Batch solution of scalar and block-tridiagonal equations on many-core accelerators

Endre László

1 University of Oxford, Oxford e-Research Center, Oxford, United Kingdom
2 Pázmány Péter Catholic University, Faculty of Information Technology, Budapest, Hungary

23 October 2013
Many-Core Seminar Series
OeRC, University of Oxford
Overview

• ADI (Alternating Direction Implicit) Finite Difference scheme
  – Solving the heat equation in 3D
  – Practical application
  – Tridiagonal system of equations

• Solving tridiagonal systems
  – Motivation for an efficient solver
  – Memory access patterns of the tridiagonal solver
  – Algorithms: Thomas/TDMA, CR/PCR, RD/PP

• Thomas (TDMA) algorithm
  – Optimization on GPUs:
    1. shared memory - transposition
    2. register shuffle – transposition
  – Optimization on AVX and Xeon Phi

• CR/PCR (Parallel Cyclic Reduction)
  – Optimization on GPUs:
    1. CR in shared memory – Mike Giles’s CUDA course practical code
    2. PCR with register shuffle – Jeremy Appleyard’s code based on Mike Giles BS solver

• Benchmark for batch scalar solvers

• Conclusions for batch scalar solvers

• Block Tridiagonal solver base on the Thomas algorithm
  – Optimizations
  – Benchmarks
ADI

• Classical FD scheme
  – Cheaper then Crank-Nicolson
  – Relies on operator factoring
  – $O(\Delta t^2, \Delta x^2)$ order accurate in both space and time
  – Unconditionally stable if parameters chosen right: heat conductance is positive
  – Originally introduced by Peaceman and Rachford\(^1\)
  – Many variants so far

• Assumptions:
  – Computation (space) domain is >2D – in current case 3D
  – “Cubeish” domain: roughly equal sized dimensions
    • In CFD and Financial applications this the real-world case

• The upcoming discussion of tridiagonal solvers is in the context of the ADI method

---

Solving the 3D Poisson equation with ADI

Poisson equation:\[ \frac{du}{dt} = \nabla^2 u \quad u = u(x, y, z, t). \]

Crank-Nicolson discretization:

\[ \frac{u_{ijk}^{n+1} - u_{ijk}^n}{\Delta t} = \frac{1}{2} \left( \frac{1}{\Delta x^2} (\delta_x^2 + \delta_y^2 + \delta_z^2) u_{ijk}^{n+1} + \frac{1}{\Delta x^2} (\delta_x^2 + \delta_y^2 + \delta_z^2) u_{ijk}^n \right) \]

\[ u_{ijk}^{n+1} = u_{ijk}^n + \Delta u_{ijk} \]

\[ \delta_x^2 u_{ijk}^n = u_{i-1,jk}^n - 2u_{ijk}^n + u_{i+1,jk}^n \]

\[ \Delta u_{ijk} = \frac{\Delta t}{\Delta x^2} (\delta_x^2 + \delta_y^2 + \delta_z^2) (u_{ijk}^n + \Delta u_{ijk}) \]

\[ (1 - \lambda \delta_x^2 - \lambda \delta_y^2 - \lambda \delta_z^2) \Delta u_{ijk} = \lambda (\delta_x^2 + \delta_y^2 + \delta_z^2) u_{ijk}^n \]

Approximate factorization with \( O(\Delta x^2) \) error:

\[ (1 - \lambda \delta_x^2) (1 - \lambda \delta_y^2) (1 - \lambda \delta_z^2) \Delta u_{ijk} = \lambda (\delta_x^2 + \delta_y^2 + \delta_z^2) u_{ijk}^n \]
Solving the factored form

Factored form:

\[
(1 - \lambda \delta^2_x) (1 - \lambda \delta^2_y) (1 - \lambda \delta^2_z) \Delta u_{ijk} = \lambda \left( \delta^2_x + \delta^2_y + \delta^2_z \right) u^n_{ijk}
\]

\[
\begin{align*}
(1 - \lambda \delta^2_x) u^{(1)} &= \lambda \left( \delta^2_x + \delta^2_y + \delta^2_z \right) u^n \\
(1 - \lambda \delta^2_y) u^{(2)} &= u^{(1)} \\
(1 - \lambda \delta^2_z) \Delta u &= u^{(2)} \\
u^{n+1} &= u^n + \Delta u
\end{align*}
\]

**Kernel “preproc”**: Preprocessing

**Kernel “trid_x”**: Tridiagonal sys. along X

**Kernel “trid_y”**: Tridiagonal sys. along Y

**Kernel “trid_z”**: Tridiagonal sys. along Z

+ add contribution
The resulting tridiagonal systems

\[
\begin{align*}
\quad u^{(0)} &= \lambda \left( u^{n}_{i-1,jk} + u^{n}_{i+1,jk} + u^{n}_{ij-1k} + u^{n}_{ij+1k} + u^{n}_{ijk-1} + u^{n}_{ijk+1} - 6u^{n}_{ijk} \right) \\
au^{(1)}_{i-1,jk} + bu^{(1)}_{ijk} + cu^{(1)}_{i+1,jk} &= \\
au^{(2)}_{ij-1k} + bu^{(2)}_{ijk} + cu^{(2)}_{ij+1k} &= \\
u^{n+1}_{ijk} &= \\
\Delta u^{n}_{ijk-1} + b\Delta u^{n}_{ijk} + c\Delta u^{n}_{ijk+1} &= \\
\quad u^{n}_{ijk} + \Delta u^{n}_{ijk} \\
\end{align*}
\]

where \( a = -0.5\lambda, \ b = 1 + \lambda, \ c = -0.5\lambda \)
Solving tridiagonal systems

- Motivation for improved algorithm:
  - Poor performance in the cuSPARSE library
  - Lack of long-strided access pattern

- Assumptions:
  - Large number of systems – enough to saturate GPU
  - Relatively large systems
  - Each system has its own coefficients and RHS
  - Typically O(256 x 256) number of systems on a x O(256) cube
  - No pivoting is done: we aim problems, which are diagonally dominant

\[
a_i u_{i-1} + b_i u_i + c_i u_{i+1} = d_i
\]

\[
\begin{pmatrix}
b_0 & c_0 & 0 & 0 & \cdots & 0 \\
a_1 & b_1 & c_1 & 0 & \cdots & 0 \\
0 & a_2 & b_2 & c_2 & \cdots & 0 \\
\vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & 0 & 0 & \cdots & a_N & b_N
\end{pmatrix}
\begin{pmatrix}
u_0 \\
u_1 \\
u_2 \\
\vdots \\
u_N
\end{pmatrix}
=
\begin{pmatrix}
d_0 \\
d_1 \\
d_2 \\
\vdots \\
d_N
\end{pmatrix}
Solving the heat equation

• Finite difference solution
• Crank Nicholson method is expensive in >2D:
  – Sparse matrix
  – Large matrix bandwidth -> unstructured access pattern
• ADI method:
  – Decompose the system into 1D problems by splitting
  – Solve 3 tridiagonal systems -> structured access patterns
  – Solve a tridiagonal system with Thomas, CR (Cyclic Reduction) or PCR (Parallel CR), RD (Recursive Doubling)

\[
\frac{\partial u}{\partial t} = \nabla^2 u
\]

- x-dim
  \[
  (1 - \lambda \delta_x^2)u^{(1)} = \lambda (\delta_x^2 + \delta_y^2 + \delta_z^2)u_{ijk}^n
  \]

- y-dim
  \[
  (1 - \lambda \delta_y^2)u^{(2)} = u^{(1)}
  \]

- z-dim
  \[
  (1 - \lambda \delta_z^2)\Delta u_{ijk} = u^{(2)}
  \]
Thomas algorithm

- Gaussian elimination
  - Sequential algorithm
- 2 passes:
  - **Forward**: eliminate lower diagonal
  - **Backward**: substitute/solve the system
- **O(N)** computation complexity
  - 2*N steps
  - 8*N FLOP total
- 6*N storage requirement
  - 3*N coefficients and N RHS
  - 2*N buffer memory requirement
- 9*N data traffic complexity
- Diagonal dominance is required for stability! (Pivoting?)
- Experiments show that for small systems (100s) Thomas algorithm is more efficient than CR with shared memory
Thomas algorithm

Forward pass

\[ i = 0: \quad c_0^* = \frac{c_0}{b_0}, \quad d_0^* = \frac{d_0}{b_0} \]

\[ i = 1 \ldots (N_T - 1): \quad c_i^* = \frac{c_i}{b_i - a_i c_{i-1}^*}, \quad d_i^* = \frac{d_i - a_i d_{i-1}^*}{b_i - a_i c_{i-1}^*} u \]

Backward pass

\[ i = N_T - 1: \quad u_{N_T - 1} = d_i^* \]

\[ i = (N_T - 2) \ldots 0: \quad u_i = d_i^* - c_i^* u_{i+1} \]
cuSPARSE

- Gtsv functions
- Inefficient
- Extra space requirements: 768MB for a $256^3$ problem
- Uses two kernel calls
- Need to transpose the “layers” of the domain
ADI Solver works in 4 steps:

1: Preprocessing

\[ u^{(0)} = \lambda (\delta_x^2 + \delta_y^2 + \delta_z^2) u_{ijk}^n \]

2: Solve X dim

\[ (1 - \lambda \delta_x^2) u^{(1)} = u^{(0)} \]

3: Solve Y dim

\[ (1 - \lambda \delta_y^2) u^{(2)} = u^{(1)} \]

4: Solve Z dim

\[ (1 - \lambda \delta_z^2) \Delta u_{ijk} = u^{(2)} \]

1 iteration

PreProc \rightarrow Trid-X \rightarrow Trid-Y \rightarrow Trid-Z
Data alignment

3D data linearly laid out in memory:

\[ \text{index} = k \cdot NX \cdot NY + j \cdot NX + i \]
Stencil based read access pattern: \[ u^{(0)} = \lambda (\delta_x^2 + \delta_y^2 + \delta_z^2) u_{ijk} \]

- Threads layed on X-Y plane
  - Iterate along dimension Z
- Coalesced memory access:
  - 11 (global) arrays: \( a_x b_x c_x a_y b_y c_y a_z b_z c_z u du \)
  - Stride=1 (between threads)
  - Offset=NX*NY (within thread)

4x4 Threads
\[(1 - \lambda \delta^2_y) u^{(2)} = u^{(1)}\]

- Threads layed on X-Z plane
  - Iterate along dimension Y
- Coalesced memory access:
  - 4 (global) arrays: \(a_y, b_y, c_y, du\)
  - 2 (local) arrays: \(c_2, d_2\)
- Stride=1 (between threads)
- Offset=NX (within thread)

index = \(k \cdot NX \cdot NY + j \cdot NX + i\)
\[(1 - \lambda \delta_z^2) \Delta u_{ijk} = u^{(2)}\]

- Threads layed on X-Y plane
  - Iterate along dimension Z
- Coalesced memory access:
  - 4 (global) arrays: \(a_z, b_z, c_z, du\)
  - 2 (local) arrays: \(c_z, d_z\)
- Stride=1 (between threads)
- Offset=NX * NY (within thread )
\[(1 - \lambda \delta_x^2)u^{(1)} = u^{(0)}\]

- Threads layer on Y-Z plane
  - Iterate along dimension X
- Non-coalesced memory access:
  - 4 (global) arrays: \(a_x, b_x, c_x, du\)
  - 2 (local) arrays: \(c_2, d_2\)
- Stride=NX * NY (between threads)
- Offset=1 (within thread)
Access patterns

• Different access patterns along different axis
  – X dim: worst
    • 4byte/128byte = 3.125% cache line utilization
  – Y dim: best
    • perfectly coalesced, 100% utilization
  – Z dim: moderate
    • coalesced with moderate TLB miss rate?

x13.5 difference!
Optimized X-solver
(shared memory)

Forward pass:
1. Wrap a warp (32 threads) into 4x8 blocks to perform non-caching loads
2. Load 32x8 size tiles into shared memory:
   - 8 steps of 4x8S blocks loads
3. Transpose data into registers:
   - `float a[8];` is compiled to 8 registers if array indexing is known in compile time
4. Perform calculation with the 8 values along X dimension
5. Repeat from 2 until end of X-dim is reached

Backward pass:
1. Do the same backward: transpose + store
Pros:
• Non-caching (32byte) loads
  – 100% memory access utilization
• No need for __syncthreads():
  – Threads share data only within warp -> safe
  – Asynchronous
• Significant (average) speedup:
  – Fermi: 5.7x, 80+ GB/s
  – Kepler: 8.7x, 120+ GB/s

Cons:
• Moderate shared memory and register needs
  – Moderate occupancy
• High “Global memory replay”
  – Reported by NVVP
  – In case of cache (TLB?) miss, instruction are reissued
  – Suggests that some memory access pattern is still needs improvement
Bottleneck: TLB hit rate?

- CUDA uses Unified Virtual Address Space
- Virtual address space uses memory pages
- Memory page frame pointers are cached in TLB (Translation Lookaside Buffer) translate
- TLB is a “coarser” cache that
  - goes hand in hand with LLC
  - translates address tag to frame pointer
  - caches frame pointers from main memory
- On NVIDIA devices TLB is hardware implemented and page sizes can not be changed
- Small page size -> high TLB miss rate on long stride memory access (like in trid-x)
- Solution: load data along dimension X in big chunks
Optimized X-solver
(register shuffle)

Forward pass:
1. Wrap 32 threads into 8x4 blocks to perform float4 vector loads
2. Load 32x16 size tiles into registers:
   - 4 threads read 4 consecutive float4 vectors = 64bytes
   - Do this 4 times for rows under each other
3. Transpose data within 4 threads:
   - 4 threads exchange data with __shfl()
4. Perform calculation with the 16 values along X dimension
5. Repeat from 2 until end of X-dim is reached

Backward pass:
1. Do the same backward: transpose + store
Analysis
(register shuffle based trid-X solver)

Pros:
• Non-caching (2x32=64byte) loads
  – 100% memory access utilization
• Fast intra-warp communication
  – Due to register-level communication
• Significant speedup:
  – Kepler: 10x, 150 GB/s

Cons:
• High register pressure (~96):
  – 25% occupancy, still gives the best performance
  – Might mean problem when more arithmetics are involved
• Works only on Kepler SM >=3.0
Tridiagonal solver (X-dim) speedup

On NVIDIA Tesla K20c

GPU kernel execution times

Peak speedup on 256³ SP float cube

x6.3

x11.8

On NVIDIA Tesla K20c
Overall speedup

- 2 socket 6 core *Sandy Bridge* E5-2640 @2.5GHz, 2x15 MB LLC Cache per socket, 2x42.6GB/s
- *NVIDIA Tesla K20c*, 2688 CUDA cores, 250GB/s
Average bandwidth of the whole problem

Avg. Bandwidth [GB/s], GB = $10^9$ Byte SP

Avg. Bandwidth [GB/s], GB = $10^9$ Byte DP

- SIMD
- SIMD_OMP
- PHI
- CUDA_base
- CUDA_shared
- CUDA_reg
- CUDA_cuSPARSE

CUBE size [grid points]
Total execution time

![Graph showing total execution time for different methods and cube sizes.](image)
Speedups of the X dimension solver
Optimization on Y and Z dimension

• Instead of solving 1 system per thread
  – Solve 4 systems per thread with float4 vectors or
  – Solve 2 systems per thread with double2 vectors

• Pros:
  – Reduces index arithmetic
  – Fills up the memory pipeline more efficiently
  – Gives ~15% speedup

• Cons:
  – Only work well for aligned access patterns

• `helper_math.h` in CUDA SDK comes handy
  – Overloads operations for float4 and double2
CR - Cyclic Reduction

• 2 passes:
  – **Forward**: reduce systems to subsystems
  – **Backward**: solve the subsystems

• Log(N) complexity on N processors
  – 2*log(N)-1 steps
  – 17*N FLOP total

\[-a_n x_{n-1} + x_n - c_n x_{n+1} = d_n, \quad n = 1, \ldots, N\]

for even \(n\) add \(a_n\) times row \(n-1\) and \(c_n\) times row \(n+1\):

\[-a_n a_{n-1} x_{n-2} + (1-a_n c_{n-1} - c_n a_{n+1}) x_n - c_n c_{n+1} x_{n+2} = d_n + a_n d_{n-1} + c_n d_{n+1}, \quad n = 2, 4, \ldots\]

Eliminates \(x_{n-1}\) and \(x_{n+1}\)

Normalize

\[-a_n^* x_{n-2} + x_n - c_n^* x_{n+2} = d_n^*, \quad n = 2, 4, \ldots\]
PCR - Parallel CR

- 1 pass (instead of 2 in CR):
  - Forward: reduce systems to subsystems
  - In every step:
    - The number of systems is doubled
    - The number of unknowns in these systems are halved
  - For the end of the forward step the whole system is solved

- Log(N) complexity on N processors
  - 2*log(N)-1 steps
  - 17*N FLOP total

- 5*N data traffic complexity

\[
\begin{pmatrix}
  b_0 & c_0 & 0 & 0 & \cdots & 0 \\
  a_1 & b_1 & c_1 & 0 & \cdots & 0 \\
  0 & a_2 & b_2 & c_2 & \cdots & 0 \\
  \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
  0 & 0 & 0 & 0 & \cdots & a_N & b_N
\end{pmatrix}
\begin{pmatrix}
  u_0 \\
  u_1 \\
  u_2 \\
  u_3 \\
  \vdots \\
  u_N
\end{pmatrix}
= 
\begin{pmatrix}
  d_0 \\
  d_1 \\
  d_2 \\
  d_3 \\
  \vdots \\
  d_N
\end{pmatrix}
\]
Modified PCR

- PCR + Gauss Jordan hybrid
- 256 long system size is assumed in the following discussion
  - The problem is divided into 32 8-long chunks
  - Every chunk is solved by a thread with Gauss-Jordan:
    • lower and upper off-diagonals are eliminated in a forward and backward sweep
    • Eliminating upper values would introduce more non-zeros -> pentadiagonal
    • At this point the system can not be solved, since neighboring/overlapping values are not known
  - Use PCR to solve the overlapping values
  - Solve the 8-long chunks with back-substitution

- Algorithm:
  - Step 0: divide problem across the threads of a warp and do G-J on 8-long systems
  - Step 1: eliminate lower and upper off-diagonal of 8-long systems
  - Step 2: do PCR where every thread handles 2 rows on the previously reduced system. That is, solve a 2x32=64 sized system with PCR
  - Step 3: solve the 8-sized chunks with back-substitution
## Results

### Thomas vs PCR+GJ

<table>
<thead>
<tr>
<th>NVIDIA Card</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>K20c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GTX680</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NVIDIA Card</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>K20c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GTX680</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Conclusions
Thomas vs. PCR

Thomas
• Pros:
  – Easy to implement for non-1-strided dimension
  – Low register pressure for non-1-strided dimension
  – Low arithmetic complexity
• Cons:
  – O(N), O(N^2) in total for N systems
  – Relatively high data traffic: 9 x N long arrays -> more serious effect of memory-bound
  – Difficult to implement for 1-strided dimension
  – High register pressure for 1-strided dimension

PCR+GJ
• Pros:
  • O(log(N)) complexity on N processors, so O(N*log(N)) in total for N systems
  • Relatively low data traffic: 5 x N long arrays -> serious savings on memory traffic
• Cons:
  • Only works for systems that can fit in the buffers (shared mem., registers)
  • Difficult to implement for 1-stride and even more difficult for non-1-stride
  • High register pressure
  • Higher arithmetic complexity -> but still memory bound

• What about multiGPU solvers?
  • Thomas would fit better: O(1) communication
    • Communication can be batched -> more efficient
    • PCR requires significantly more communication
  • Pivoting is probably easier with Thomas
Conclusion

- **ADI with >3D faces the problem of long-strided memory access**

- **Shared memory implementation**
  - To get coalesced access
  - Efficient transpose in shared memory
  - ~6x speedup @ 50% occupancy

- **Register implementation**
  - To get coalesced access and overcome TLB bottleneck
  - Fast transpose with __shfl() 
  - ~12x speedup @ 12.5% occupancy

- **Lower occupancy doesn’t necessarily lead to worse performance**
  - Volkov, “Better performance at lower occupancy,” GTC2010

- **TLB miss hits performance seriously:**
  - Wong et al., “Demystifying GPU microarchitecture through microbenchmarking,” IEEE, 2010

---

**GPU kernel execution times**

<table>
<thead>
<tr>
<th></th>
<th>Exec. time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trid-X</td>
<td>50</td>
</tr>
<tr>
<td>Trid-X shared</td>
<td>30.3</td>
</tr>
<tr>
<td>Trid-X reg</td>
<td>18.2</td>
</tr>
</tbody>
</table>

\[x6.3\] \[x11.8\]
Block Tridiagonal Thomas algorithm

- A block tridiagonal solver becomes memory computation bound after a certain block size
- Arithmetic complexity of Thomas, CR, PCR, RD suggest that Thomas algorithm is a better choice for block tridiagonals
  - Or is it? To what extent?
- CR, PCR and RD are common choice in many publications
- Our first choice is Thomas
Block Tridiagonal problem

\[ A_i u_{i-1} + B_i u_i + C_i u_{i+1} = d_i \]

\[
\begin{pmatrix}
  B_0 & C_0 & 0 & 0 & \cdots & 0 \\
  A_1 & B_1 & C_1 & 0 & \cdots & 0 \\
   0 & A_2 & B_2 & C_2 & \cdots & 0 \\
   \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
   0 & 0 & 0 & \cdots & A_N & B_N
\end{pmatrix}

\begin{pmatrix}
  u_0 \\
  u_1 \\
  u_2 \\
  u_3 \\
  \vdots \\
  u_N
\end{pmatrix} =

\begin{pmatrix}
  d_0 \\
  d_1 \\
  d_2 \\
  d_3 \\
  \vdots \\
  d_N
\end{pmatrix}

\( A_i, B_i, C_i \in \mathbb{R}^{M \times M}, u, d \in \mathbb{R}^M \) and typically \( M \in [2, 6] \)

In a CFD application
Block Tridiagonal Thomas algorithm

The *Forward pass* modifies to:

\[ i = 0 \]

\[ C_0^* = \frac{B_0^{-1}C_0}{d_0^*} = \frac{B_0^{-1}d_0}{B_0^{-1}} \]

\[ i = 1, \ldots, N_T \]

\[ C_i^* = \frac{(B_i - A_iB_{i-1}^*)^{-1}C_i}{d_i^*} = \frac{(B_i - A_iB_{i-1}^*)^{-1}(d_i - A_id_{i-1}^*)}{(B_i - A_iB_{i-1}^*)^{-1}} \]

The *Backward pass* modifies to:

\[ i = N_T - 1 \]

\[ u_{N_T-1} = d_i^* \]

\[ i = N_T - 2, \ldots, 0 \]

\[ u_i = d_i^* - C_i^*u_{i+1} \]
Optimization

• Warp level optimization
  – \( M \) number of threads share work to solve a row
  – floor\((32/M)\) systems are solved in parallel by a warp
  – Rest of the threads in the warp are idle

• Matrix operations done by the cooperating threads

• Threads read data in a coalesced way
Data storage

• Dictated by the warp level optimization
• Blocks are loaded from global memory to registers
  – Coalesced memory access
  – Every cooperating thread has a column of a block
  – \( \mathbf{C} \) is a block of coefficients: either \( \mathbf{A} \), \( \mathbf{B} \) or \( \mathbf{C} \)
  – \( \mathbf{C}_i^p \) is the \( i \)-th row of problem \( p \)

\[
C_1^1, C_1^2, \ldots, C_1^P, \ldots, C_2^1, C_2^2, \ldots, C_2^P, \ldots, C_n^1, C_n^2, \ldots, C_n^P, \ldots, C_N^1, C_N^2, \ldots, C_N^P
\]
Matrix-matrix multiplication with cooperating threads

\[ C_{ij} = \sum_{k} A_{ik} B_{kj}. \]

// Thread t, has A(0,t), A(1,t), ... , A(N-1,t) stored in registers.
// Similarly for B and C.
// Let s be one element of shared memory

// Each thread executes the following code:

for k = 0 --> dim-1
    for i = 0 --> dim-1
        thread k: s = A(i,k)
        C(i,t) += s * B(k,t)
Matrix-vector multiplication with cooperating threads

\[ c_i = \sum_j A_{ij} b_j. \]

```c
// Thread t, has A(0,t), A(1,t), ..., A(N-1,t) and vector element b(t) stored in registers.
// Let s( ) be an array in shared memory of length dim

// Each thread executes the following code:

    for( i = 0 --> dim-1 ){
        idx = (t+i)%dim
        s(idx) += A(idx,t)*b(t)
    }

    b(t) = s(t)
```

To avoid shared memory bank conflicts, strided a shared memory access is used.

**NO BANK CONFLICT**

\[
\begin{pmatrix}
  c_1 \\
  c_2 \\
  c_3
\end{pmatrix} = 
\begin{pmatrix}
  A_{1,1} b_1 \\
  A_{2,2} b_2 \\
  A_{3,3} b_3
\end{pmatrix} + 
\begin{pmatrix}
  A_{1,3} b_3 \\
  A_{2,1} b_1 \\
  A_{3,2} b_2
\end{pmatrix} + 
\begin{pmatrix}
  A_{1,2} b_2 \\
  A_{2,3} b_3 \\
  A_{3,1} b_1
\end{pmatrix}
\]

**BANK CONFLICT**
Gauss-Jordan solver with cooperating threads

```c
// Thread t, has A(0,t), A(1,t), ... , A(N-1,t),
// B(0,t), B(1,t), ... , B(N-1,t) and element b(t) stored in registers.
// Let s be an element in shared memory
// Each thread executes the following code:

for i = 0 --> N-1

    // choose the pivot
    thread i: s = A(i,i)

    // divide row i by the pivot
    A(i,t) = A(i,t) / s
    B(i,t) = B(i,t) / s

    // broadcast the vector element from the pivot row
    thread i: s = b(i)
    b2 = s

    for j = 0 --> N-1

        if i != j

            thread i: s = A(j,i)

            // subtract a multiple of the pivot row
            // to make all entries above and below the
            // pivot 0
            A(j,t) -= s * A(i,t)
            B(j,t) -= s * B(i,t)

            thread j: b(j) -= s * b2 //equivalent to A(i,j) * b(i)
```

AX=B G-J solve
Ax=b G-J solve
Performance results

Run Time

(a) single

(b) double
Compute

(a) single

(b) double
Throughput

(a) single

(b) double
Future work

• Test results against Intel MKL and LAPACK
• Extend PCR capability to fit larger problems efficiently
• Compare result with Recursive Doubling
• Create library quality software based on results