L10: Dense Linear Algebra on GPUs

Administrative Issues

- Next assignment, linear algebra
  - Handed out by Friday
  - Due before spring break
  - handin CS6235 lab 3 <probfile>*

Outline

- Triangular solve assignment from last year (briefly)

Reading:
Benchmarking GPUs to tune dense linear algebra, SC08, November 2008.
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
Last Year's Assignment

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

Symmetric Matrix Multiply (SSYMM)

```c
float a[N][N], b[N][N], c[N][N];
float t1, t2;
for (j = 0; j < N; j++) {
  for (i = 0; i < N; i++) {
    t1 = b[j][i];
    t2 = 0;
    for (k = 0; k < i - 1; k++) {
      c[j][k] += t1 * a[i][k];
      t2 += b[k][j] * a[i][k];
    }
    c[j][i] += t1 * a[i][i] + t2;
  }
}
```

See: [http://www.netlib.org/blas/ssymm.f](http://www.netlib.org/blas/ssymm.f)

Equivalent to:

```c
cublasSsym('l' /* left operator */, 'u' /* upper triangular */, N, N, 1.0, d_A, N, d_B, N, 1.0, d_C, N);
```

Performance Issues?

- + Abundant data reuse
- - Difficult edge cases
- - Different amounts of work for different <j,k> values
- - Complex mapping or load imbalance
Tiled Matrix Multiply Using Thread Blocks

- One block computes one square sub-matrix $P_{sub}$ of size BLOCK_SIZE
- One thread computes one element of $P_{sub}$
- Assume that the dimensions of M and N are multiples of BLOCK_SIZE and square shape

Recall this code from L4

```
for (int ii = 0; ii < Width/TI; ii++)
    for (int jj = 0; jj < Width/TJ; jj++)
        for (int k = 0; k < kk; k += TK)
            sum += M[(ii*TI+i)*k] * N[k][jj*TJ+j];

P[(ii*TI+i)*jj*TJ+j] = sum;
```

Why? Significant reuse can be managed within registers

Preview: SGEMM (CUBLAS 2.x/3.x) on GPUs

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

<table>
<thead>
<tr>
<th>GPU Rule-of-Thumb</th>
<th>Lecture (for GTX 280)</th>
<th>Lecture (for Fermi C2050)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threading</td>
<td>Generate lots of threads (up to 512/block) to hide memory latency</td>
<td>Only 64 threads/block provides 2 warps, sufficient to hide latency plus conserves registers</td>
</tr>
<tr>
<td>Shared memory</td>
<td>Use to exploit reuse across threads</td>
<td>Communicate shared data across threads and coalesce global data</td>
</tr>
<tr>
<td>Registers</td>
<td>Use for temporary per-thread data</td>
<td>Exploit significant reuse within a thread</td>
</tr>
<tr>
<td>Texture memory</td>
<td>Not used</td>
<td>Not used</td>
</tr>
</tbody>
</table>
Comparison with MKL (Intel)

Volkov Slides 5-17, 24

CHiLL and CUDA-CHiLL: Compiler-Based Auto-Tuning Technology

- Increase compiler effectiveness through auto-tuning and specialization
- Provide high-level interface to code generation (recipe) for library or application developer to suggest optimization
- Bridge domain decomposition and single-socket locality and parallelism optimization
- Auto-tuning for different optimization goals: performance, energy, reliability

Decision Algorithm: Computational Decomposition

- Block Parallelism
- Thread Parallelism

Decision Algorithm: Data Staging

- Data Staging
- Shared Memory
- Registers
- Cudaize
- Unrolling

`tile_by_index(\{j\}, \{TJ\}, \{l1_control=\"jj\}\}; \{ii, jj, i, j\})`

Data Reuse
- inside thread
- across threads

Final Loop Order

CUDA-CHiLL Recipe

`N = 1024`
`T = TJ = 32`
`tile_by_index(\{i\}, \{TI, TJ\}, \{l1_control=\"ii\}\}; \{ii, jj, i, j\})`
`normalize_index(\"i\")`
`cudaize(\"mv_GPU\", \{a=N, b=N, c=N*N\}; \{block=\"ii\", thread=\"jj\"\})`
`copy_to_shared(\"tx\", \"b\", 1)`
`copy_to_registers(\"p\", \"a\")`
`unroll_to_depth(1)`

An added Complication
- What if input matrix \(C\) is transposed?
- What happens to global memory coalescing?
- What can be done?
CUDA-ChiLL for Matrix Multiply (CUBLAS 2.x version)

```
init("mm.sp2", "MarkedLoop")
tile_by_index({"i", "j"}, {TI, TJ},
  {l1_control="ii", l2_control="jj"},
  {"ii", "jj", "i", "j"})
tile_by_index({"k"}, {TK},
  {l1_control="kk"},
  {"ii", "jj", "kk", "i", "j", "k"}, strided)
tile_by_index({"i"}, {TJ},
  {l1_control="ty", l1_tile="tx"},
  {"ii", "jj", "kk", "tx", "ty", "j", "k"})

-- Assign loop levels to thread space and name the kernel
cudaize("mm_GPU",
  {a=N*N, b=N*N, c=N*N},
  -- array sizes for data copying
  {block={"ii", "jj"}, thread={"tx", "ty"}})

-- Copy the "c" array usage to registers
6. copy_to_registers("kk", "c")
5. copy_to_shared("ty", "b")

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

7. unroll_to_depth(2)
```

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

CUDA-ChiLL for Matrix Multiply (CUBLAS 2.x version)

```
float P1[16];
__shared__ float P2[16][17];
bx = blockIdx.x,
by = blockIdx.y;

for (t6 = 0; t10 <= 1008; t6+=16)
  {
    P2[tx][4*ty:4*ty+3] = b[16*by+4*ty:16*by+4*ty+3][tx+t6];
    __syncthreads();
    P1[0:15] += a[t6][64*bx+16*ty+tx]*P2[0][0:15]
    P1[0:15] += a[t6+1][64*bx+16*ty+tx]*P2[1][0:15]
    ...
    P1[0:15] += a[t6+15][64*bx+16*ty+tx]*P2[15][0:15]
    __syncthreads();
  }
for (i = 0; i < n; i++)
for (j = 0; j < n; j++)
c[16*by:16*by+15][tx+64*bx+16*ty] = P1[0:15];
```

Direct Comparison: Automatically-Generated Matrix-Matrix Multiply Scripts

GTX-280 implementation
Mostly corresponds to CUBLAS 2.x and Volkov's SC08 paper

```
1. by_index({"i", "j"}, {TI, TJ},
  {l1_control="ii", l2_control="jj"},
  {"i", "j"})
2. by_index({"k"}, {TK},
  {l1_control="kk"},
  {"kk", "i", "j", "k"})
3. by_index({"i"}, {TJ},
  {l1_control="ty", l1_tile="tx"},
  {"i", "j", "k", "tx", "ty", "j", "k"})

Different computation decomposition
a is shared memory; both a and b are read through texture memory
```

```
1. by_index({"i", "j"}, {TI, TJ},
  {l1_control="ii", l2_control="jj"},
  {"i", "j"})
2. by_index({"k"}, {TK},
  {l1_control="kk"},
  {"kk", "i", "j", "k"})
3. by_index({"i"}, {TJ},
  {l1_control="ty", l1_tile="tx"},
  {"i", "j", "k", "tx", "ty", "j", "k"})

Different computation decomposition
a is shared memory, both a and b are read through texture memory
```

Direct Comparison: Automatically-Generated Matrix-Matrix Multiply Scripts

GTX-280 implementation
Mostly corresponds to CUBLAS 2.x and Volkov's SC08 paper

```
1. by_index({"i", "j"}, {TI, TJ},
  {l1_control="ii", l2_control="jj"},
  {"i", "j"})
2. by_index({"k"}, {TK},
  {l1_control="kk"},
  {"kk", "i", "j", "k"})
3. by_index({"i"}, {TJ},
  {l1_control="ty", l1_tile="tx"},
  {"i", "j", "k", "tx", "ty", "j", "k"})

Different computation decomposition
a is shared memory; both a and b are read through texture memory
```

```
1. by_index({"i", "j"}, {TI, TJ},
  {l1_control="ii", l2_control="jj"},
  {"i", "j"})
2. by_index({"k"}, {TK},
  {l1_control="kk"},
  {"kk", "i", "j", "k"})
3. by_index({"i"}, {TJ},
  {l1_control="ty", l1_tile="tx"},
  {"i", "j", "k", "tx", "ty", "j", "k"})

Different computation decomposition
a is shared memory, both a and b are read through texture memory
```

BLAS Library Kernel Optimization

```
MV Results
MM Results
VV Results
TMV Results
```

```
BLAS Library Kernel Optimization

```
MV Results
MM Results
VV Results
TMV Results
```

```
BLAS Library Kernel Optimization

```
MV Results
MM Results
VV Results
TMV Results
```

```
BLAS Library Kernel Optimization

```
MV Results
MM Results
VV Results
TMV Results
```

```
BLAS Library Kernel Optimization

```
MV Results
MM Results
VV Results
TMV Results
```

```
```
MV Results
MM Results
VV Results
TMV Results
```
Next Class

• Pascal:
  – Sparse linear algebra
  – Sparse graph algorithms
• Appropriate data representations