L11: Dense Linear Algebra on GPUs, cont.

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)
  - handin cs6963 prop <pdffile>
  - A few minutes at end of class to help form groups

Outline

- Project
  - Suggested topics
- Triangular solve assignment
- More on dense linear algebra
- LU

Project Ideas

- Suggestions from Nvidia (Nathan Bell via Pete Shirley, Steve Parker):
  - Sparse solvers
    - Tailor computation to matrix structure
    - Optimized SpMV for multiple vectors
    - Block Krylov methods
    - Block eigensolvers
  - Parallel preconditions
  - Parallel Graph Algorithms
    - (Maximal) Independent sets
    - Graph colorings
    - Partitioning of graphs that minimize a given metric (e.g. edge cut)
    - Optimal network flows
    - Graph clustering
More Project Ideas

• Suggestions from Steve Parker:
  – Non-uniform, unstructured
  – Double precision, cluster-based

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

A Few Details

• C stores multi-dimensional arrays in row major order
• Fortran (and MATLAB) stores multi-dimensional arrays in column major order
  – Confusion alert: BLAS libraries were designed for FORTRAN codes, so column major order is implicit in CUBLAS!

SGEMV (single-precision)

sgemv (matrix vector multiplication) and transposed sgemv

Recall transpose vs. column-major assumption

// This is transposed due to // Fortran assumption // Follows coalesced accesses // Place b in shared memory for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        c[j] += a[i*n+j] * b[j];

// Non-transposed // Store a in a shared memory // and access in a different order for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        c[j] += a[i*n+j] * b[j];

Non-transposed more than 2X slower for 8K matrix!

Dependences in 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];
}

Which loop(s) "carry" dependences?
Which loop(s) is(are) safe to execute in parallel?

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

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

Results provided by Malik Khan, Gabe Rudy, Chun Chen
Recent Results

Suppose your matrix is not evenly divided by the tile sizes (64 and 16)?

512x512 matrix sgemm:
CUBLAS 258 Gflops
Our compiler 284 Gflops

500x500 matrix sgemm:
CUBLAS 167 Gflops
Our compiler 223 Gflops

Breaking it Down: Tiling

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: Higher-Level Interface (in progress)

init("mm.sp2", "MarkedLoop")
tile_control("(i,j)", (TI,TJ),
(1,1_control="ii", 1,2_control="jj"),
("i", "j", "i", "j"))
tile_control("(k)", (TK), (1,1_control="kk"),
("i", "j", "kk", "i", "j", k), strided)
tile_control("(i,j)", (TI),
(1,1_control="ty", 1,1_tile="tx"),
("i", "j", "kk", "tx", "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)", 0})

--Copy the "c" array usage to registers
copy_to_registers("kk", "c", "tx", ty")
copy_to_shared("ty", "y")

--Unroll two innermost loop levels fully
unroll_to_depth(1)

---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)", 0})

--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)", 0})
Breaking it Down: Tiling

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];

1. tile_control(“i”, “j”), {TI, TJ},
   {l1_control=“ii”, l2_control=“jj”},
   (“ii”, “jj”, “i”, “j”)
2. tile_control(“k”), {TK}, {l1_control=“kk”},
   (“ii”, “jj”, “kk”, “i”, “j”, “k”), strided
3. tile_control(“i”), {TI},
   {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},
 {block=“ii”, “jj”}, {thread=“tx”, “ty”})

--- Copy the “c” array usage to registers
6. copy_to_registers(“tx”, “ty”)
5. copy_to_registers(“k”, “c”, “t”, “tx”, “ty”)
LU Decomposition (no pivoting)

- LAPACK version decomposes this into three parts: mini-LU, (S/D)TRSM, & (S/D) GEMM
- Row pivoting (used for numeric stability) reorders matrix and computation

DO K=1,N-1
  DO I=K+1,N
    A(I,K)=A(I,K)/A(K,K)
  END DO
  DO J=K+1,N
    A(J)=A(J)-A(J,K)*A(K,J)
  END DO

What's Coming

- See Khan/Rudy poster on Friday!
- Before Spring Break
  - Two application case studies from newly published Kirk and Hwu
  - Sparse linear algebra
  - Review for Midterm