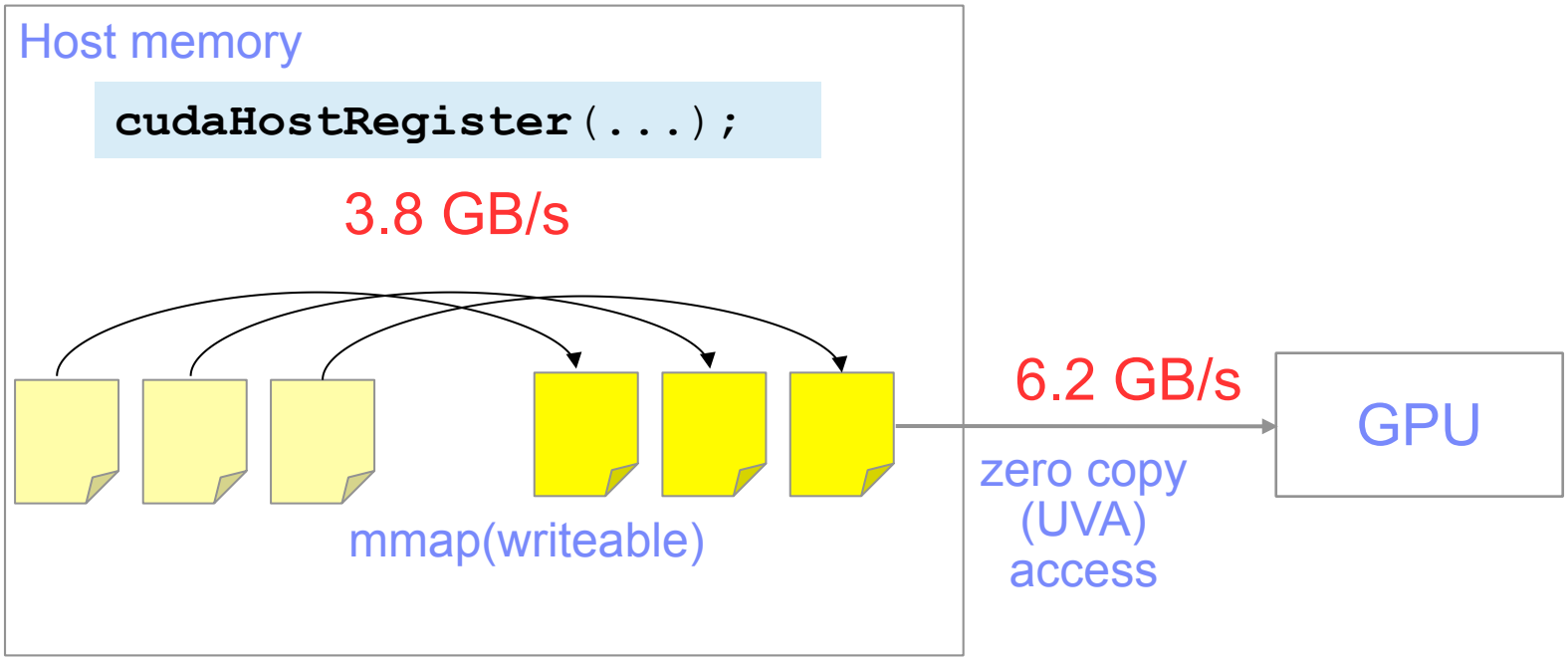


64-bit coalesced read/write access to host memory using ZCA on a GTX580

- 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?)

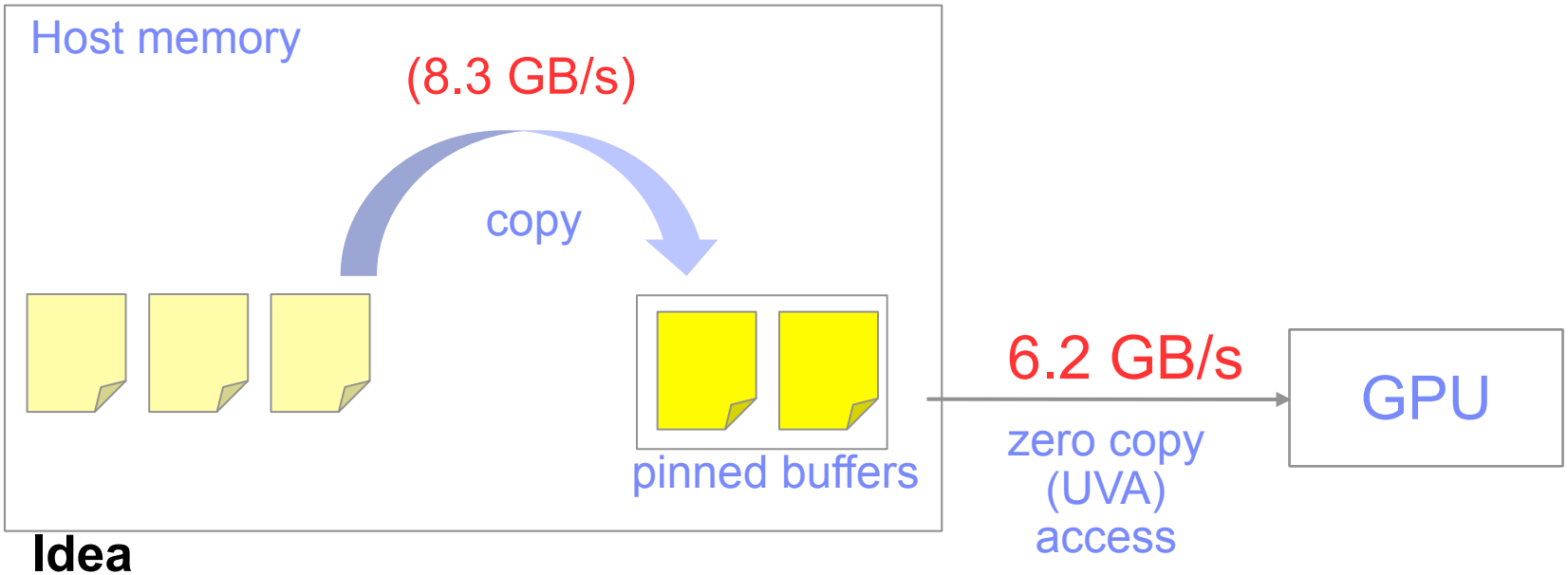
→ copy page before pinning

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?

Agenda

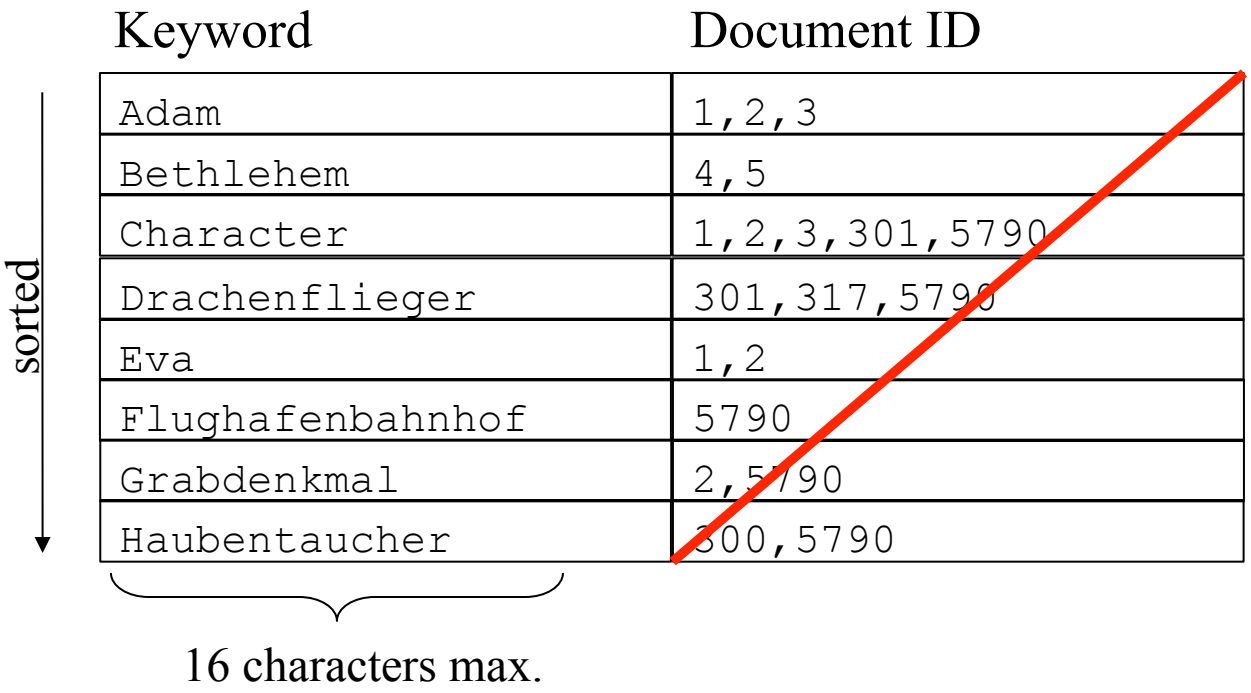
Large Scale Data Management on the GPU

- 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

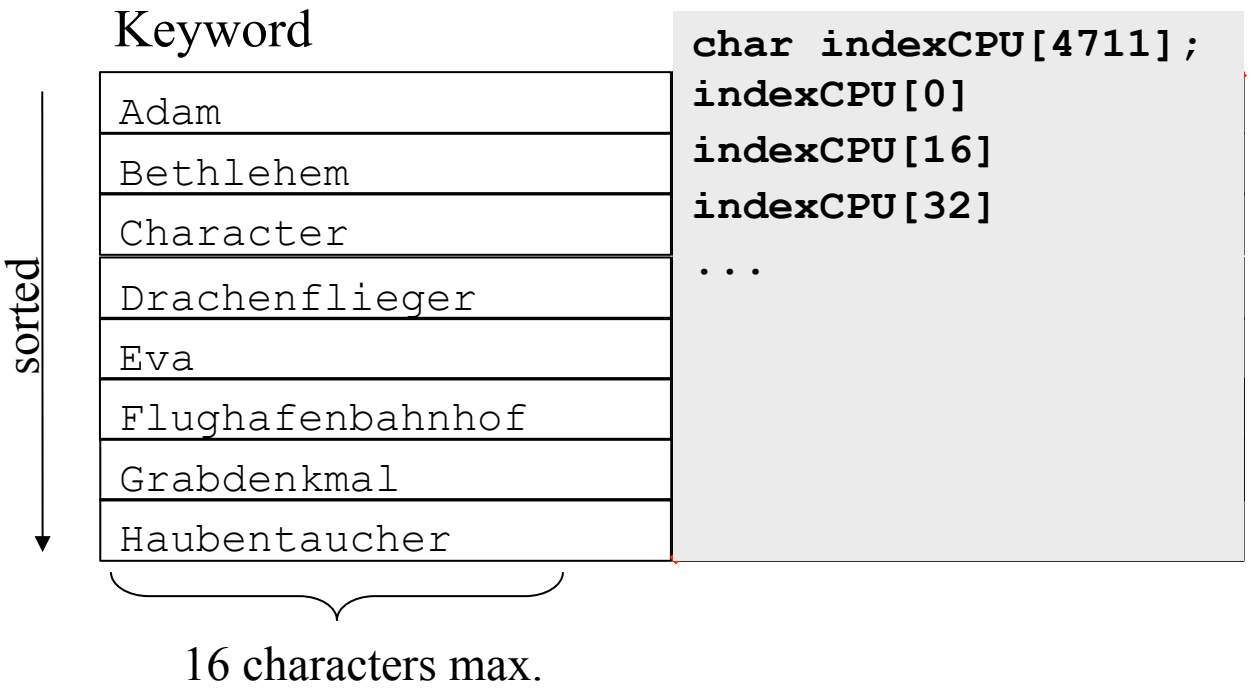
A Simple implementation of (index) search

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

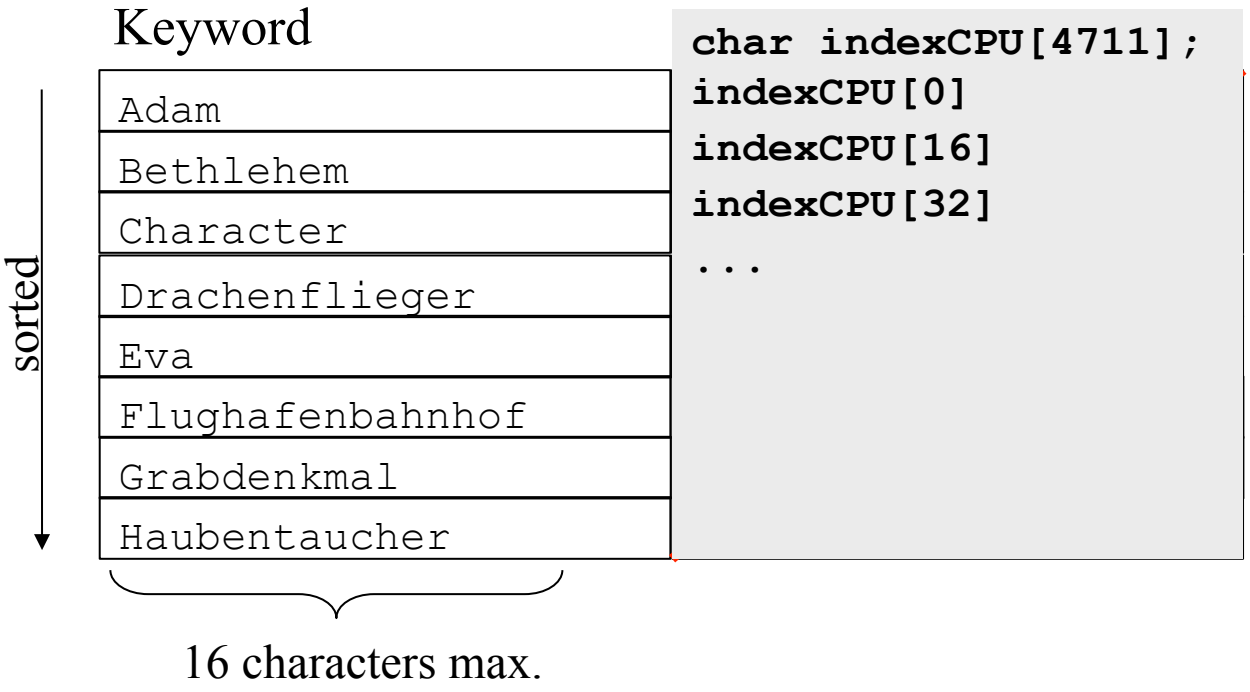
A Simple implementation of (index) search



A Simple implementation of (index) search



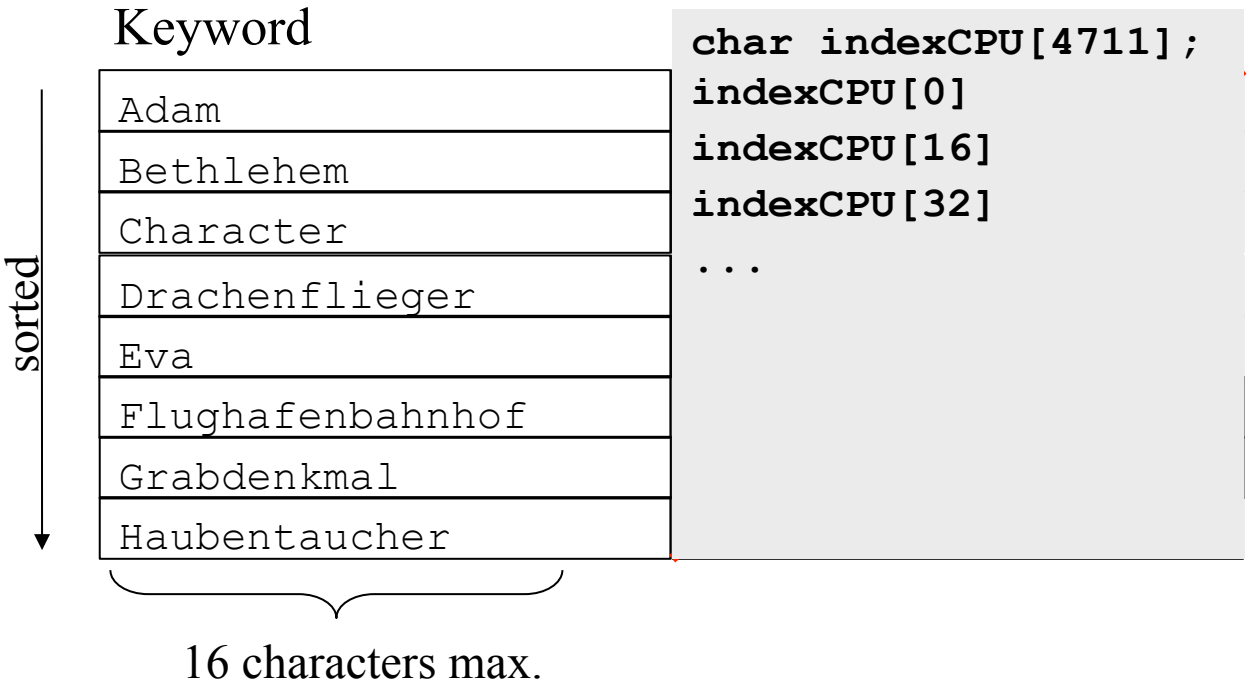
A Simple implementation of (index) search



- 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



- 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(30,0,0);
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

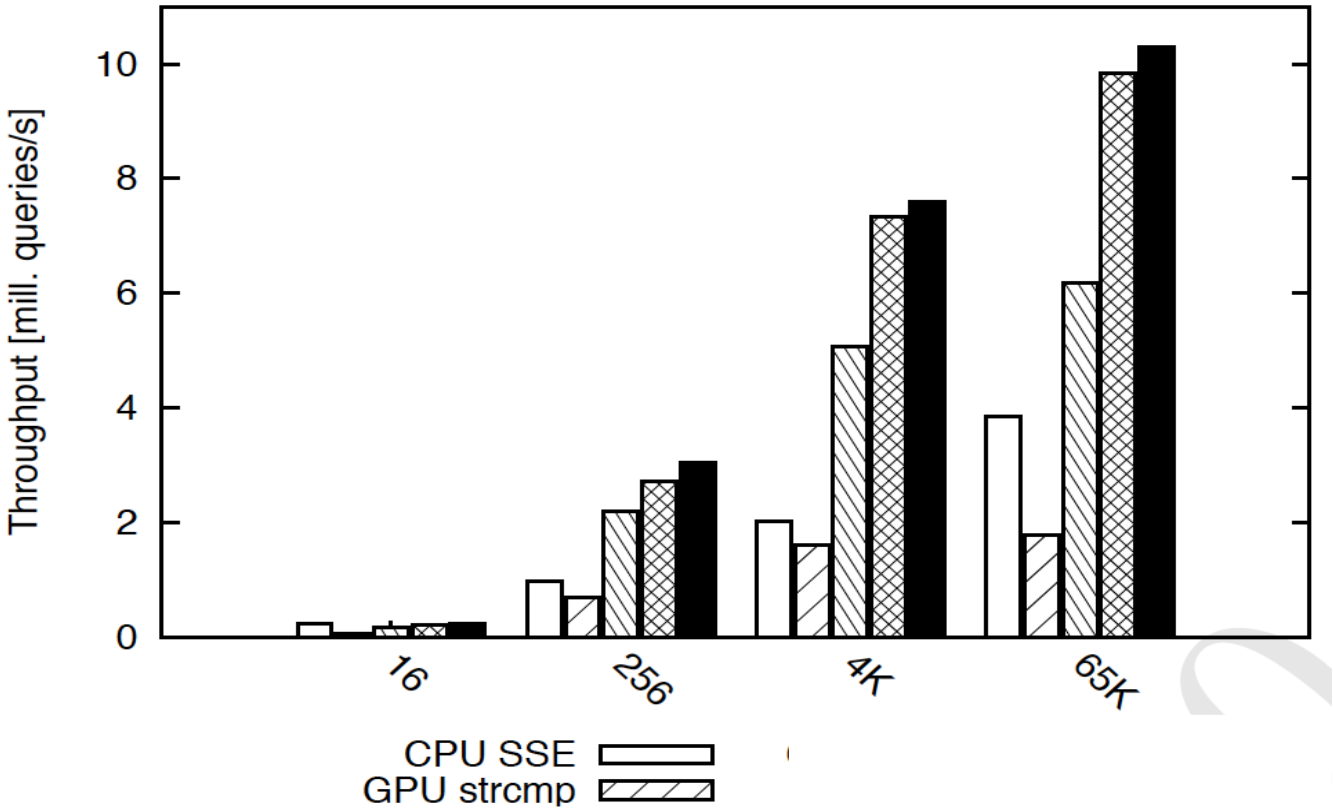
```
__device__ char* bsearchGPU(char *key, char *base, int n, int size){
    char *mid_point;
    int  cmp;

    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));
}
```

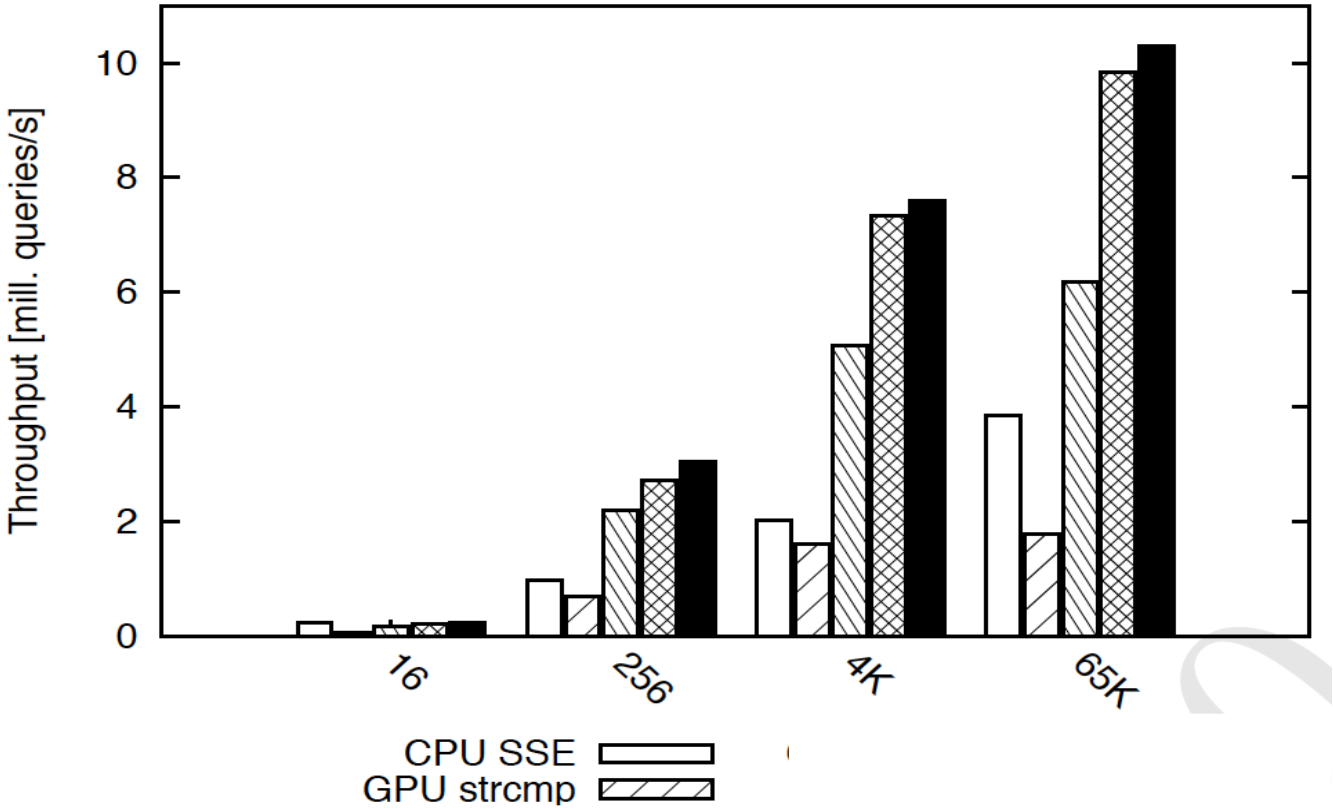
Binary Search on the GPU

- 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

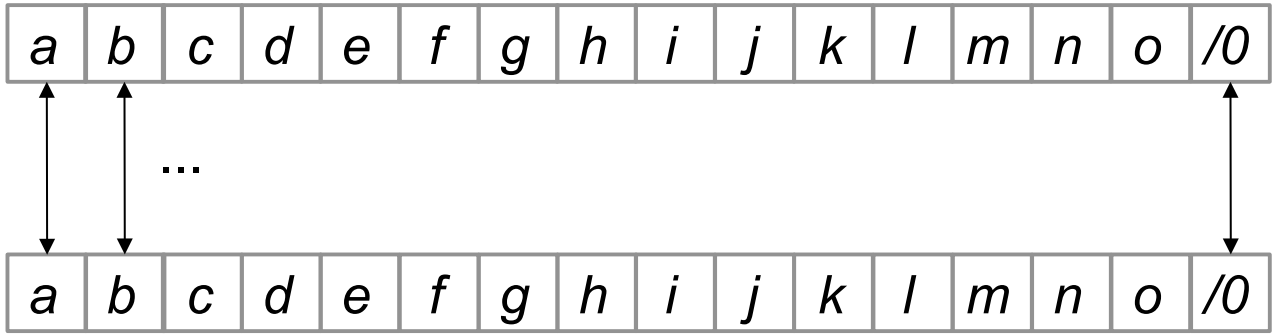


- It's slower than a CPU implementation for all data set sizes!
  - Let's try some optimizations ...

Search requires to compare

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

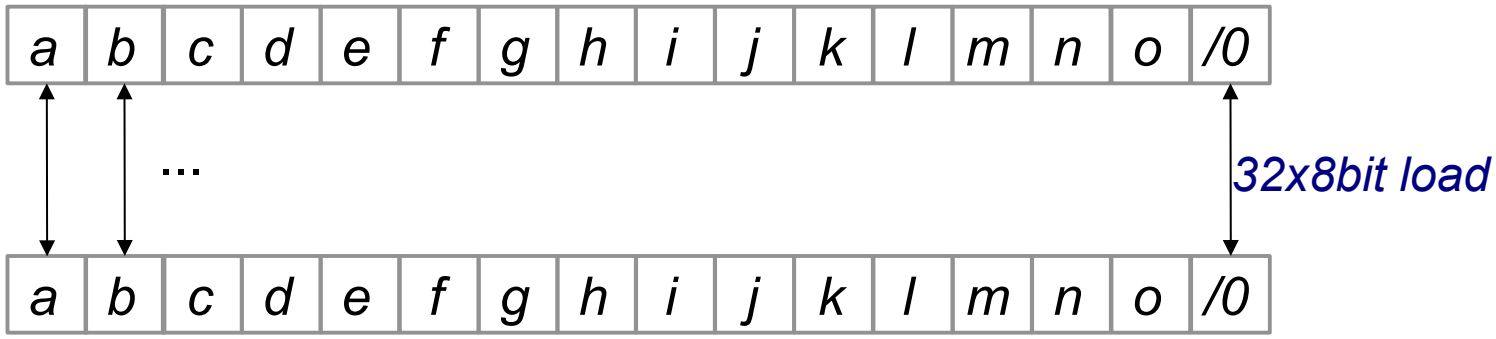
```
int strcmp(const char* s1, const char* s2){
        while (*s1 == *s2++)
                if (*s1++ == 0)return 0;
        return (*s1 - *(s2 - 1));
}
```



Search requires to compare

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

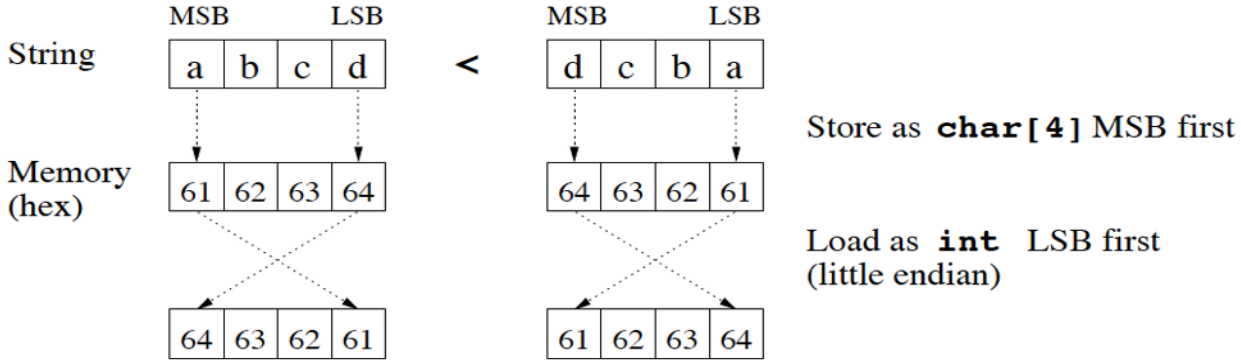
```
int strcmp(const char* s1, const char* s2){
        while (*s1 == *s2++)
                if (*s1++ == 0)return 0;
        return (*s1 - *(s2 - 1));
}
```



- Byte-wise memory access is known to be slow

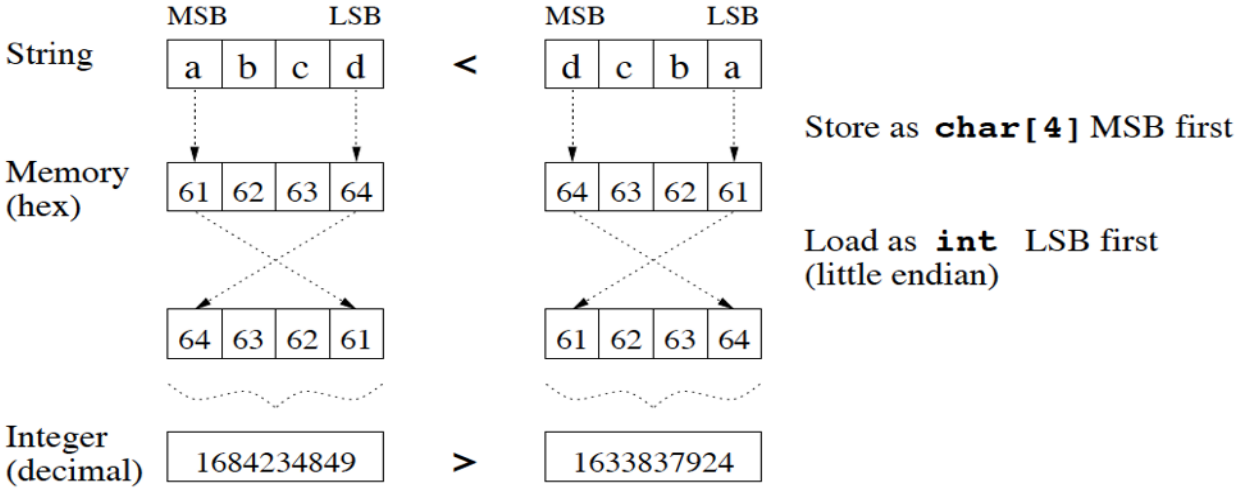
Optimizing compare operations

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



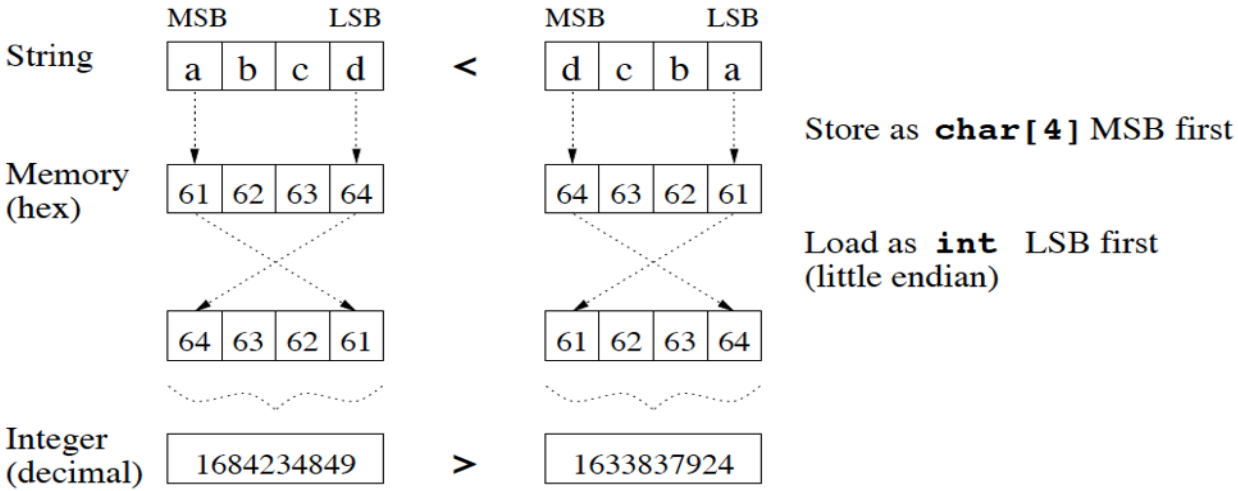
Optimizing compare operations

- 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

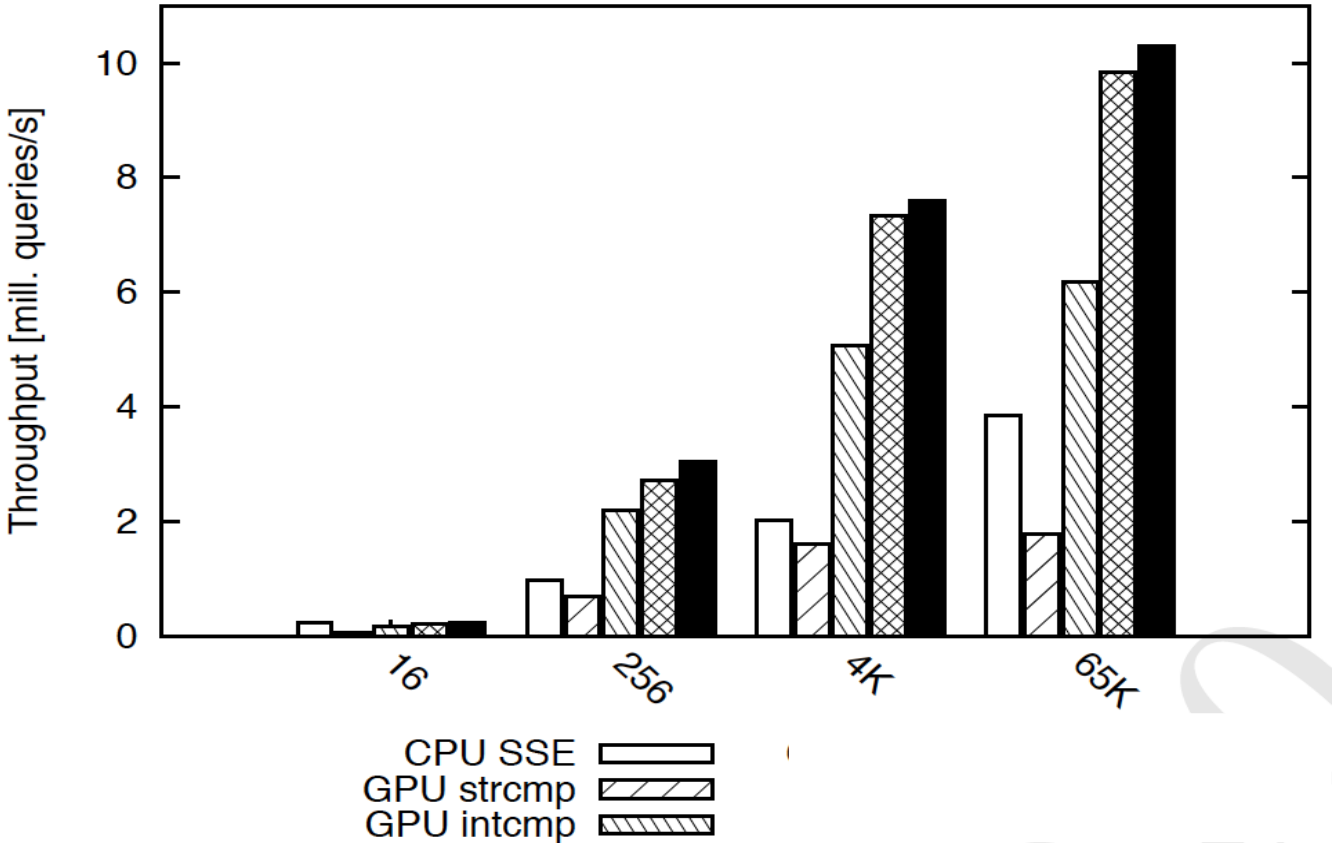
- Comparing integer vectors (bswap for <> skipped for clarity)

```
__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

- We can use shared memory like L1

x 16 for each comparison !!!

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

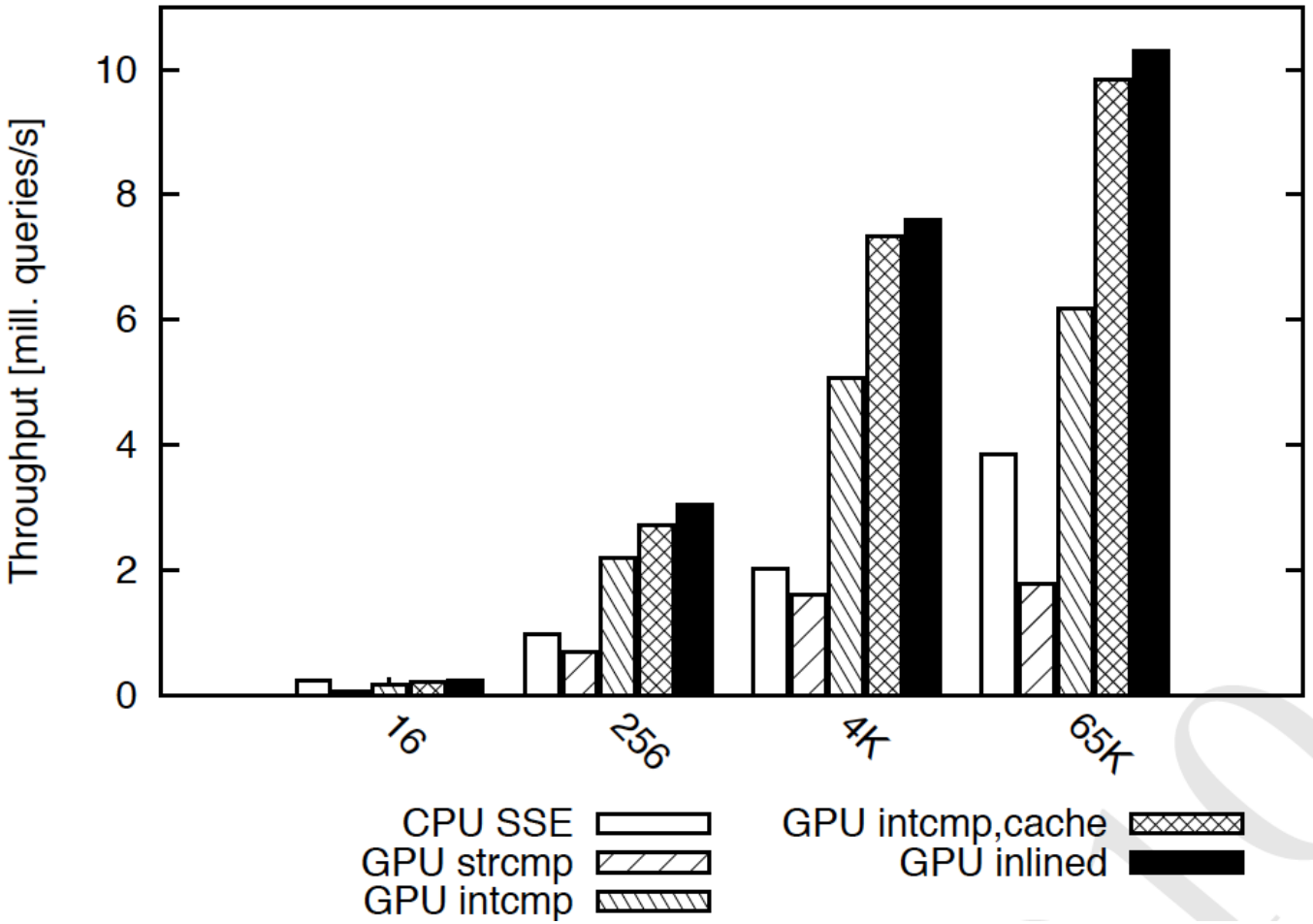
- We can use shared memory like L1

x 16 for each comparison !!!

```
__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

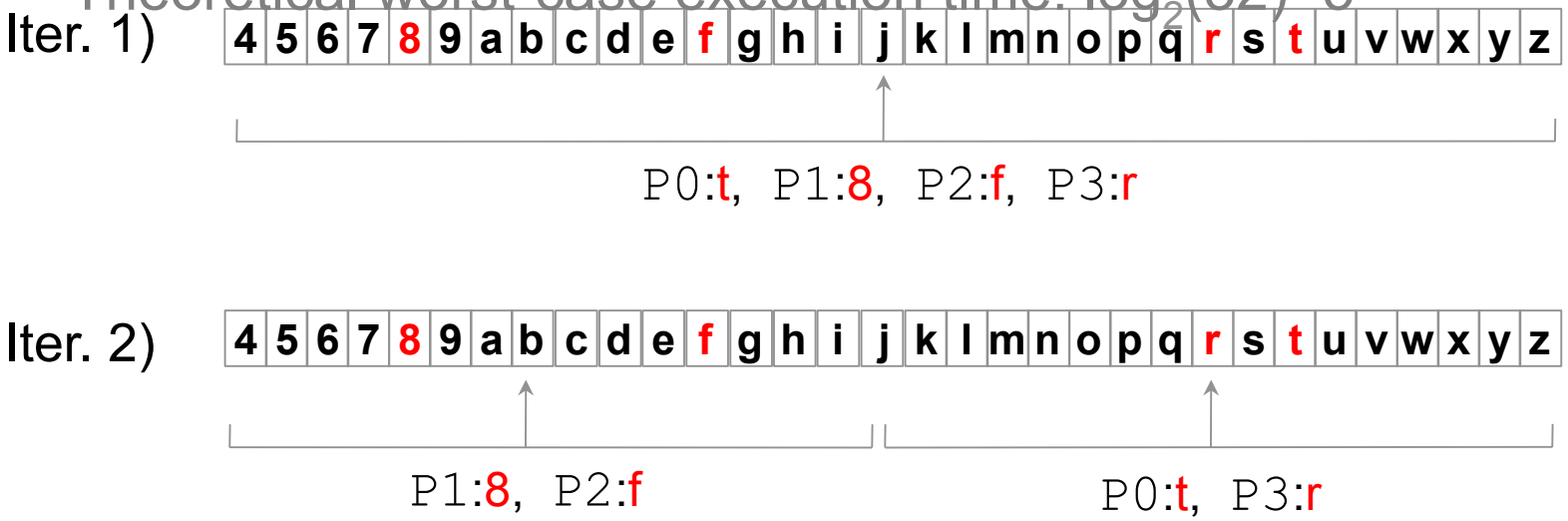


Multi-threaded Binary Search – Example

- 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: $\log_2(32)=5$

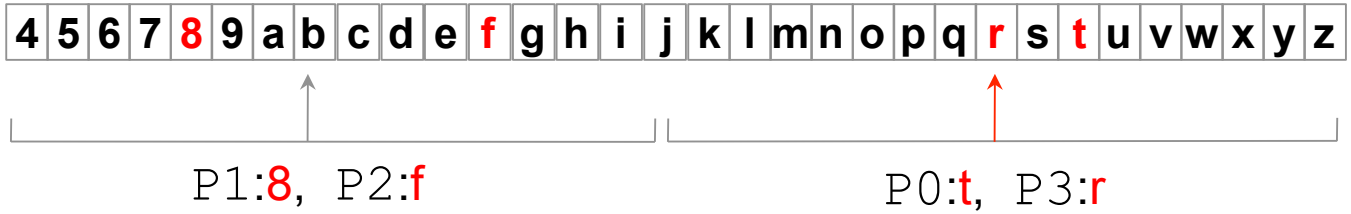
Multi-threaded Binary Search – Example

- 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: $\log_2(32)=5$

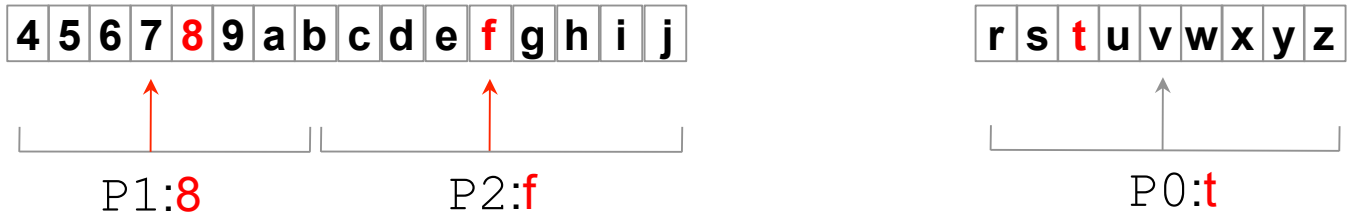


Multi-threaded Binary Search – Example

Iter. 2)

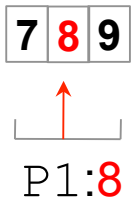


Iter. 3)



Iter. 4)

Iter. 5)



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?

Questions?

Appendix: CPU caches

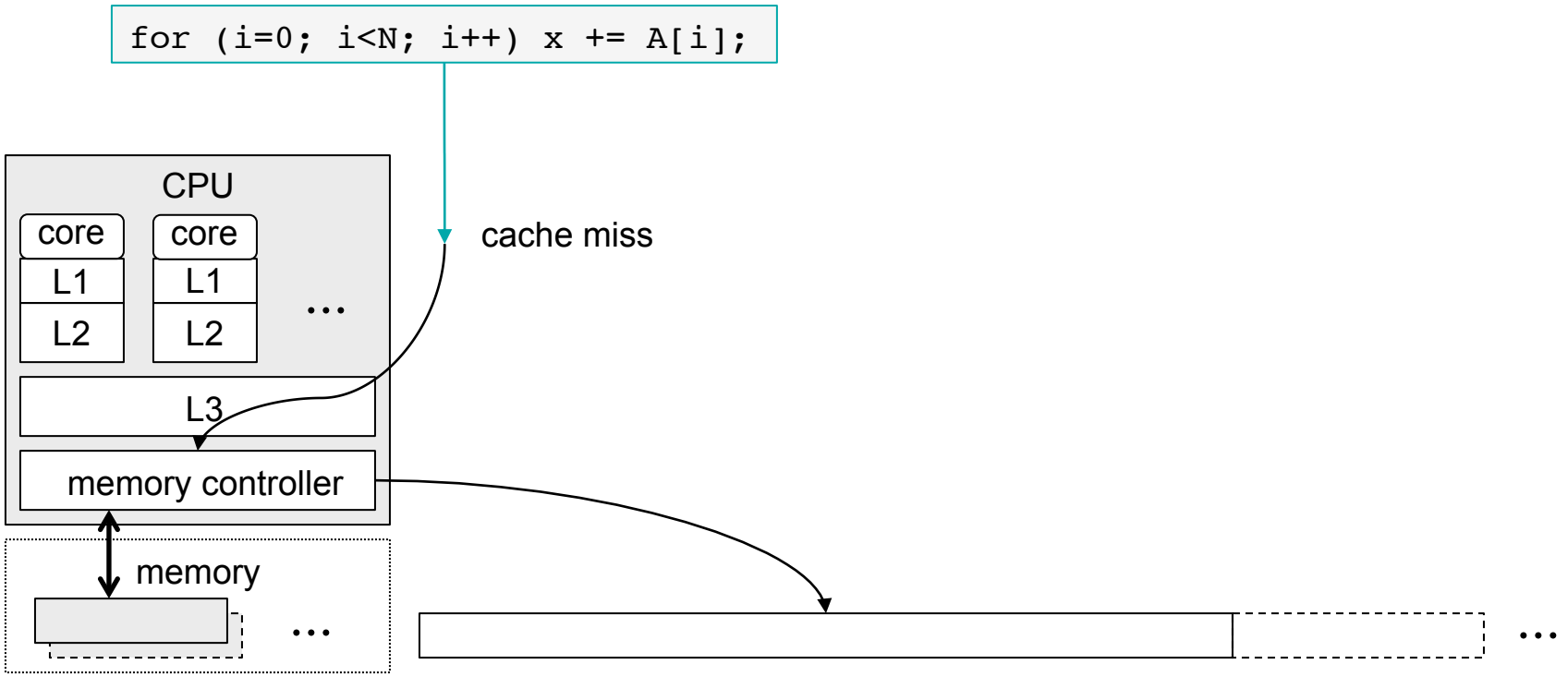
Caches – the good

- Memory is physically accessed at cache line granularity, e.g. 64Byte
- Sequential memory access:

```
for (i=0; i<N; i++) x += A[i];
```

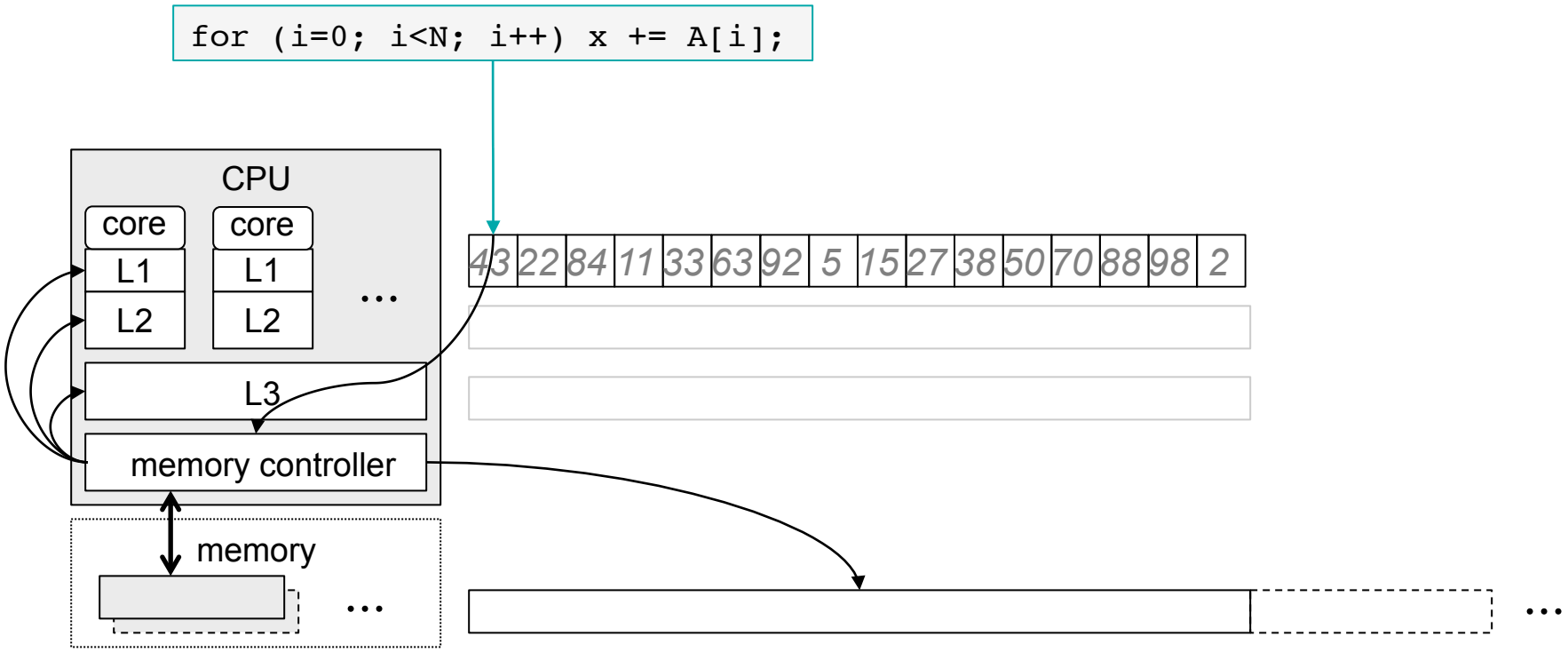
Caches – the good

- Memory is physically accessed at cache line granularity, e.g. 64Byte
- Sequential memory access:



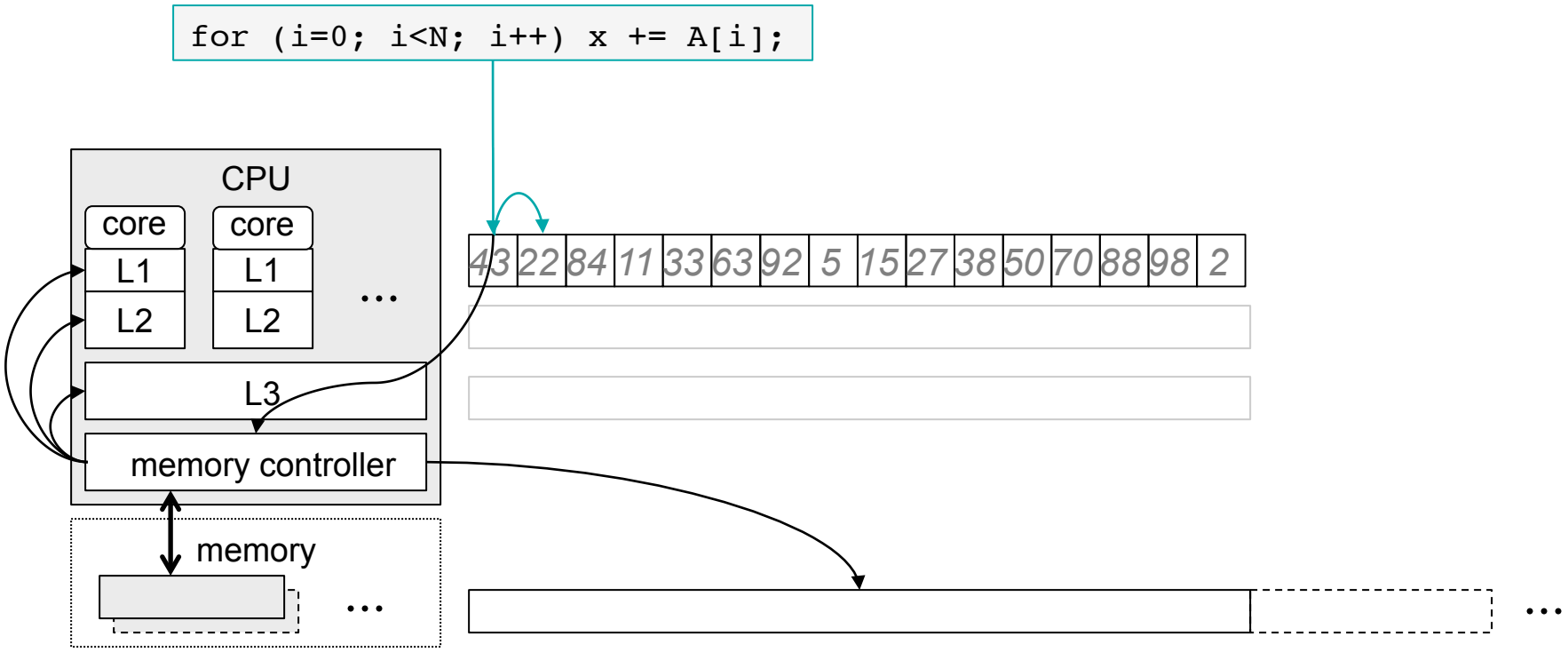
Caches – the good

- Memory is physically accessed at cache line granularity, e.g. 64Byte
- Sequential memory access:



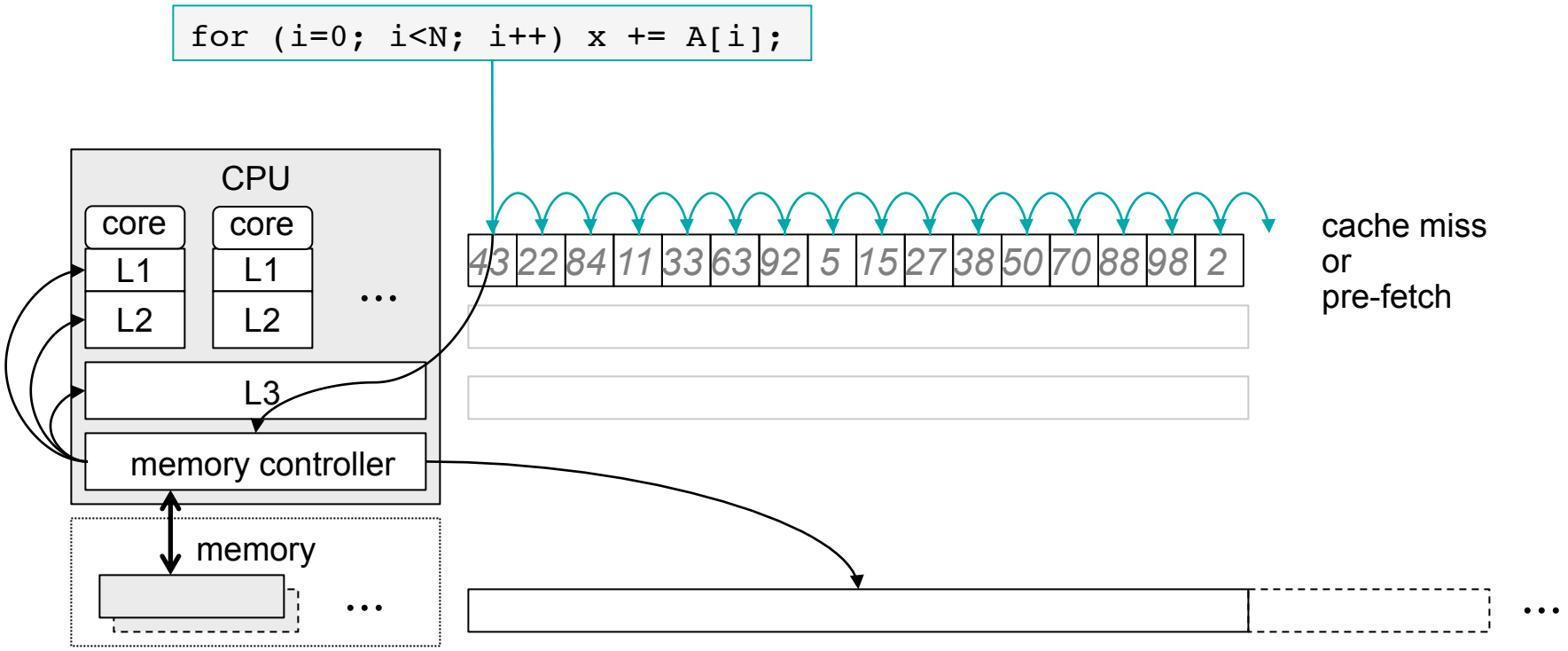
Caches – the good

- Memory is physically accessed at cache line granularity, e.g. 64Byte
- Sequential memory access:



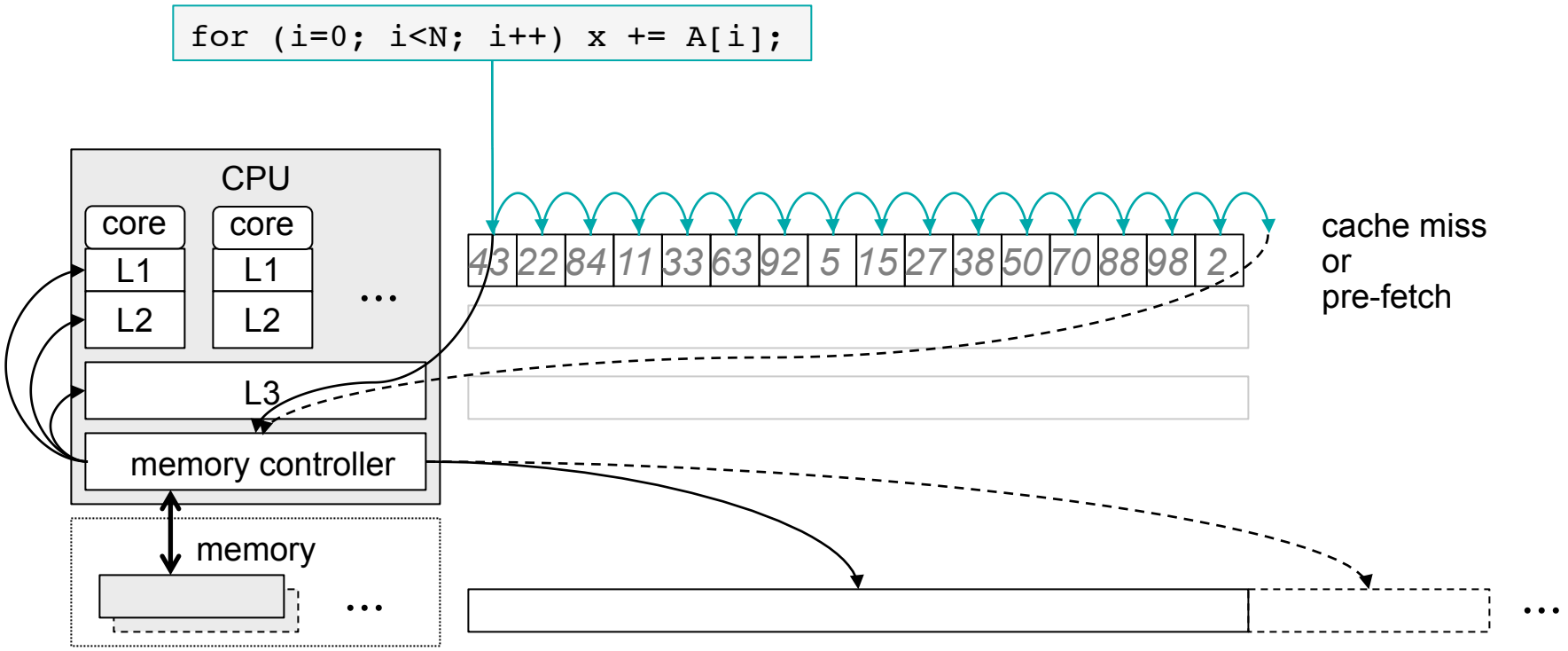
Caches – the good

- Memory is physically accessed at cache line granularity, e.g. 64Byte
- Sequential memory access:



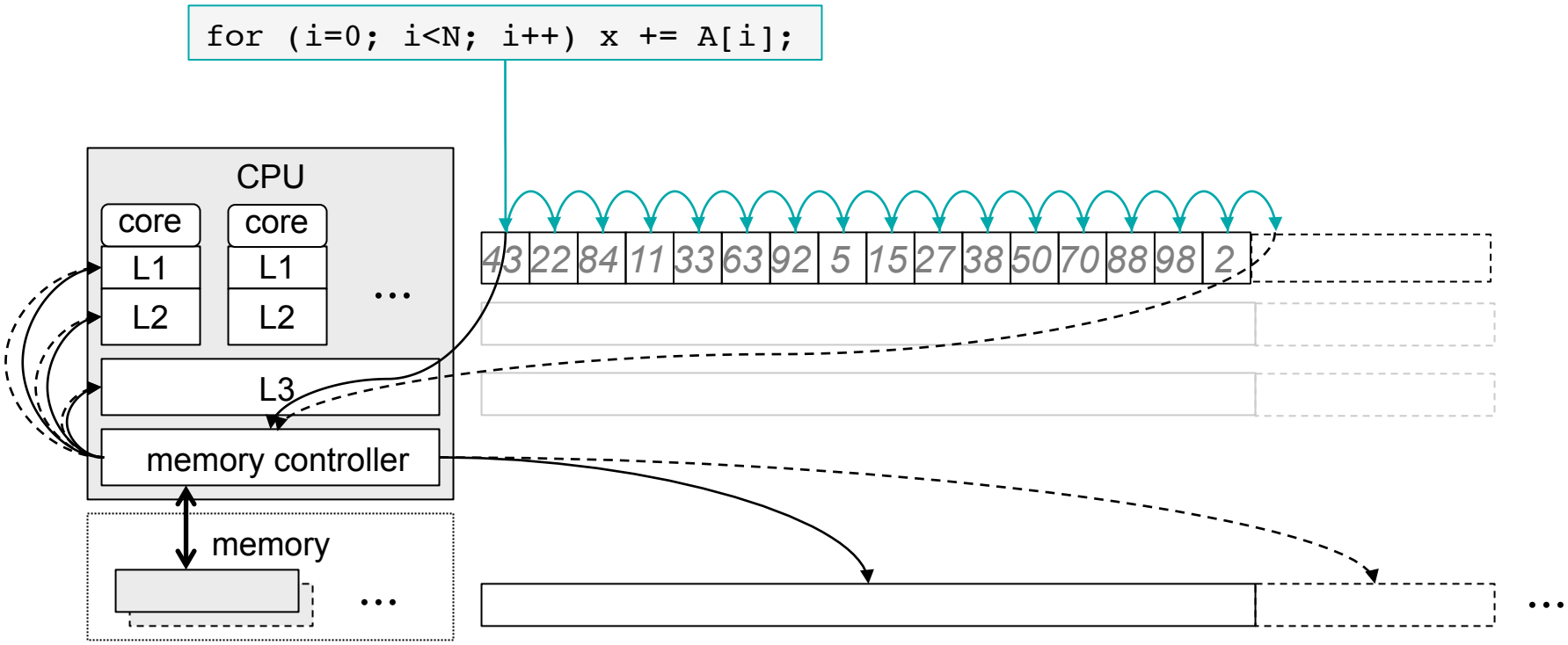
Caches – the good

- Memory is physically accessed at cache line granularity, e.g. 64Byte
- Sequential memory access:



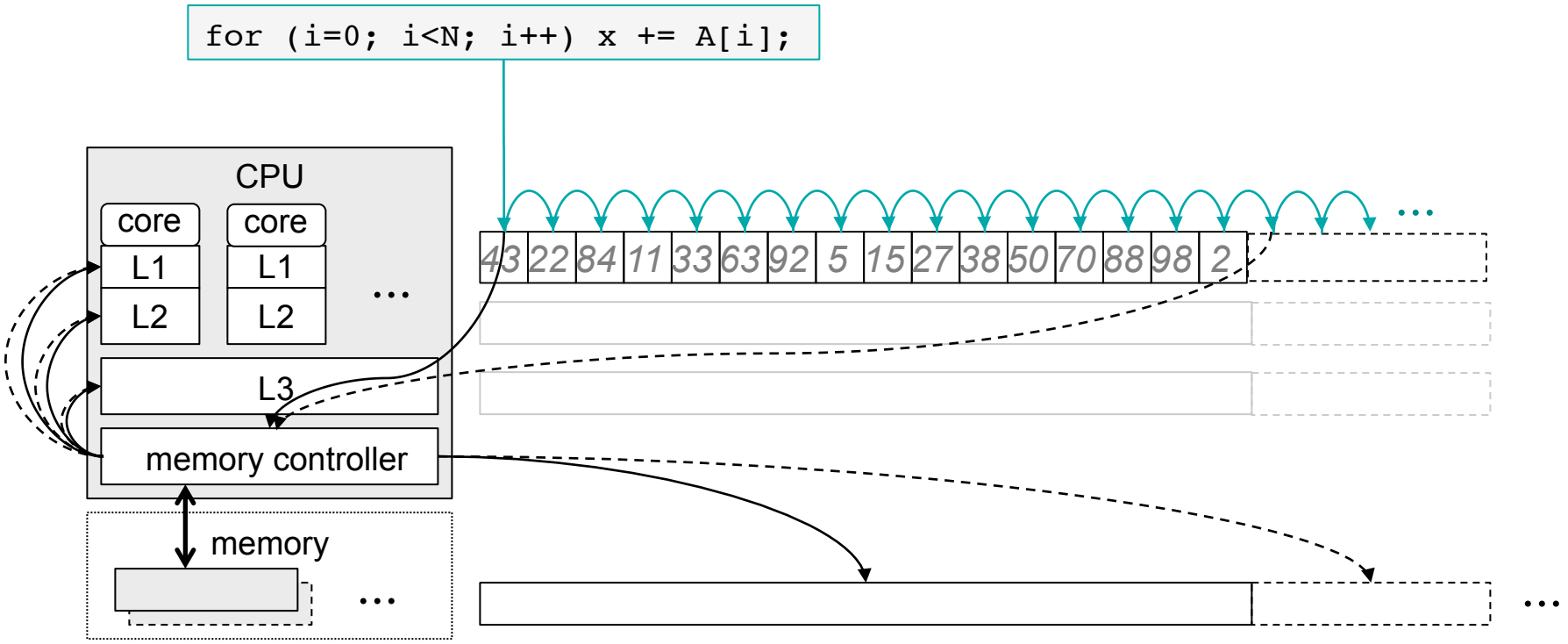
Caches – the good

- Memory is physically accessed at cache line granularity, e.g. 64Byte
- Sequential memory access:



Caches – the good

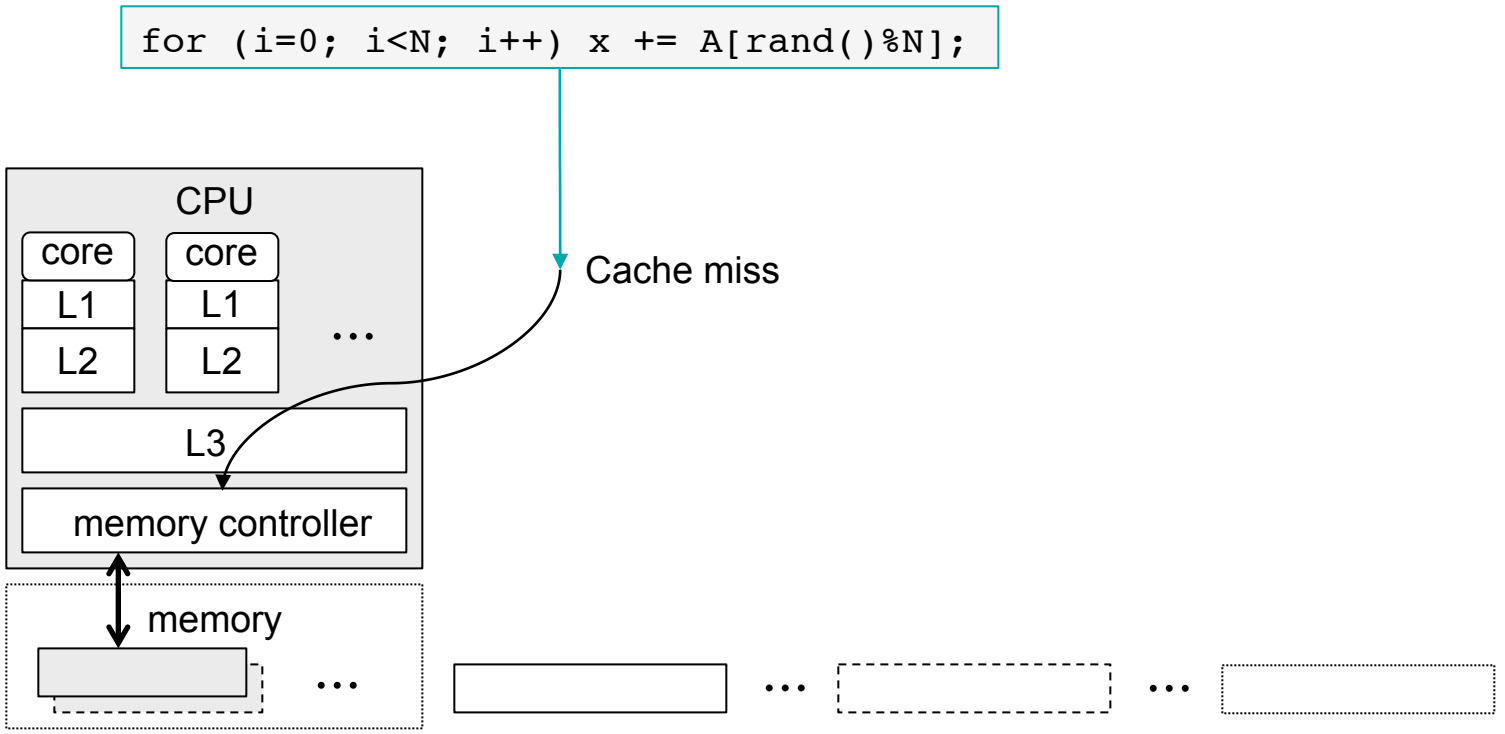
- Memory is physically accessed at cache line granularity, e.g. 64Byte
- Sequential memory access:



➔ linear memory access maximizes cache & bandwidth utilization

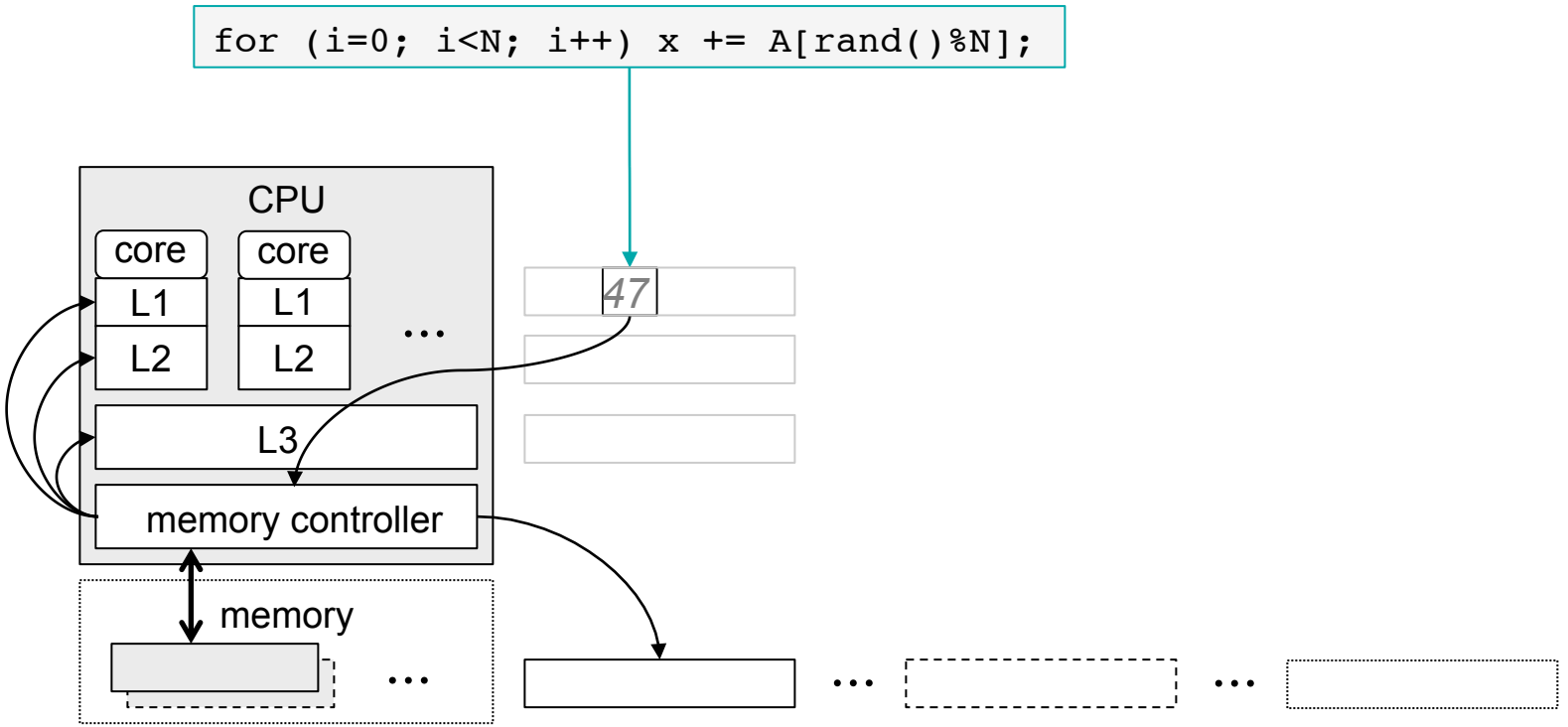
Caches – the bad

- Memory is physically accessed at cache line granularity, e.g. 64Byte
- Random memory access:



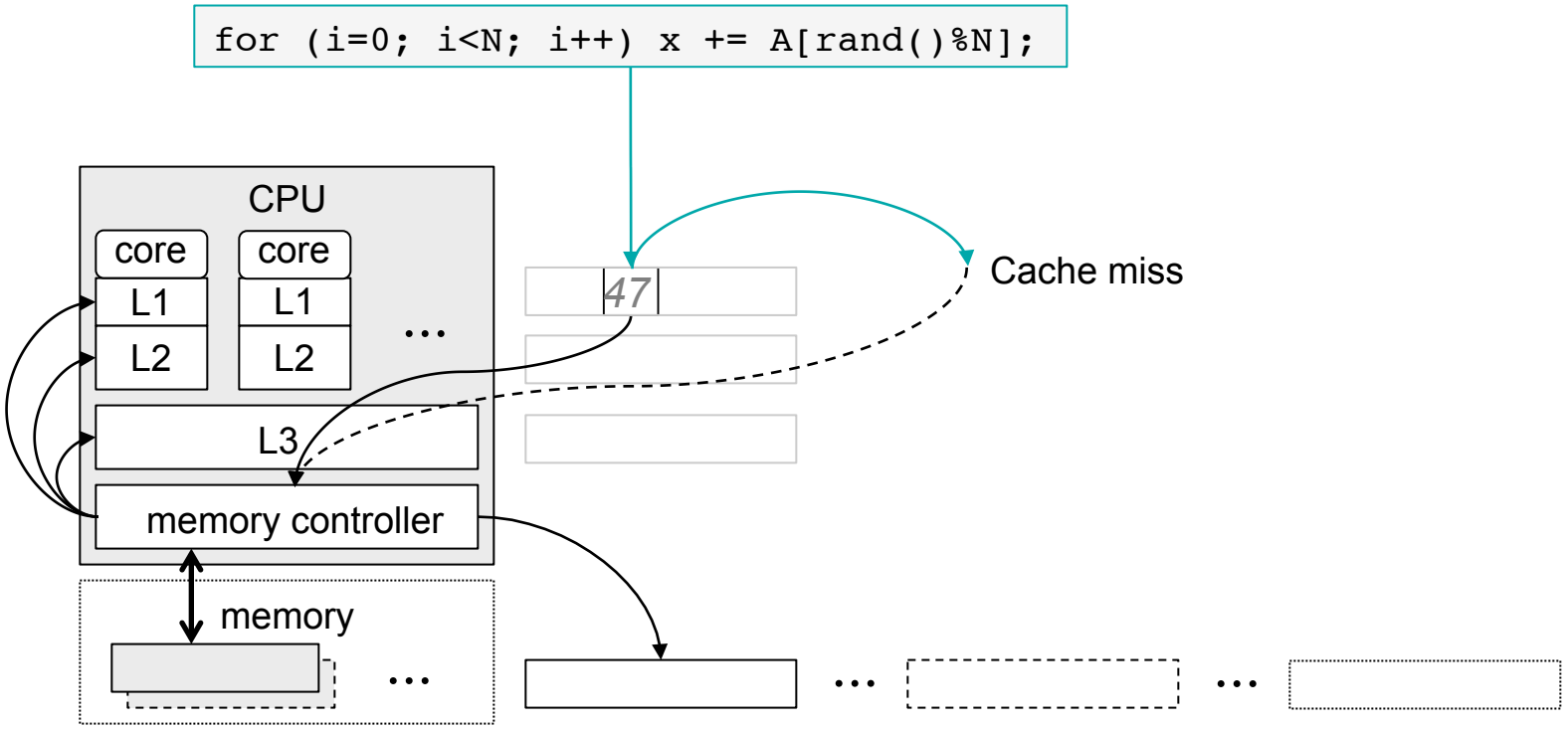
Caches – the bad

- Memory is physically accessed at cache line granularity, e.g. 64Byte
- Random memory access:



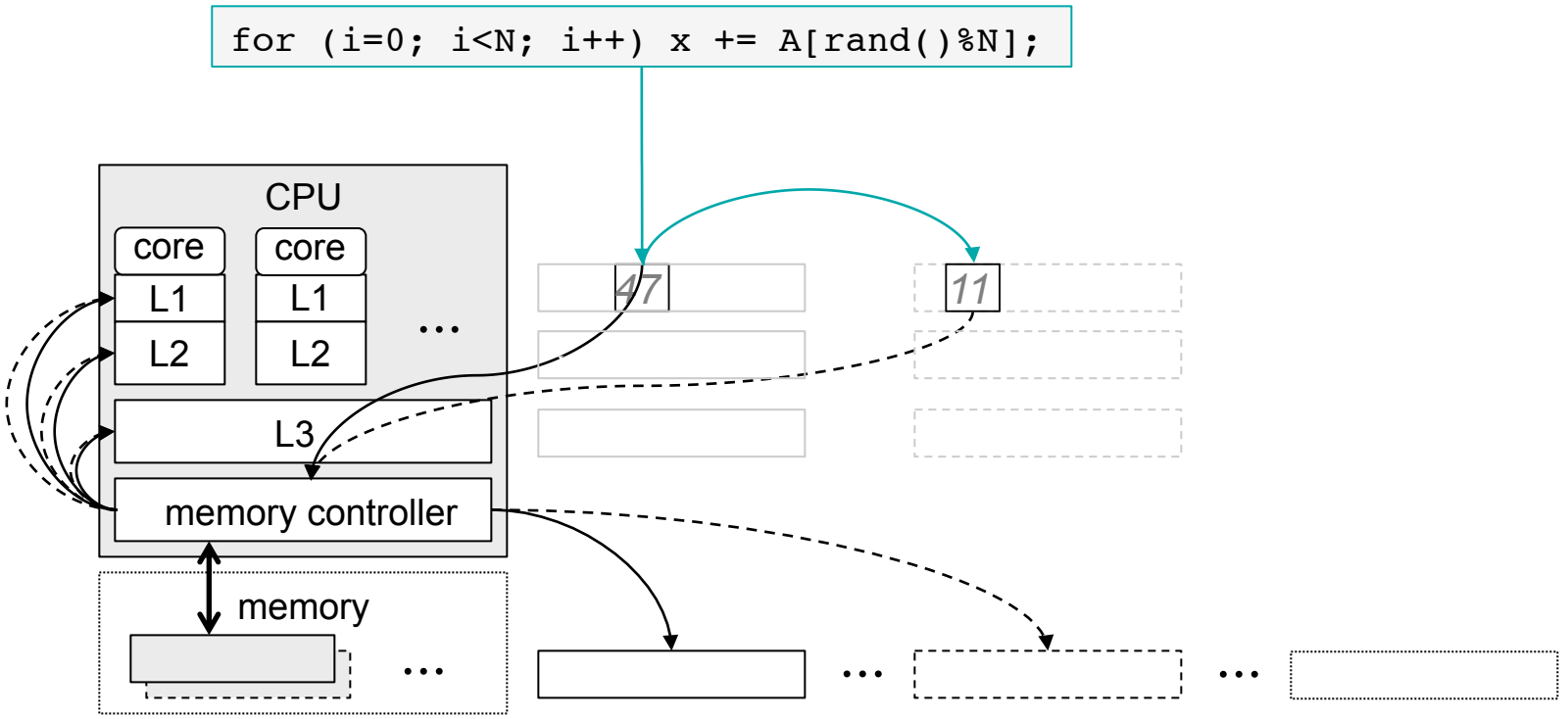
Caches – the bad

- Memory is physically accessed at cache line granularity, e.g. 64Byte
- Random memory access:



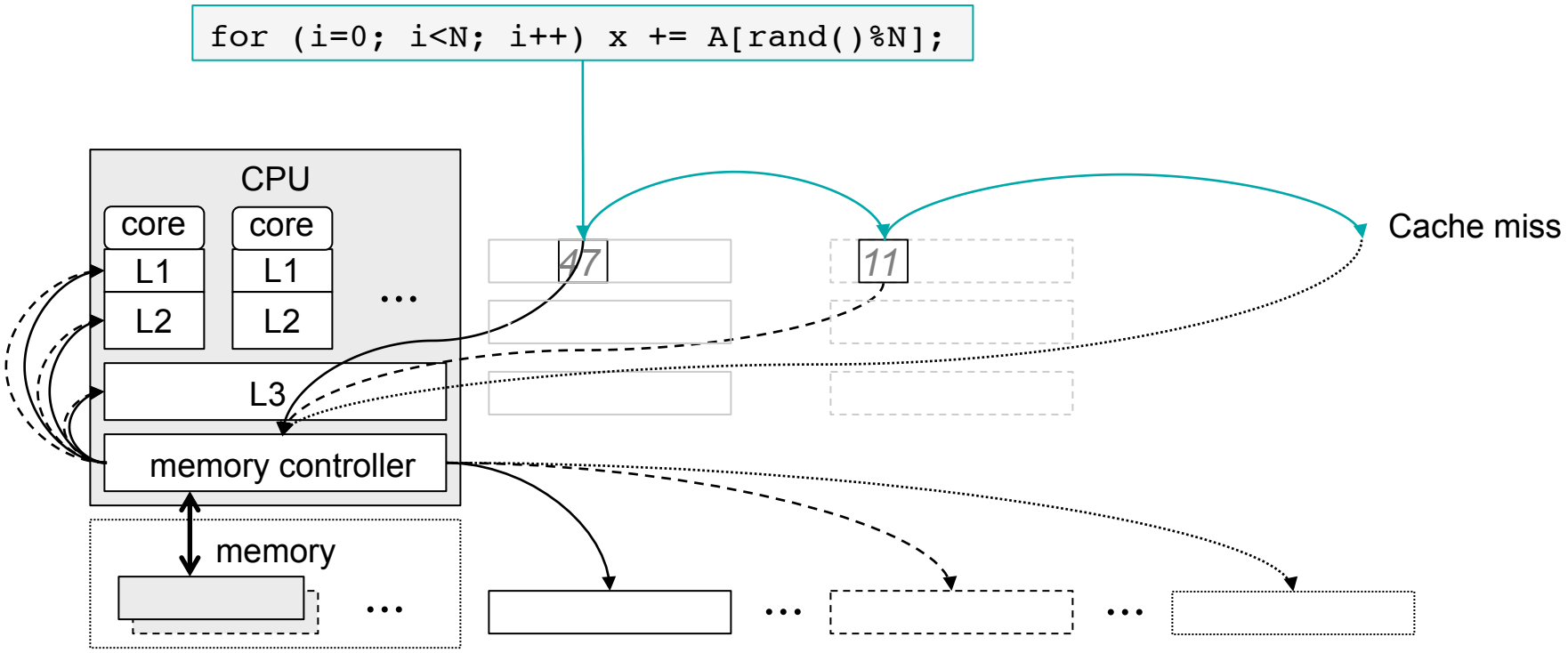
Caches – the bad

- Memory is physically accessed at cache line granularity, e.g. 64Byte
- Random memory access:



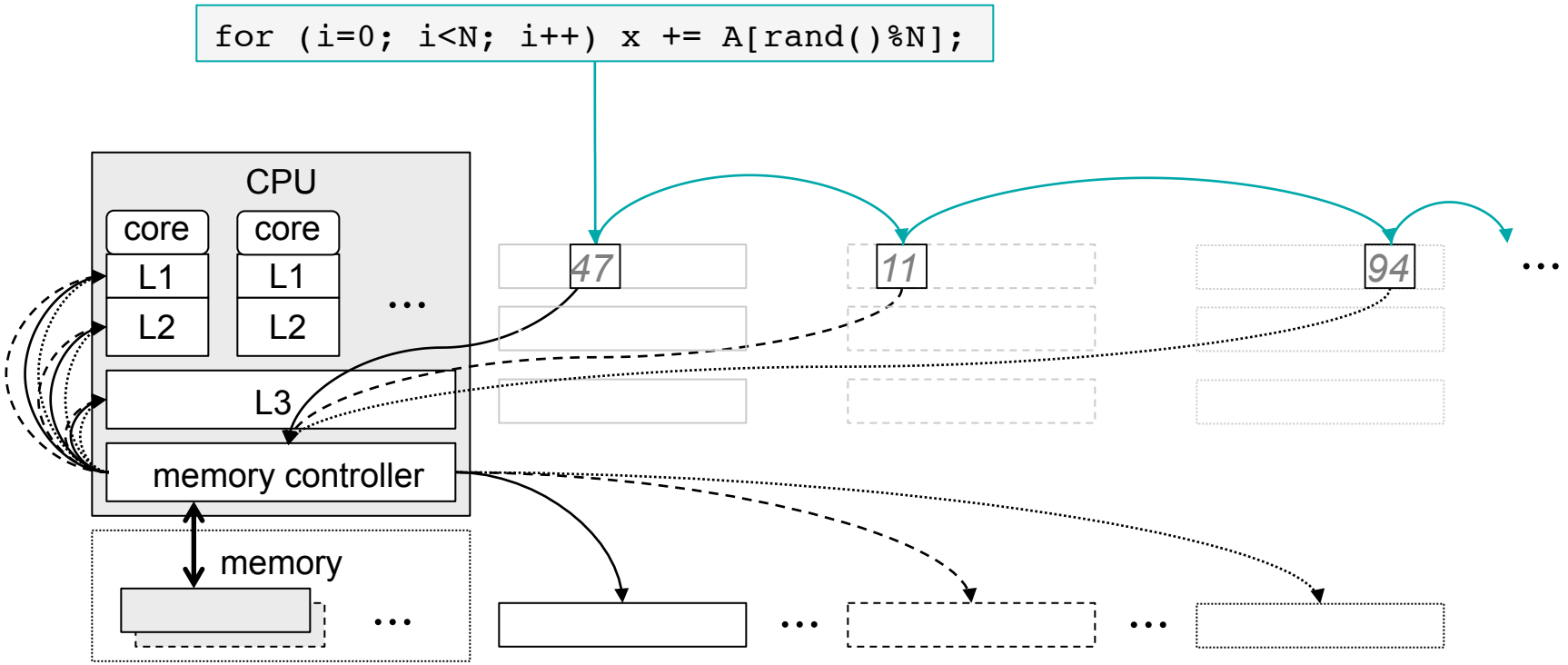
Caches – the bad

- Memory is physically accessed at cache line granularity, e.g. 64Byte
- Random memory access:



Caches – the bad

- Memory is physically accessed at cache line granularity, e.g. 64Byte
- Random memory access:



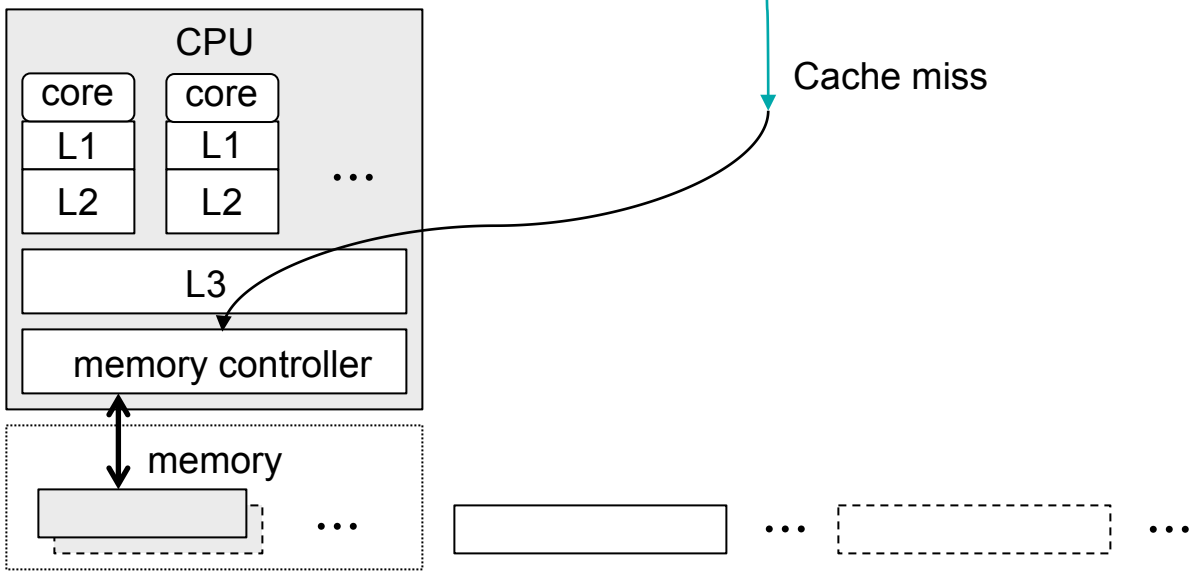
➔ Random memory access wastes up to 98.5%* of bandwidth

* assuming we only need 1 Byte of a 64 Byte cache line

Caches – the ugly

- Memory is physically accessed at cache line granularity, e.g. 64Byte
- Writes:

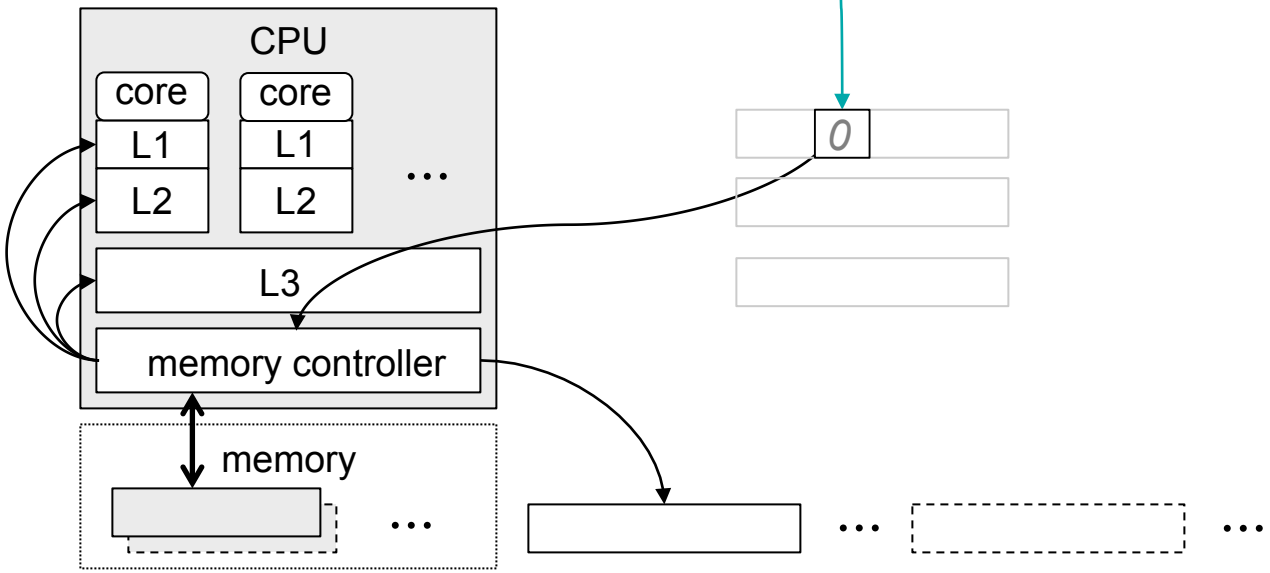
```
for (i=0; i<N; i++) A[rand()%N] = i;
```



Caches – the ugly

- Memory is physically accessed at cache line granularity, e.g. 64Byte
- Writes:

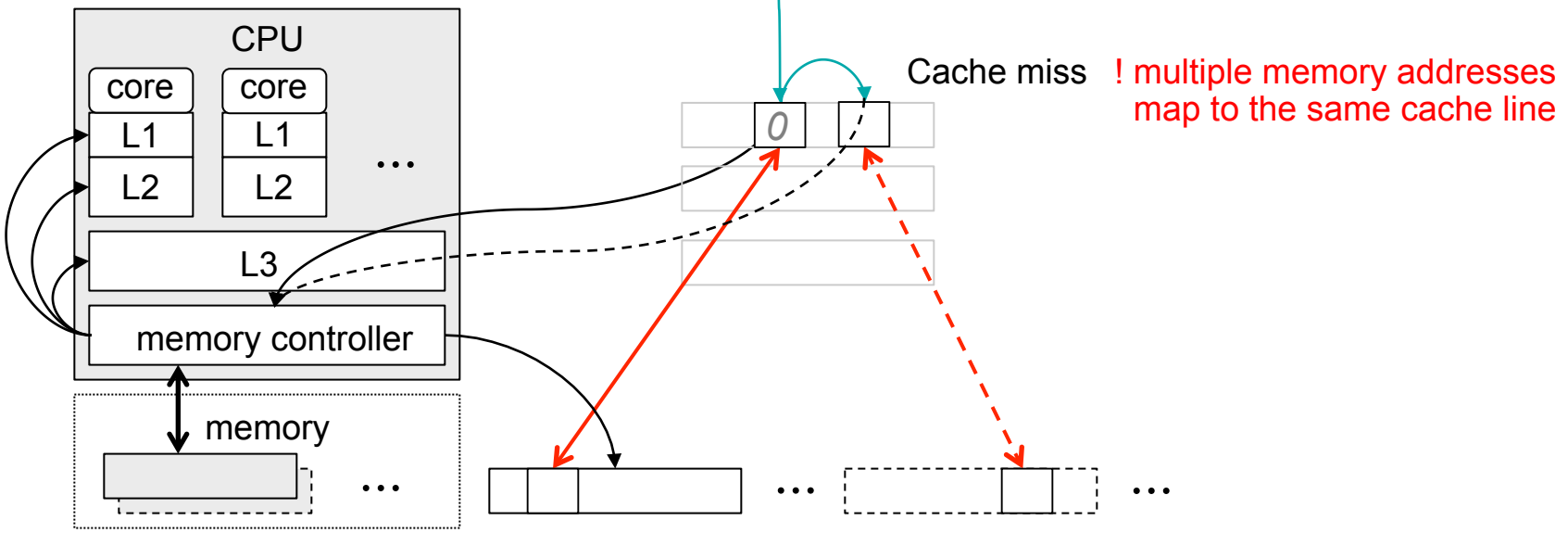
```
for (i=0; i<N; i++) A[rand()%N] = i;
```



Caches – the ugly

- Memory is physically accessed at cache line granularity, e.g. 64Byte
- Writes:

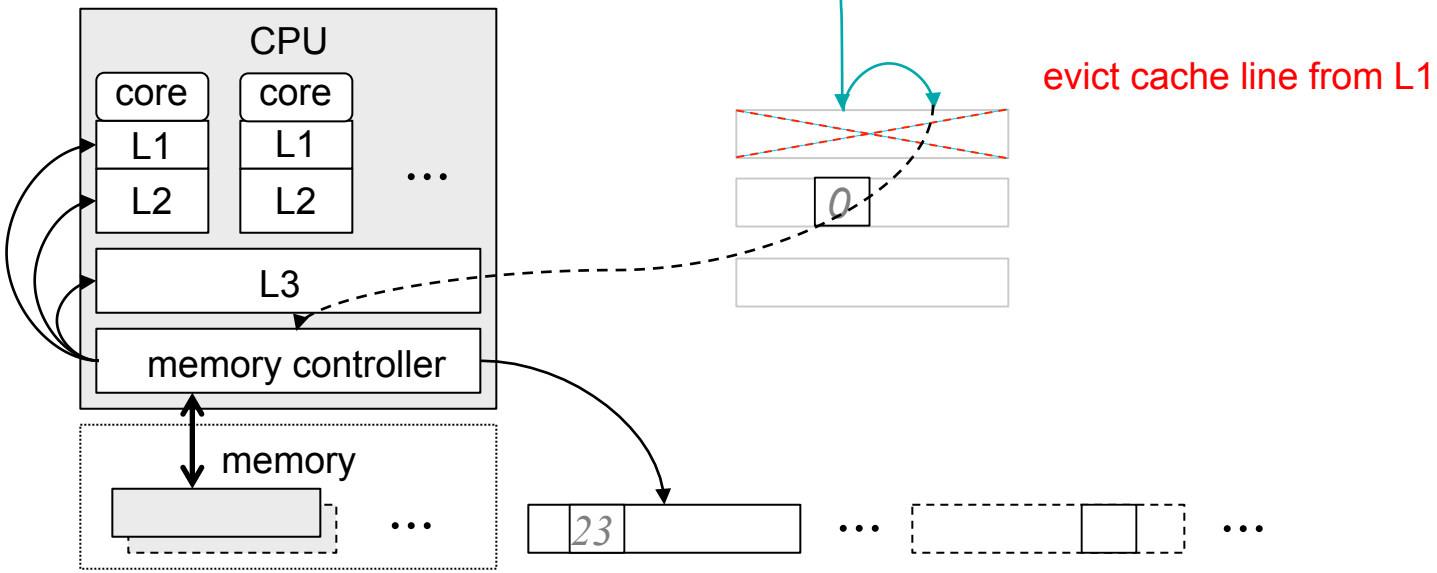
```
for (i=0; i<N; i++) A[rand()%N] = i;
```



Caches – the ugly

- Memory is physically accessed at cache line granularity, e.g. 64Byte
- Writes:

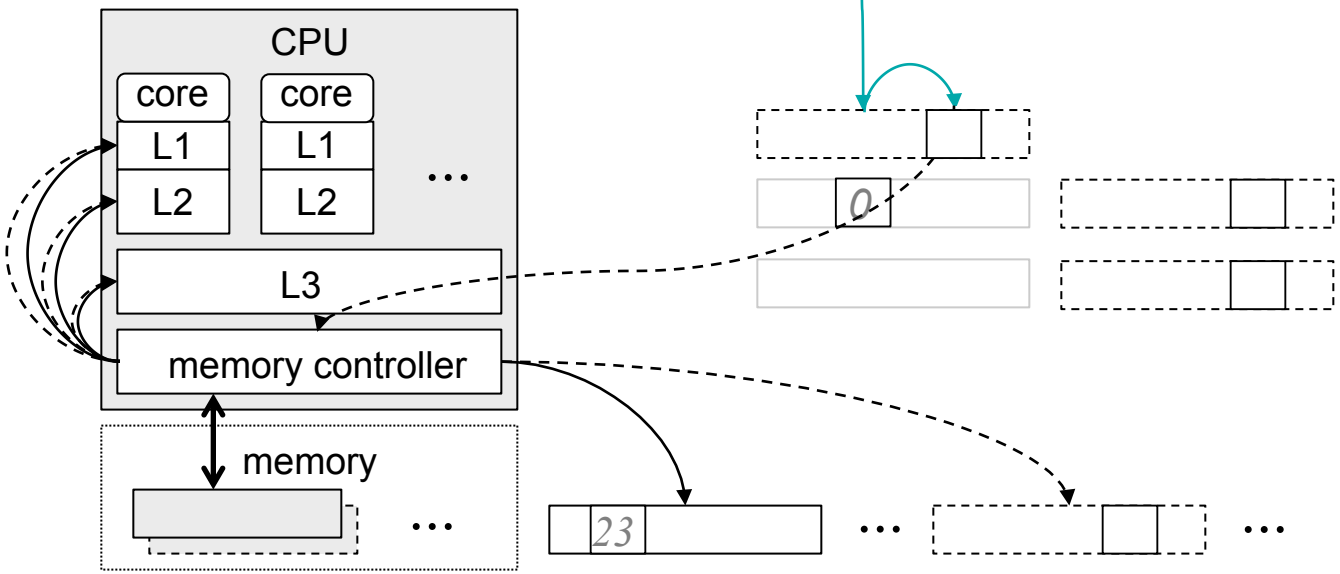
```
for (i=0; i<N; i++) A[rand()%N] = i;
```



Caches – the ugly

- Memory is physically accessed at cache line granularity, e.g. 64Byte
- Writes:

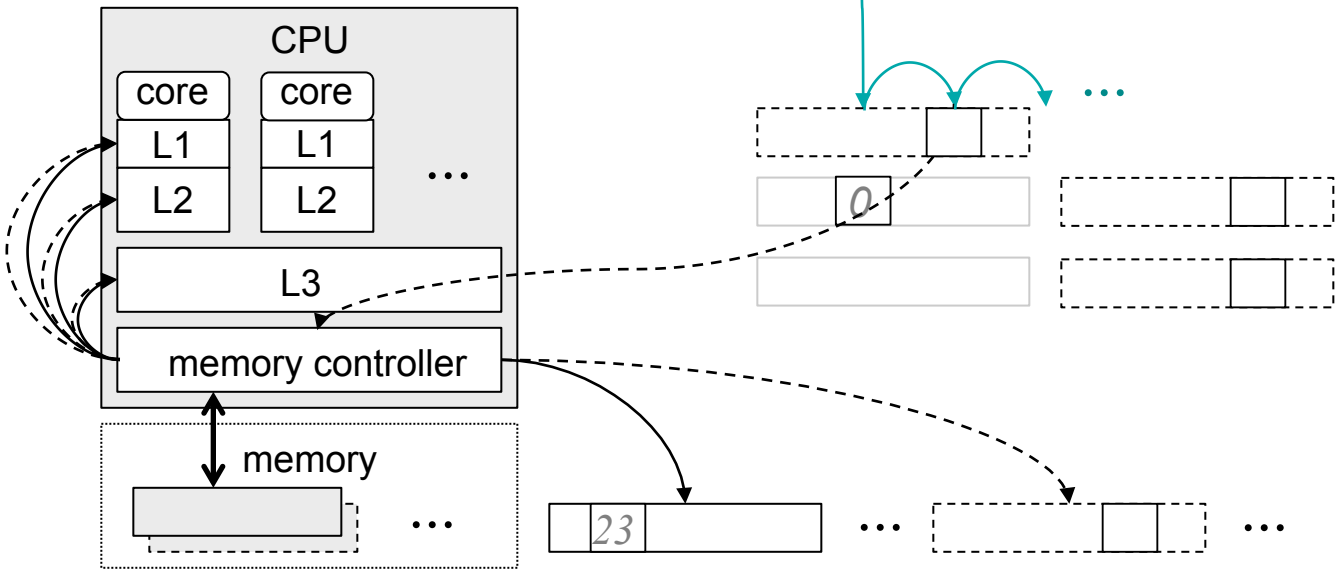
```
for (i=0; i<N; i++) A[rand()%N] = i;
```



Caches – the ugly

- Memory is physically accessed at cache line granularity, e.g. 64Byte
- Writes:

```
for (i=0; i<N; i++) A[rand()%N] = i;
```



- Many memory addresses map into the same cache line(s)
- “Dirty” cache line needs to be evicted before new one loads

➔ Writes effectively turn into read-modify-write, c.f. disks