CSC266 Introduction to Parallel Computing using GPUs
Optimizing for Caches

Sreepathi Pai

October 4, 2017

URCS

Outline

Cache Performance Recap

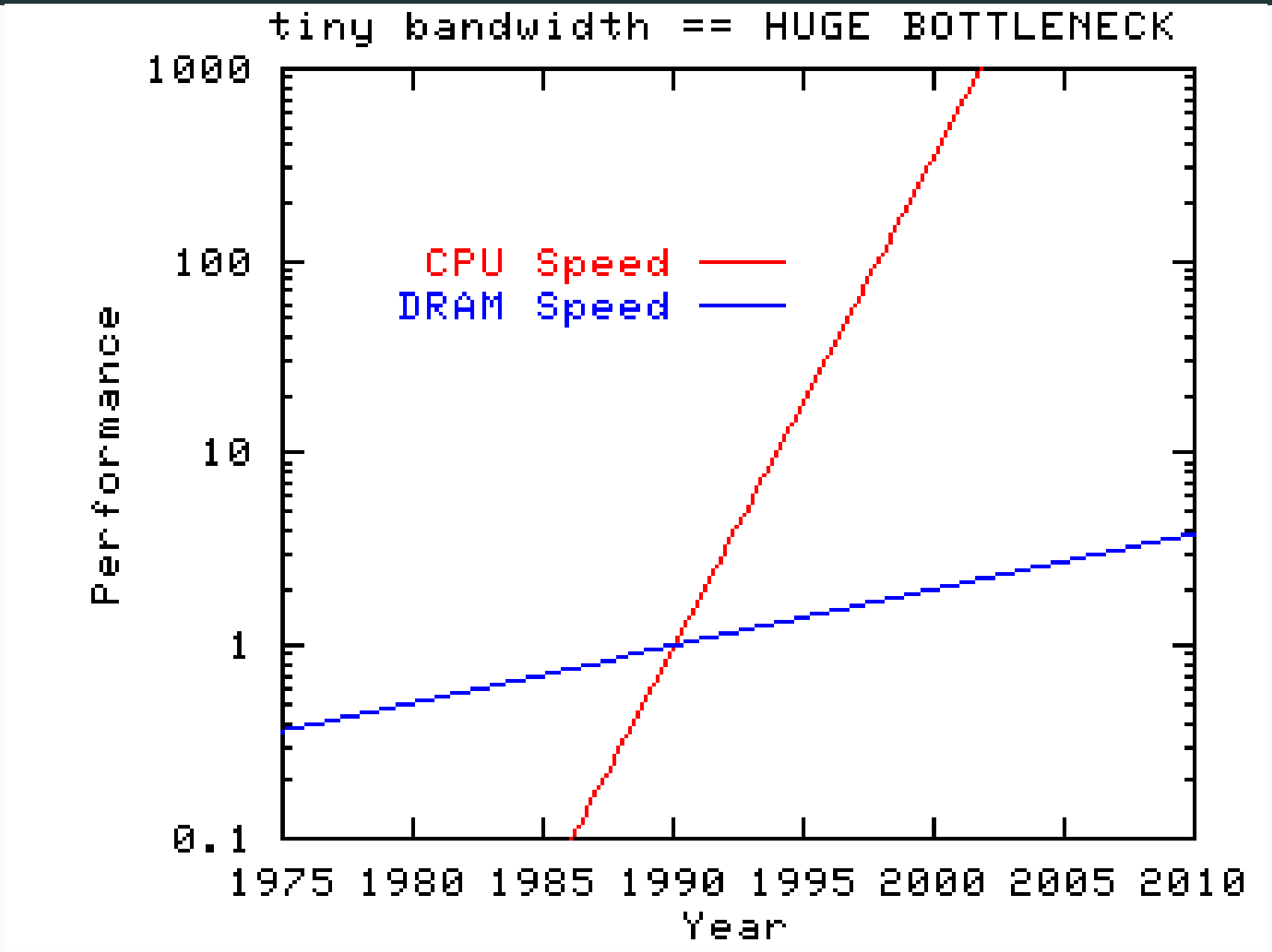
Data Layout

Reuse Distance

Besides the Cache

Outline

Memory vs CPU



from STREAM

Misses (Recap)

- Compulsory/Cold
- Capacity
- Conflict
- Our Goal: Reduce Misses

Outline

Data Layout Preliminaries

- Caches are optimized for:
  - Spatial locality
  - Temporal locality
- Caches store *lines*, 64 to 128 bytes from contiguous locations
  - Memory is partitioned into lines
- Important considerations:
  - Footprint (total size of data)
  - Working set (total size of data accessed in a given period of time)

Data Layout for Arrays

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 & 9 \end{bmatrix}$$

- Row Major
  - C
  - Array elements in same row are next to each other
  - Memory: $\begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{bmatrix}$
- Column Major
  - FORTRAN
  - Array elements in same column are next to each other
  - Memory: $\begin{bmatrix} 0 & 5 & 1 & 6 & 2 & 7 & 3 & 8 & 4 & 9 \end{bmatrix}$

Data Layout for Structures

- What is the size in bytes of each of the structures below?

```
struct pt {
  char ptvalid;

  int x;
  char xvalid;

  int y;
  char yvalid;
} lines[100];

struct pt2 {
  char ptvalid;
  char xvalid;
  char yvalid;

  int x;
  int y;
} lines[100];
```

Packing in Structures

```
struct pt {
  char ptvalid;

  int x;
  char xvalid;

  int y;
  char yvalid;
} lines[100];
```

- Uses 20 bytes per element, padding/packing (3 bytes) inserted after `ptvalid, xvalid, yvalid`

```
struct pt2 {
  char ptvalid;
  char xvalid;
  char yvalid;

  int x;
  int y;
} lines[100];
```

- Uses 12 bytes per element, padding/packing (1 byte) inserted

Dynamic data structures and “Pointer-chasing” code

```
struct node {
   struct node *next;
   int data;
};

n = list->head;
while(n) {
   ...
   n = n->next;
};
```

Dynamic data structures and "Pointer-chasing" code

- Most dynamic data structures (i.e. allocated dynamically)
- Members keeps links (pointers) to other members of the structure
  - Linked lists
  - Trees
- Code to traverse these data structures can hinder some processor optimizations
  - Sometimes called "irregular" code
  - Memory equivalent of indirect/conditional branches

Summary of Data Layout

- Understand how data is laid out in memory
  - Contiguous memory accesses are the fastest (due to spatial locality)
  - Aligned memory accesses are the fastest
- Check for excessive packing
  - May cause poor utilization
- Dynamically allocated structures can perform poorly
  - Use sparingly in high-performance code
  - Consider using custom memory allocators that are backed by contiguous memory

Outline

Reuse Distance (Stack Distance)

Reuse distance for a cache line (or address) is the number of intervening *unique* references to other lines between any two consecutive references to the same line.

$$A \quad B \quad C \quad C \quad D \quad B \quad F \quad A$$

- Reuse distance for second B is 2
- Reuse distance for last A is 4
- Reuse distance for second C is 0
- Reuse distance for others is ∞

Interpreting Reuse Distance

- A miss with reuse distance ∞ implies cold miss
- A miss with reuse distance $<$ cache size (in lines) implies conflict miss
- A miss with reuse distance $>$ cache size (in lines) implies capacity miss
- Note: these assume LRU is the replacement algorithm
- P.S.: these are not *exact rules*
- P.P.S: the only takeway is that changing (specifically, reducing) reuse distance can result in better cache behaviour.

Goal: Reduce misses by decreasing reuse distance

How can you reduce reuse distance?

Register Blocking/Scalarization in SAXPY (before)

- $\mathbf{y} = \alpha\mathbf{x} + \mathbf{y}$

```
 ...
for(int j = 0; j < N; j++) {
   y[i] = y[i] + alpha * x[j];
}
```

Register Blocking/Scalarization in SAXPY (after)

- $\mathbf{y} = \alpha\mathbf{x} + \mathbf{y}$

```
temp = y[i];
for(int j = 0; j < N; j++) {
   temp = temp + alpha * x[j];
}
y[i] = temp;
```

- Automatically by any compiler that can determine that y and x do not alias

Strip-mining

```
min = minimum(y, 0, N);
max = maximum(y, 0, N);
```

- where `minimum` and `maximum` may be implemented as:

```
int minimum(int *a, int start, int end) {

    int tmp = a[start];
    for(int i = start + 1; i < end; i++) {
        if(a[i] < tmp) tmp = a[i];
    }

}
```

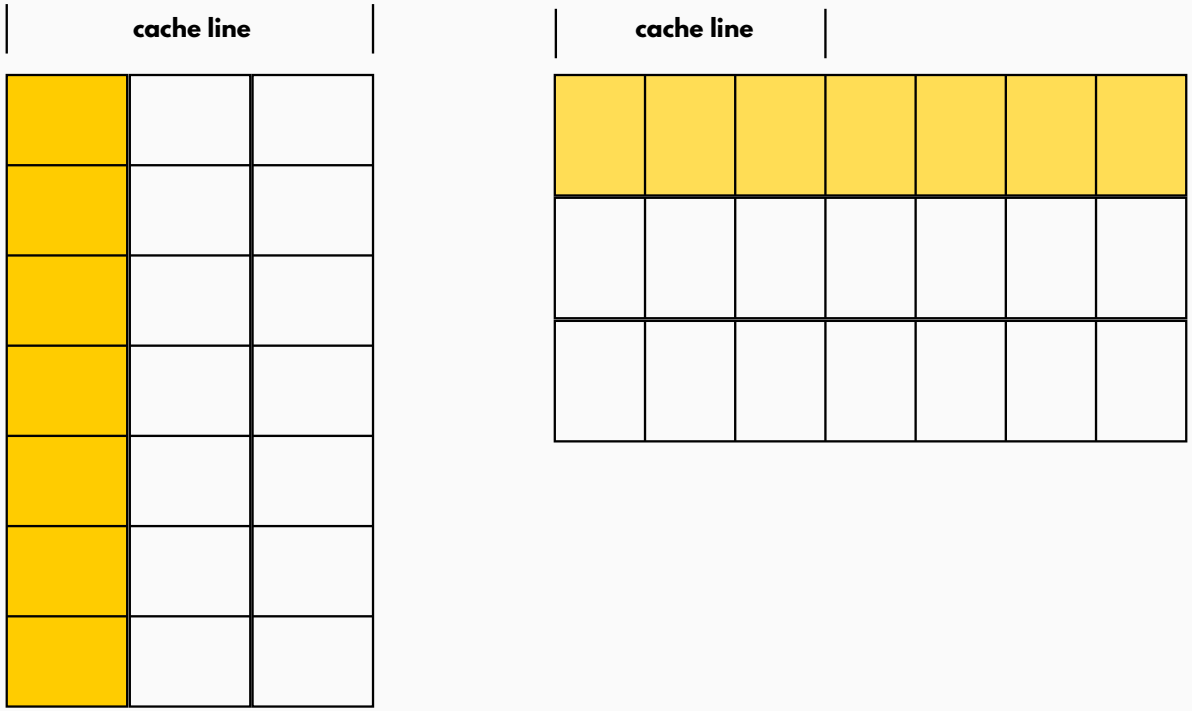
Strip-mining (after)

```
min = y[0];
max = y[0];

for(int j = 0; j < N; j+=WIDTH) {
    end = j + WIDTH;
    if(end > N) end = N;

    tmp_min = minimum(y, j, end);
    tmp_max = maximum(y, j, end);

    if(tmp_min < min) min = tmp_min;
    if(tmp_max > max) max = tmp_max;
}
```

Loop Blocking/Tiling

```
for(int i = 0; i < m; i++)
   for(int j = 0; i < n; j++)
      A_out[j * m + i] = A_in[i * n + j];
```

After one full inner iteration (j)

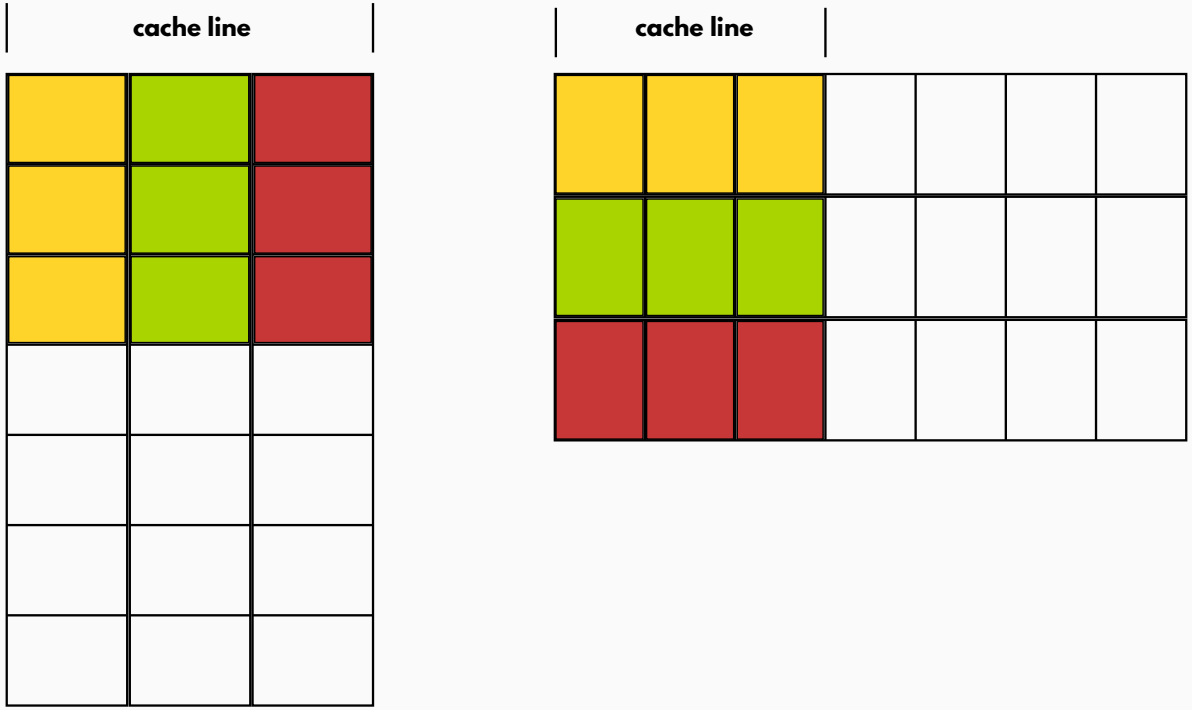


- A_{in} has good cache reuse
- A_{out} has poor cache reuse (reuse distance $= n$)

Loop Blocking (after)

```
for(int i = 0; i < m; i+=WIDTH)
   for(int j = 0; i < n; j+=HEIGHT)

     for(int ii=0; ii < WIDTH && (i+ii) < m; ii++)
         for(int jj=0; jj < HEIGHT && (j+jj) < n; jj++)

             A_out[(j+jj) * m + (i+ii)] = A_in[(i+ii) * n + (j+jj)]
```

After one full tile iteration (*ii* and *jj*)



- A_{in} has good cache reuse
- A_{out} has good cache reuse (reuse distance $= WIDTH$)

Determining blocking parameters

- Which cache must the block fit?
  - L1?
  - L2?
  - LLC?
- How to determine best performing values of WIDTH and HEIGHT?
  - "Auto-tuning"
  - Search through space of all possible values

Other loop transformations to improve cache behaviour

- Loop interchange
  - *IJK* vs *IKJ* (previous lecture)
  - Analyze that loop for reuse distance
- Loop fusion
  - merge two loops together if reuse can be improved
  - check dependencies are not violated!
- Many more
  - need to verify legality when applying
  - check dependencies are not violated

Outline

Other components in the memory subsystem

The following play an important role in the memory subsystem:

- Prefetcher
- Virtual Memory Manager/Pager

The Prefetcher

To avoid compulsory misses, we can prefetch data.

- Software techniques (programmer)
  - issue loads in advance of reading
  - use software prefetch instructions
  - useful when no hardware prefetcher present
- Hardware techniques (automatic)
  - automatically detect a "stream" and fetch from it
  - limited in what it can do
- Fundamental performance concerns
  - Timeliness (is prefetch just-in-time?)
  - Usefulness (was data from prefetch read?)
  - Cache Pollution (prefetches displaces useful lines from cache space)
  - Bandwidth (prefetches compete with "actual" loads)

The Hardware Prefetcher

- "Next line prefetcher" (L1)
  - Fetch next cache line on multiple accesses to a line
- "Stream/Stride prefetcher" (L2/LLC)
  - Triggered by multiple misses
  - Detects stride
  - Fetches cache lines that match the stride
  - Does not fetch past 4096 bytes (x86)

Virtual Memory and Paging

Why did the memory request read from disk?

Virtual Memory

- Every process has its own address space
  - for protection, isolation, etc.
  - Two different processes may store data at the same virtual address
- Data may exist:
  - Physical RAM
  - Disk (Swapped out)
- Virtual to Physical address mappings are handled by the OS
  - stored in page tables
  - which may also be swapped out!
- Every [why?] memory reference by a program must be:
  - translated into its physical equivalent
  - involves reading the page table

Translation Look-aside Buffer (TLB)

- Small cache (what else?)
- Caches page table entries (about 64 or so)
- Fully-associative
- Consulted on every memory request
- Miss in TLB implies walking the page table
  - OSes are lazy, may only create a page on a miss!
- One solution to lower TLB misses: "Huge Pages"
  - 4MB vs 4KB
  - Requires OS and programmer support

Software Managed Caches

- On-chip memory managed by programmer
- Separate address space
- Prevalent on DSPs, GPUs
- Will deal with these when studying GPUs

Conclusion

- Memory accesses will remain primary bottleneck for most applications
  - Big data ...
- Compilers can help with regular, array-based code
  - Especially if you use FORTRAN to write it
  - Situation getting better for C
  - What about other programming languages?
- Irregular, dynamic data-structure based code must be optimized by programmer
  - Few general and automatic solutions
  - Ideas similar to strip-mining/blocking may still be applicable