Administrative Issues

- Next assignment, triangular solve
  - Due 5PM, Friday, March 5
  - handin cs6963 lab 3 <probfile>*
- Project proposals (discussed today)
  - Due 5PM, Wednesday, March 17 (hard deadline)
  - A few minutes at end of class to help form groups

Outline

- Triangular solve assignment
- Project
  - Ideas on how to approach
  - Construct list of questions

Reading:
- Paper link: http://portal.acm.org/citation.cfm?id=1413402
- Talk link: http://www.eecs.berkeley.edu/~volkov/volkov08-sc08talk.pdf

Triangular Solve (STRSM)

for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
        if (B[j*n+k] != 0.0f) {
            for (i = k+1; i < n; i++)
                B[j*n+i] -= A[k * n + i] * B[j * n + k];
        }

Equivalent to:

    cublasStrsm('l' /* left operator */, 'l' /* lower triangular */, 'N' /* not transposed */, 'u' /* unit triangular */, 
    N, N, alpha, d_A, N, d_B, N);

See: http://www.netlib.org/blas/strsm.f
Assignment

- Details:
  - Integrated with simpleCUBLAS test in SDK
  - Reference sequential version provided
1. Rewrite in CUDA
2. Compare performance with CUBLAS 2.0 library

Performance Issues?

- + Abundant data reuse
- - Difficult edge cases
- - Different amounts of work for different \(j,k\) values
- - Complex mapping or load imbalance

Reminder: Outcomes from Last Year's Course

- Paper and poster at Symposium on Application Accelerators for High-Performance Computing
  - Poster:
    - Solving Eigenvalue Equations on Translated Surface Mesh with CUDA
      - Poster #57 - Fu, Zhisong, University of Utah (United States)
  - Poster at NVIDIA Research Summit
    - Poster #47 - Fu, Zhisong, University of Utah (United States)
  - Papers at Industrial Advisory Board meeting
  - Integrated into Masters theses and PhD dissertations
  - Jobs and internships

Projects

- 2-3 person teams
- Select project, or I will guide you
  - From your research
  - From previous classes
  - Suggested ideas from faculty, Nvidia (ask me)
- Example (published):
- Steps
  1. Proposal (due Wednesday, March 17)
  2. Design Review (in class, April 5 and 7)
  3. Poster Presentation (last week of classes)
  4. Final Report (due before finals)
1. Project Proposal (due 3/17)

- Proposal Logistics:
  - Significant implementation, worth 55% of grade
  - Each person turns in the proposal (should be same as other team members)

- Proposal:
  - 3-4 page document (11pt, single-spaced)
  - Submit with handin program:
    "handin cs6963 prop <pdf-file>"

Content of Proposal

I. Team members: Name and a sentence on expertise for each member
II. Problem description
   - What is the computation and why is it important?
   - Abstraction of computation: equations, graphic or pseudo-code, no more than 1 page
III. Suitability for GPU acceleration
   - Amdahl’s Law: describe the inherent parallelism. Argue that it is close to 100% of computation. Use measurements from CPU execution of computation if possible.
   - Synchronization and Communication: Discuss what data structures may need to be protected by synchronization, or communication through host.
   - Copy Overhead: Discuss the data footprint and anticipated cost of copying to/from host memory.
IV. Intellectual Challenges
   - Generally, what makes this computation worthy of a project?
   - Point to any difficulties you anticipate at present in achieving high speedup
Preview: Dense Linear Algebra on GPUs

- SGEMM result is not from algorithm of L5
- Why? Significant reuse can be managed within registers
- Comparison:

<table>
<thead>
<tr>
<th>GPU Rule-of-Thumb</th>
<th>This Lecture (SGEMM)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threading</td>
<td>Generate lots of threads (up to 512/block) to hide memory latency</td>
</tr>
<tr>
<td>Shared memory</td>
<td>Use to exploit reuse across threads</td>
</tr>
<tr>
<td>Registers</td>
<td>Use for temporary per-thread data</td>
</tr>
</tbody>
</table>

SGEMM on Current CPUs and GPUs

(Volkov Slides 3-17, 24)
Latest Research: CUDA-CHiLL

- Automatically generate CUDA for NVIDIA GPU from sequential code plus script
- Abstraction permits parallel threads & staging of data
- Heterogeneous code generation: Alternative scripts generate CUDA, OpenMP or sequential code tuned for memory hierarchy

CUDA-CHiLL: Higher-Level Interface (in progress)

```
init("mm.sp2", "MarkedLoop")
tile_contr_l(\[i\], \[j\], \[T1,TJ\], \[l1_contr="ii", l2_contr="jj"\])
tile_contr_l(\[i\], \[j\], \[l1_contr="kk"\], \[i\], \[j\], \[k\], stripped)
tile_contr_l(\[i\], \[j\], \[l1_contr="ty", l2_tile="tx"\], \[i\], \[j\], \[k\], \[tx\], \[ty\], \[j\], \[k\])

-- Assign loop levels to thread space and name the kernel
cudaim("mm_GPU", \[a=m\], \[b=n\], \[i=n\]), \[a=n\]).

-- Copy the "c" array usage to registers

for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      c[0:15][tx+64*bx+16*ty] += a[16*i][64*bx+16*ty+tx]*P2[0]
                             += a[16*i+1][64*bx+16*ty+tx]*P2[1]
                             += a[16*i+2][64*bx+16*ty+tx]*P2[2]
                             += a[16*i+3][64*bx+16*ty+tx]*P2[3]
                             += a[16*i+4][64*bx+16*ty+tx]*P2[4]
                             += a[16*i+5][64*bx+16*ty+tx]*P2[5]
                             += a[16*i+6][64*bx+16*ty+tx]*P2[6]
                             += a[16*i+7][64*bx+16*ty+tx]*P2[7]
                             += a[16*i+8][64*bx+16*ty+tx]*P2[8]
                             += a[16*i+9][64*bx+16*ty+tx]*P2[9]
                             += a[16*i+10][64*bx+16*ty+tx]*P2[10]
                             += a[16*i+11][64*bx+16*ty+tx]*P2[11]
                             += a[16*i+12][64*bx+16*ty+tx]*P2[12]
                             += a[16*i+13][64*bx+16*ty+tx]*P2[13]
                             += a[16*i+14][64*bx+16*ty+tx]*P2[14]
                             += a[16*i+15][64*bx+16*ty+tx]*P2[15]

-- Automatically-generated CUDA
```

CUDA-CHiLL: Automatically-Generated Code

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      c[0:15][tx+64*bx+16*ty] += a[16*i][64*bx+16*ty+tx]*P2[0]
                             += a[16*i+1][64*bx+16*ty+tx]*P2[1]
                             += a[16*i+2][64*bx+16*ty+tx]*P2[2]
                             += a[16*i+3][64*bx+16*ty+tx]*P2[3]
                             += a[16*i+4][64*bx+16*ty+tx]*P2[4]
                             += a[16*i+5][64*bx+16*ty+tx]*P2[5]
                             += a[16*i+6][64*bx+16*ty+tx]*P2[6]
                             += a[16*i+7][64*bx+16*ty+tx]*P2[7]
                             += a[16*i+8][64*bx+16*ty+tx]*P2[8]
                             += a[16*i+9][64*bx+16*ty+tx]*P2[9]
                             += a[16*i+10][64*bx+16*ty+tx]*P2[10]
                             += a[16*i+11][64*bx+16*ty+tx]*P2[11]
                             += a[16*i+12][64*bx+16*ty+tx]*P2[12]
                             += a[16*i+13][64*bx+16*ty+tx]*P2[13]
                             += a[16*i+14][64*bx+16*ty+tx]*P2[14]
                             += a[16*i+15][64*bx+16*ty+tx]*P2[15]

```

Final Result: Copy C to registers, B to shared memory and unroll

Steps 1 through 4 tile (for computation, data)
1. Copy the "c" array usage to registers
2. tile_contr_l(\[i\],[j], \[T1,TJ\])
3. tile_contr_l(\[i\],[j], \[l1_contr="ty", l2_tile="tx"\])
4. unroll_to_depth(2)

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < n; k++)
      c[0:15][tx+64*bx+16*ty] += a[16*i][64*bx+16*ty+tx]*P2[0]
                             += a[16*i+1][64*bx+16*ty+tx]*P2[1]
                             += a[16*i+2][64*bx+16*ty+tx]*P2[2]
                             += a[16*i+3][64*bx+16*ty+tx]*P2[3]
                             += a[16*i+4][64*bx+16*ty+tx]*P2[4]
                             += a[16*i+5][64*bx+16*ty+tx]*P2[5]
                             += a[16*i+6][64*bx+16*ty+tx]*P2[6]
                             += a[16*i+7][64*bx+16*ty+tx]*P2[7]
                             += a[16*i+8][64*bx+16*ty+tx]*P2[8]
                             += a[16*i+9][64*bx+16*ty+tx]*P2[9]
                             += a[16*i+10][64*bx+16*ty+tx]*P2[10]
                             += a[16*i+11][64*bx+16*ty+tx]*P2[11]
                             += a[16*i+12][64*bx+16*ty+tx]*P2[12]
                             += a[16*i+13][64*bx+16*ty+tx]*P2[13]
                             += a[16*i+14][64*bx+16*ty+tx]*P2[14]
                             += a[16*i+15][64*bx+16*ty+tx]*P2[15]
```

```
float P1[16];
shared float P2[16][17];
be = blockIdx.x, ty = threadIdx.y,
   tx = threadIdx.x, bx = blockIdx.y;
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < N; k++)
      c[0:15][tx+64*bx+16*ty] += a[16*i][64*bx+16*ty+tx]*P2[0]
                             += a[16*i+1][64*bx+16*ty+tx]*P2[1]
                             += a[16*i+2][64*bx+16*ty+tx]*P2[2]
                             += a[16*i+3][64*bx+16*ty+tx]*P2[3]
                             += a[16*i+4][64*bx+16*ty+tx]*P2[4]
                             += a[16*i+5][64*bx+16*ty+tx]*P2[5]
                             += a[16*i+6][64*bx+16*ty+tx]*P2[6]
                             += a[16*i+7][64*bx+16*ty+tx]*P2[7]
                             += a[16*i+8][64*bx+16*ty+tx]*P2[8]
                             += a[16*i+9][64*bx+16*ty+tx]*P2[9]
                             += a[16*i+10][64*bx+16*ty+tx]*P2[10]
                             += a[16*i+11][64*bx+16*ty+tx]*P2[11]
                             += a[16*i+12][64*bx+16*ty+tx]*P2[12]
                             += a[16*i+13][64*bx+16*ty+tx]*P2[13]
                             += a[16*i+14][64*bx+16*ty+tx]*P2[14]
                             += a[16*i+15][64*bx+16*ty+tx]*P2[15]
```

```
init("mm.sp2", "MarkedLoop")
tile_contr_l(\[i\],[j], \[T1,TJ\], \[l1_contr="ii", l2_contr="jj"\])
tile_contr_l(\[i\],[j], \[l1_contr="kk"\], \[i\], \[j\], \[k\], stripped)
tile_contr_l(\[i\],[j], \[l1_contr="ty", l2_tile="tx"\], \[i\], \[j\], \[k\], \[tx\], \[ty\], \[j\], \[k\])
```

Gabe Rudy Master’s thesis
Next Class

• See Khan/Rudy poster on Friday!
• More complex dense linear algebra
• Some specific global synchronization strategies to support these