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:

Paper: Volkov, V., and Demmel, J. W. 2008. 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

Volkov code: http://forums.nvidia.com/index.php?showtopic=47689&st=40&p=314014&#entry314014

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)

```
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 = t2 + b[k][j]*a[i][k];
    }
    c([j][i] += t1*a[i][i] + t2;
  }
}
```

Equivalent to:

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

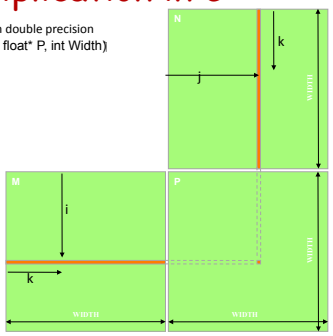
See: http://www.netlib.org/blas/ssymm.f

Performance Issues?

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

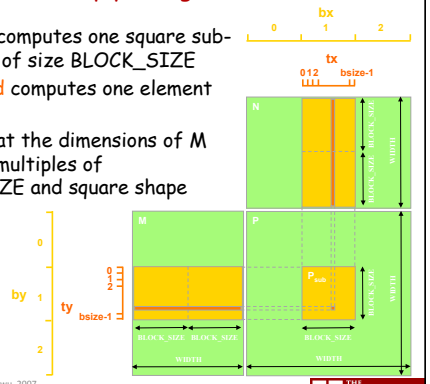
Reminder from L5: Matrix Multiplication in C

```
// Matrix multiplication on the (CPU) host in double precision
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
{
   for (int i = 0; i < Width; ++i)
      for (int j = 0; j < Width; ++j) {
         double sum = 0;
         for (int k = 0; k < Width; ++k) {
            double a = M[i * width + k];
            double b = N[k * width + j];
            sum += a * b;
         }
         P[i * Width + j] = sum;
      }
}
```



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



THE UNIVERSITY OF UTAH

Recall this code from L4

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

Block dimensions – must be unit stride

Thread dimensions – must be unit stride

Tiling for shared memory, no need to change

THE UNIVERSITY OF UTAH

Final Code (from text, p. 87)

```
__global__ void MatrixMulKernel (float *Md, float *Nd, float *Pd, int Width) {
1.    __shared__ float Mds [TILE_WIDTH] [TILE_WIDTH];
2.    __shared__ float Nds [TILE_WIDTH] [TILE_WIDTH];
3 & 4.    int bx = blockIdx.x; int by = blockIdx.y;  int tx = threadIdx.x; int ty = threadIdx.y;
//Identify the row and column of the Pd element to work on
5 & 6.    int Row = by * TILE_WIDTH + ty;    int Col = bx * TILE_WIDTH + tx;
7.        float Pvalue = 0;
// Loop over the Md and Nd tiles required to compute the Pd element
8.        for (int m=0; m < Width / TILE_WIDTH; ++m) {
// Collaborative (parallel) loading of Md and Nd tiles into shared memory
9.          Mds [ty] [tx] = Md [Row*Width + (m*TILE_WIDTH + tx)];
10.         Nds [ty] [tx] = Nd [(m*TILE_WIDTH + ty)*Width + Col];
11.         __syncthreads();                    // make sure all threads have completed copy before calculation
12.         for (int k = 0; k < TILE_WIDTH; ++k)    // Update Pvalue for TKxTK tiles in Mds and Nds
13.           Pvalue += Mds [ty] [k] * Nds [k] [tx];
14.         __syncthreads();                    // make sure calculation complete before copying next tile
          } // m loop
15.       Pd [Row*Width + Col] =  Pvalue;
}
```


THE UNIVERSITY OF UTAH

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

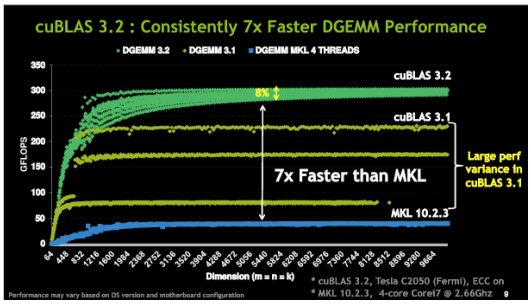
- SGEMM result is not from algorithm of L5
- Why? Significant reuse can be managed within registers

	GPU Rule-of-Thumb	Lecture (for GTX 280)	Lecture (for Fermi C2050
Threading	Generate lots of threads (up to 512/ block) to hide memory latency	Only 64 threads/block provides 2 warps, sufficient to hide latency plus conserves registers	More cores/block, fewer registers/thread, so use 96 threads/block
Shared memory	Use to exploit reuse across threads	Communicate shared data across threads and coalesce global data	Communicate shared data across threads and coalesce global data
Registers	Use for temporary per-thread data	Exploit significant reuse within a thread	Exploit significant reuse within a thread
Texture memory	Not used	Not used	Increase bandwidth for global memory through parallel accesses

THE UNIVERSITY OF UTAH

Volkov Slides 5-17, 24

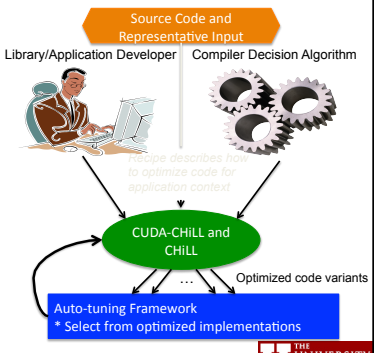
Comparison with MKL (Intel)



Slide source: http://www.scribd.com/doc/47501296/CUDA-3-2-Math-Libraries-Performance

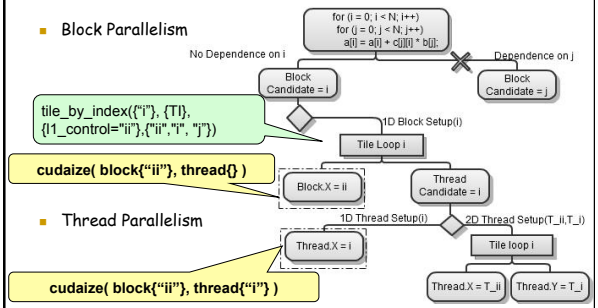
CHiLL and CUDA-CHiLL: Compiler-Based Auto-Tuning Technology

- 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

copy to shared memory

Data Reuse across threads

a[i] = a[i]+ c[j][i]* b[j] ;

copy to registers

Data Reuse inside thread

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

Final Loop Order

```
// Block (N/TI,1), Threads (TI,1)
for (ii = 0; i < N/TI; i++)          //Block.X
  for (jj = 0; j < N/TJ; j++)        //Tile Cont. (Shared Memory)
    for (i = 0; i < TI; i++)         //Thread.X
      for (j = 0; j < TJ; j++)
```

cudaize(block{"ii"}, thread{"i"})

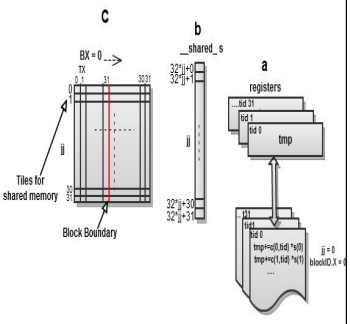
unroll to depth(1)

CUDA-CHiLL Recipe

```
N = 1024
TI= TJ = 32
tile_by_index({"i","j"}, {TI,TJ},
    {l1_control="ii",
    l2_control="k"},{"ii", "jj","i",
    "j"})
normalize_index("i")

cudaize("mv_GPU", {a=N,
    b=N, c=N*N},{block={"ii"},
    thread={"i"}})

copy_to_shared("tx", "b", 1)
copy_to_registers("jj", "a")
unroll_to_depth(1)
```



Matrix-Vector Multiply: GPU Code

Generated Code: with Computational decomposition only.

```
__global__ GPU_MV(float* a, float* b, float** c) {
int bx = blockIdx.x; int tx = threadIdx.x;
int i = 32*bx+tx;
for (j = 0; j < N; j++)
        a[i] = a[i] + c[j][i] * b[j];
}
```

Final MV Generated Code: with Data staged in shared memory & registers.

```
__global__ GPU_MV(float* a, float* b, float** c) {
int bx = blockIdx.x; int tx = threadIdx.x;
__shared__ float bcpy[32];
float acpy = a[tx + 32 * bx];
for (jj = 0; jj < 32; jj++) {
    bcpy[tx] = b[32 * jj + tx];
    __syncthreads();
   //this loop is actually fully unrolled
   for (j = 32 * jj; j <= 32 * jj + 32; j++)
      acpy = acpy + c[j][32 * bx + tx] * bcpy [j];
    __syncthreads();
}
a[tx + 32 * bx] = acpy; }
```

Performance outperforms CUBLAS 3.2 (see later slide)

An added Complication

- What if input matrix C is transposed?
  a[i] += c[i][j] * b[j];
- What happens to global memory coalescing?
- What can be done?

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

```
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];
```

```
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
copy_to_registers("kk", "c", {"tx","ty"})
copy_to_shared("ty", "b")
--Unroll two innermost loop levels fully
unroll_to_depth(2)
```

Gabe Rudy Master's thesis

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

```
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];
```

```
Steps 1 through 4 tile (for computation, data)
--Copy the "c" array usage to registers
5. copy_to_registers("kk", "c", {"tx","ty"})
6. copy_to_shared("ty", "b")
--Unroll two innermost loop levels fully
7. unroll_to_depth(2)
```

B goes into shared memory

C goes into registers and is copied back at end

```
float P1[16];
__shared__ float P2[16][17];
bx = blockIdx.x, by = blockIdx.y;
tx = threadIdx.x, ty = threadIdx.y;
P1[0:15] = c[16*by:16*by+15][tx+64*bx+16*ty];
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();
}
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 tile_by_index({"i","j"},{TI,TJ},{l1_control="ii",l2_control="jj"},
                {"ii","jj","i","j"})
2 tile_by_index({"k"},{TK},{l1_control="kk"},
                {"ii","jj","kk","i","j","k"},strided)
3 tile_by_index({"i"},{TJ},{l1_control="tt",l1_tile="tt"},
                {"ii","jj","kk","tt","t","j","k"})
4 cudaize("mm_GPU",{a=N*N,b=N*N,c=N*N},
          {block={"ii","jj"},thread={"tt","ss"}})
5 copy_to_shared("tx","b",-16)
6 copy_to_registers("kk","c")
7 copy_to_texture("b")
8 unroll_to_depth(2)
```

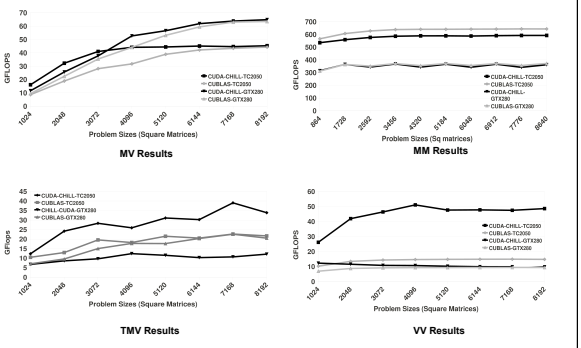
```
1  tile_by_index({"i","j"},{TI,TJ},{l1_control="ii",l2_control="jj"},
                 {"ii","jj","i","j"})
2  tile_by_index({"k"},{TK},{l1_control="kk"},
                 {"ii","jj","kk","i","j","k"},strided)
3  tile_by_index({"i"},{TK},{l1_control="t",l1_tile="tt"},
                 {"ii","jj","kk","tt","t","j","k"})
4  tile_by_index({"j"},{TK},{l1_control="s",l1_tile="ss"},
                 {"ii","jj","kk","tt","t","ss","s","k"})
5  cudaize("mm_GPU",{a=N*N,b=N*N,c=N*N},
           {block={"ii","jj"},thread={"tt","ss"}})
6  copy_to_shared("tx","b",-16)
7  copy_to_texture("b")
8  copy_to_shared("tx","a",-16)
9  copy_to_texture("a")
10 copy_to_registers("kk","c")
11 unroll_to_depth(2)
```

TC2050 Fermi implementation
Mostly corresponds to CUBLAS 3.2 and MAGMA

Different computation decomposition leads to additional tile command

a in shared memory, both a and b are read through texture memory

BLAS Library Kernel Optimization



Next Class

- Pascal:
  - Sparse linear algebra
  - Sparse graph algorithms
- Appropriate data representations