L3: Memory Hierarchy Optimization I, Locality and Data Placement

Administrative

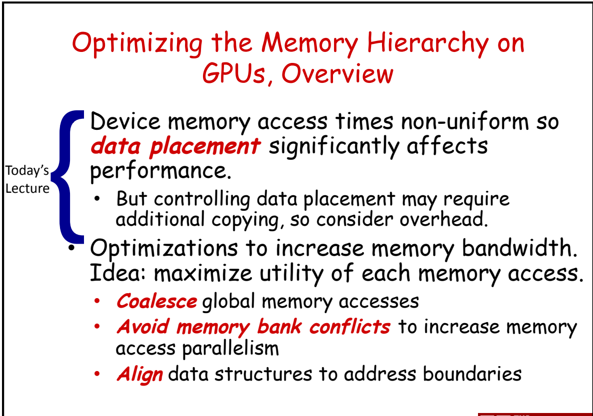
- Assignment due Friday, Jan. 18, 5 PM
  - Use handin program on CADE machines
    - "handin CS6235 lab1 <probfile>"
- Mailing list
  - CS6235@list.eng.utah.edu
    - Please use for all questions suitable for the whole class
    - Feel free to answer your classmates questions!

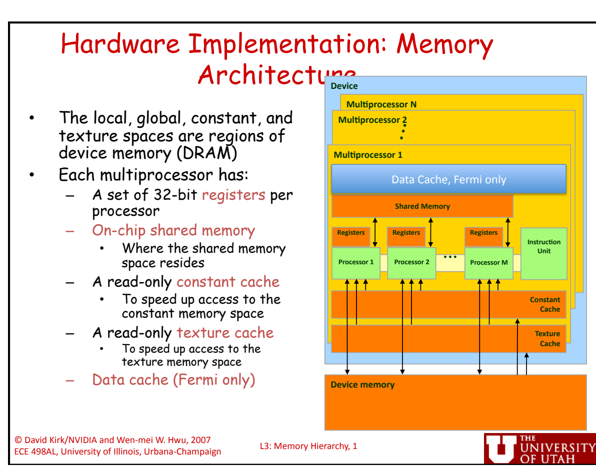
Overview of Lecture

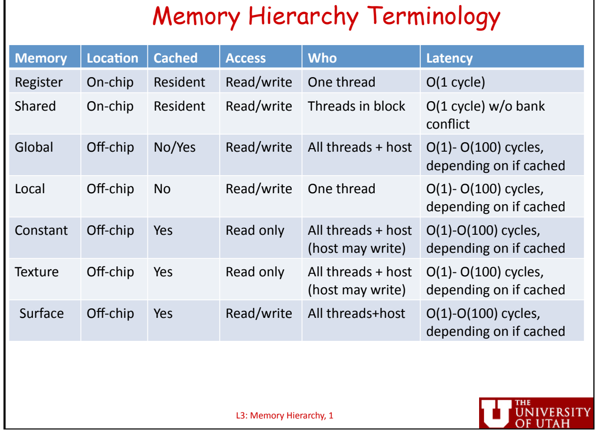
- Where data can be stored
  - And how to get it there
- Some guidelines for where to store data
  - Who needs to access it?
  - Read only vs. Read/Write
  - Footprint of data
- High level description of how to write code to optimize for memory hierarchy
  - More details Wednesday and next week
- Reading:
  - Chapter 5, Kirk and Hwu book
  - Or, http://courses.ece.illinois.edu/ece498/al/textbook/Chapter4-CudaMemoryModel.pdf

Targets of Memory Hierarchy Optimizations

- Reduce ***memory latency***
  - The latency of a memory access is the time (usually in cycles) between a memory request and its completion
- Maximize ***memory bandwidth***
  - Bandwidth is the amount of useful data that can be retrieved over a time interval
- Manage overhead
  - Cost of performing optimization (e.g., copying) should be less than anticipated gain






Memory Hierarchy Terminology

Memory	Location	Cached	Access	Who	Latency
Register	On-chip	Resident	Read/write	One thread	O(1 cycle)
Shared	On-chip	Resident	Read/write	Threads in block	O(1 cycle) w/o bank conflict
Global	Off-chip	No/Yes	Read/write	All threads + host	O(1)- O(100) cycles, depending on if cached
Local	Off-chip	No	Read/write	One thread	O(1)- O(100) cycles, depending on if cached
Constant	Off-chip	Yes	Read only	All threads + host (host may write)	O(1)-O(100) cycles, depending on if cached
Texture	Off-chip	Yes	Read only	All threads + host (host may write)	O(1)- O(100) cycles, depending on if cached
Surface	Off-chip	Yes	Read/write	All threads+host	O(1)-O(100) cycles, depending on if cached

L3: Memory Hierarchy, 1
THE UNIVERSITY OF UTAH

Reuse and Locality

- Consider how data is accessed
  - ***Data reuse:***
    - Same data used multiple times
    - Intrinsic in computation
  - ***Data locality:***
    - Data is reused and is present in "fast memory"
    - Same data or same data transfer
- If a computation has reuse, what can we do to get locality?
  - Appropriate data placement and layout
  - Code reordering transformations

CS6235 L3: Memory Hierarchy, 1 THE UNIVERSITY OF UTAH

Data Placement: Conceptual

- Copies from host to device go to some part of global memory (possibly, constant or texture memory)
- How to use SP shared memory
  - Must construct or be copied from global memory by kernel program
- How to use constant or texture cache
  - Read-only "reused" data can be placed in constant & texture memory by host
- Also, how to use registers
  - Most locally-allocated data is placed directly in registers
  - Even array variables can use registers if compiler understands access patterns
  - Can allocate "superwords" to registers, e.g., float4
  - Excessive use of registers will "spill" data to local memory
- Local memory
  - Deals with capacity limitations of registers
  - Eliminates worries about race conditions
  - ... but SLOW

Data Placement: Syntax

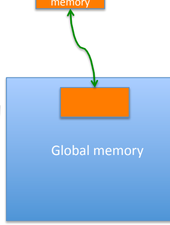
- Through type qualifiers
  - __constant__, __shared__, __local__, __device__
- Through cudaMemcpy calls
  - Flavor of call and symbolic constant designate where to copy
- Implicit default behavior
  - Device memory without qualifier is global memory
  - Host by default copies to global memory
  - Thread-local variables go into registers unless capacity exceeded, then local memory

Rest of Today's Lecture

- Mechanics of how to place data in shared memory and constant memory
- Tiling transformation to reuse data within
  - Shared memory
  - Data cache (Fermi only)
  - Constant memory

Now Let's Look at Shared Memory

- Common Programming Pattern (5.3 of CUDA 4 manual)
  - Load data into shared memory
  - Synchronize (if necessary)
  - Operate on data in shared memory
  - Synchronize (if necessary)
  - Write intermediate results to global memory
  - Repeat until done



Mechanics of Using Shared Memory

- __shared__ type qualifier required
- Must be allocated from global/device function, or as "extern"
- Examples:

```
extern __shared__ float d_s_array[];

/* a form of dynamic allocation */
/* MEMSIZE is size of per-block */
/* shared memory*/
__host__ void outerCompute() {
  compute<<<gs,bs>>>();
}
__global__ void compute() {
   d_s_array[i] = …;
}
```

```
__global__ void compute2() {
  __shared__ float d_s_array[M];

  // create or copy from global memory
  d_s_array[j] = …;
  //synchronize threads before use
  __syncthreads();
  … = d_s_array[x]; // now can use any element
 // more synchronization needed if updated

  // may write result back to global memory
  d_g_array[j] = d_s_array[j];
}
```


Reuse and Locality

- Consider how data is accessed
  - *Data reuse:*
    - Same data used multiple times
    - Intrinsic in computation
  - *Data locality:*
    - Data is reused and is present in "fast memory"
    - Same data or same data transfer
- If a computation has reuse, what can we do to get locality?
  - Appropriate data placement and layout
  - Code reordering transformations

Temporal Reuse in Sequential Code

- Same data used in distinct iterations I and I'

```
for (i=1; i<N-1; i++)
   for (j=1; j<N-1; j++)
      A[j]= A[j]+A[j+1]+A[j-1]
```

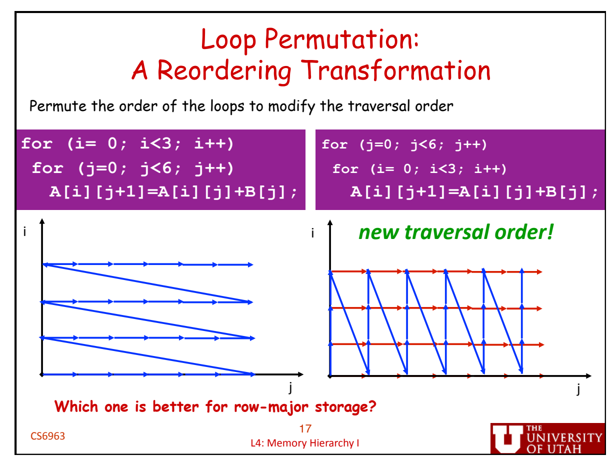
- `A[j]` has self-temporal reuse in loop `i`
- Temporal reuse between A[j] and A[j-1] across iterations I=[i,j] and I'=[i,j+1]

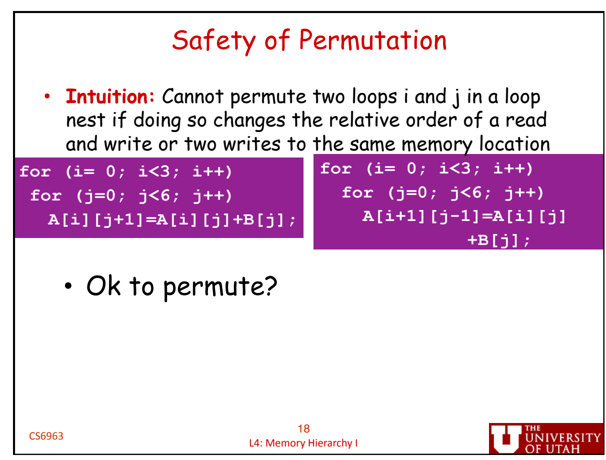
Spatial Reuse

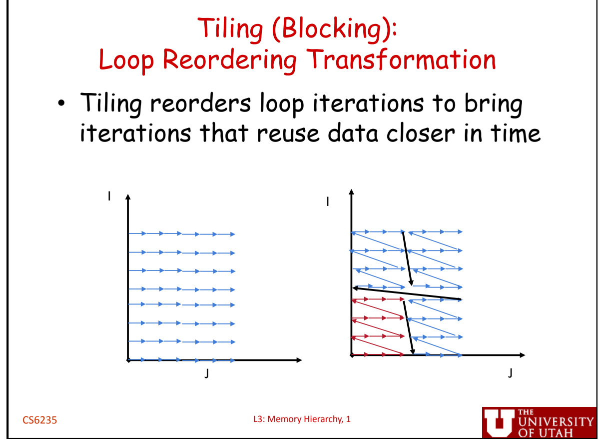
- Same data transfer (usually cache line) used in distinct iterations I and I'

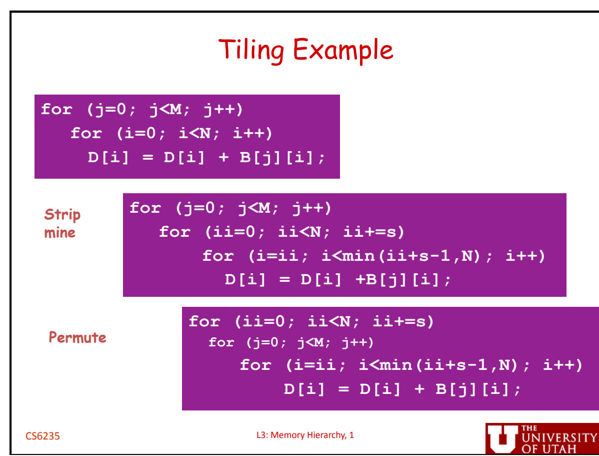
```
for (i=1; i<N; i++)
   for (j=1; j<N; j++)
      A[j]= A[j]+A[j+1]+A[j-1];
```

- `A[j]` has self-spatial reuse in loop `j`
- Spatial reuse between A[j-1], A[j] and A[j+1] in each statement
- **Multi-dimensional array note:** C arrays are stored in row-major order


Loop Permutation:
A Reordering Transformation
Permute the order of the loops to modify the traversal order
for (i= 0; i<3; i++)
for (j=0; j<6; j++)
A[i][j+1]=A[i][j]+B[j];
for (j=0; j<6; j++)
for (i= 0; i<3; i++)
A[i][j+1]=A[i][j]+B[j];
i
j
new traversal order!
Which one is better for row-major storage?
CS6963
17
L4: Memory Hierarchy I
THE UNIVERSITY OF UTAH


Safety of Permutation
• Intuition: Cannot permute two loops i and j in a loop nest if doing so changes the relative order of a read and write or two writes to the same memory location
for (i= 0; i<3; i++)
for (j=0; j<6; j++)
A[i][j+1]=A[i][j]+B[j];
for (i= 0; i<3; i++)
for (j=0; j<6; j++)
A[i+1][j-1]=A[i][j]
+B[j];
• Ok to permute?
CS6963
18
L4: Memory Hierarchy I
THE UNIVERSITY OF UTAH


Tiling (Blocking):
Loop Reordering Transformation
• Tiling reorders loop iterations to bring iterations that reuse data closer in time
I
J
CS6235
L3: Memory Hierarchy, 1
THE UNIVERSITY OF UTAH


Tiling Example
for (j=0; j<M; j++)
for (i=0; i<N; i++)
D[i] = D[i] + B[j][i];
Strip mine
for (j=0; j<M; j++)
for (ii=0; ii<N; ii+=s)
for (i=ii; i<min(ii+s-1,N); i++)
D[i] = D[i] +B[j][i];
Permute
for (ii=0; ii<N; ii+=s)
for (j=0; j<M; j++)
for (i=ii; i<min(ii+s-1,N); i++)
D[i] = D[i] + B[j][i];
CS6235
L3: Memory Hierarchy, 1
THE UNIVERSITY OF UTAH

Legality of Tiling

- Tiling is safe only if it does not change the order in which memory locations are read/written
  - We'll talk about correctness after memory hierarchies
- Tiling can conceptually be used to perform the decomposition into threads and blocks (computation partitioning)
- Tiling is also used to reduce the footprint of data to fit in limited capacity storage (more later)

CUDA Version of Example (Tiling for Computation Partitioning)

```
for (ii=0; ii<N; ii+=s)              <- Block dimension
for (i=ii; i<min(ii+s-1,N); i++)     <- Thread dimension
   for (j=0; j<N; j++)               <- Loop within Thread
      D[i] = D[i] + B[j][i];
```

```
…
<<<ComputeI(N/s,s)>>>(d_D, d_B, N);
…

__global__ ComputeI (float *d_D, float *d_B, int N) {
   int ii = blockIdx.x;
   int i = ii*s + threadIdx.x;
   for (j=0; j<N; j++)
      d_D[i] = d_D[i] + d_B[j*N+i];
   }
```

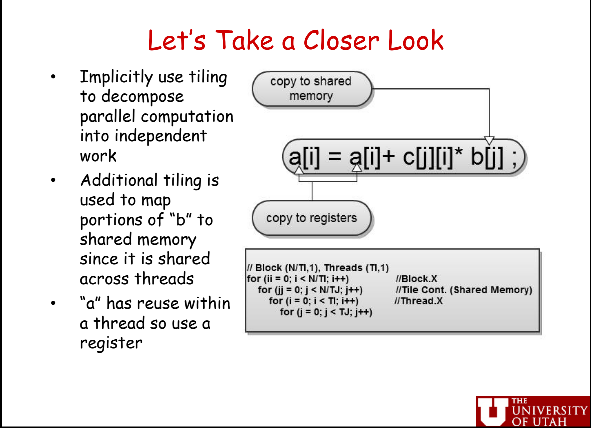

Tiling for Limited Capacity Storage

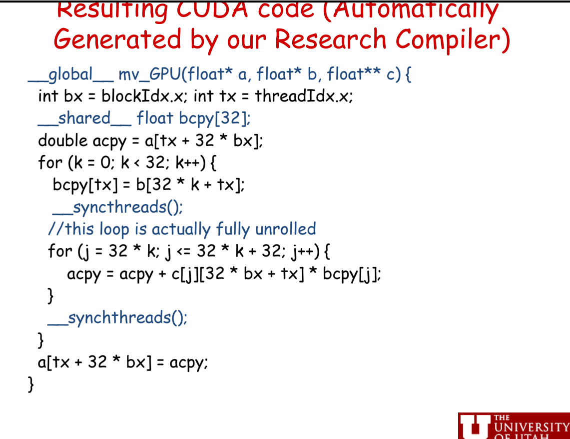
- Tiling can be used hierarchically to compute partial results on a block of data wherever there are capacity limitations
  - Between grids if total data exceeds global memory capacity
  - Across thread blocks if shared data exceeds shared memory capacity (also to partition computation across blocks and threads)
  - Within threads if data in constant cache exceeds cache capacity or data in registers exceeds register capacity or (as in example) data in shared memory for block still exceeds shared memory capacity

Memory Hierarchy Example: Matrix vector multiply

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
     a[i] += c[j][i] * b[j];
  }
}
```

Remember to:
- Consider correctness of parallelization strategy (next week)
- Exploit locality in shared memory and registers





Summary of Lecture

- How to place data in constant memory and shared memory
- Introduction to permutation and tiling transformation
- Matrix-vector multiply example

CS6235 L3: Memory Hierarchy, 1