A hybrid Cholesky decomposition algorithm for multicore CPUs with GPU accelerators

Gary Macindoe

Department of Statistical Science
University College London

8th February 2013

Cholesky Decomposition

- Used throughout Computational Statistics and Machine Learning
- Finds L such that $A = LL^T$
- "Square root" of a matrix
- $O(N^3)$ operations
- Performance bottleneck
- Applies only to symmetric, square, positive definite matrices
- Operates in the upper or lower triangle
- Provides fast ways of computing the inverse and determinant

Example use of Cholesky Decomposition

Used in multivariate Normal distribution

Example use of Cholesky Decomposition

Used in multivariate Normal distribution

- To generate random vectors $\sim \mathcal{N}(\mu, \Sigma)$

$$z \sim \mathcal{N}(0, 1)$$

$$x = \mu + \sqrt{\Sigma} z$$

Example use of Cholesky Decomposition

Used in multivariate Normal distribution

- To generate random vectors $\sim \mathcal{N}(\mu, \Sigma)$

$$z \sim \mathcal{N}(0, 1)$$

$$x = \mu + \sqrt{\Sigma} z$$

- To calculate the probability density function

$$(2\pi)^{-\frac{n}{2}} |\Sigma|^{-\frac{1}{2}} e^{-\frac{1}{2}(x-\mu)^T \Sigma^{-1} (x-\mu)}$$

Calculating the Cholesky Decomposition

Definition

Lower triangular Cholesky Decomposition $A = LL^T$

$$L_{i,j} = \begin{cases} \sqrt{A_{j,j} - \sum_{k=1}^{j-1} L_{j,k}^2} & \text{if } i == j \\ \frac{1}{L_{j,j}} \left(A_{i,j} - \sum_{k=1}^{j-1} L_{i,k} L_{j,k} \right) & \text{if } i > j \end{cases}$$

Calculating the Cholesky Decomposition

Definition

Lower triangular Cholesky Decomposition $A = LL^T$

$$L_{i,j} = \begin{cases} \sqrt{A_{j,j} - \sum_{k=1}^{j-1} L_{j,k}^2} & \text{if } i == j \\ \frac{1}{L_{j,j}} \left(A_{i,j} - \sum_{k=1}^{j-1} L_{i,k} L_{j,k} \right) & \text{if } i > j \end{cases}$$

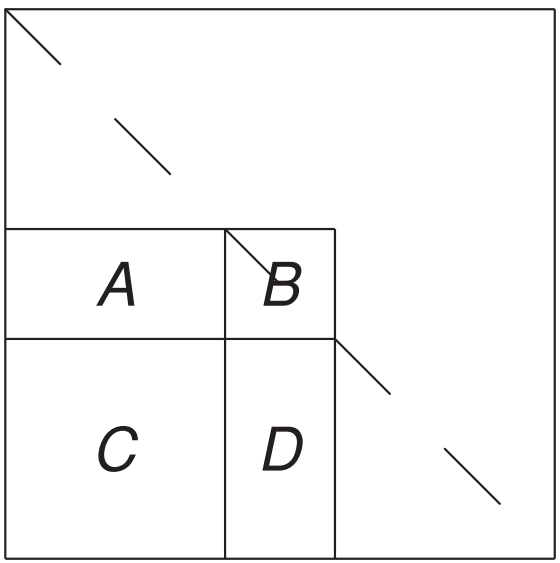
Each element $L_{i,j}$ of the lower triangular Cholesky decomposition can only be calculated after all the elements to the left on the same row $L_{i,0 \to j}$ and on the diagonal row above $L_{j,0 \to j}$. If the sum under the square root is negative then the A is not positive definite.

Blocked Cholesky decomposition

Divide the matrix into blocks and update each block using a high performance linear algebra library (BLAS).

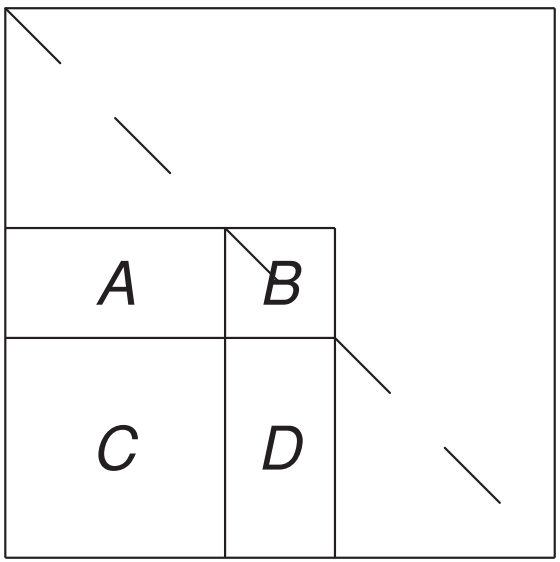
Blocked Cholesky decomposition

Divide the matrix into blocks and update each block using a high performance linear algebra library (BLAS).



Blocked Cholesky decomposition

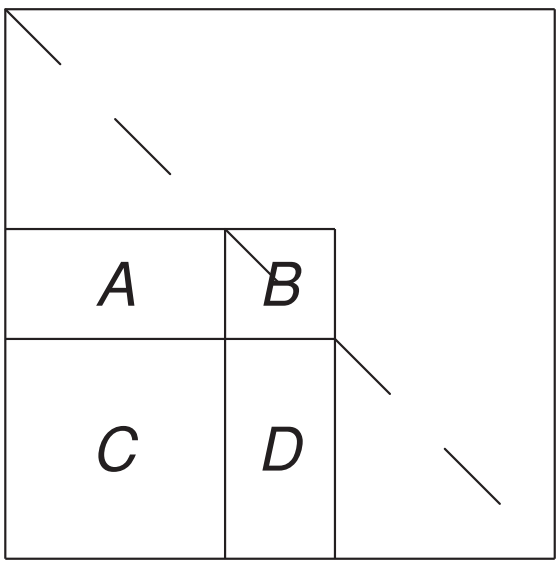
Divide the matrix into blocks and update each block using a high performance linear algebra library (BLAS).



B starts at top left and moves to bottom right

Blocked Cholesky decomposition

Divide the matrix into blocks and update each block using a high performance linear algebra library (BLAS).

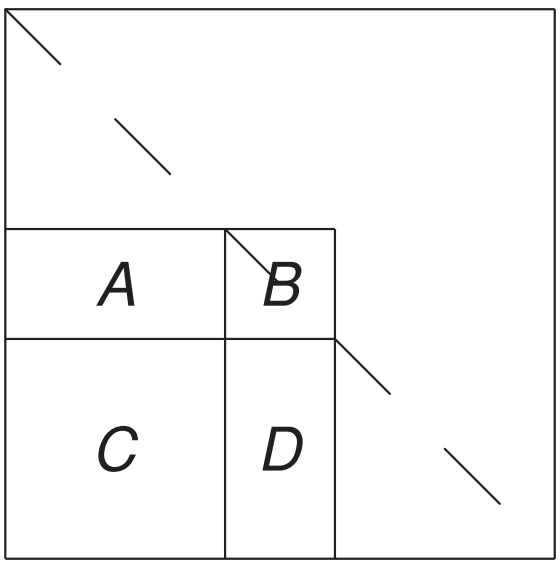


B starts at top left and moves to bottom right

1: $B = B - AA^T$

Blocked Cholesky decomposition

Divide the matrix into blocks and update each block using a high performance linear algebra library (BLAS).

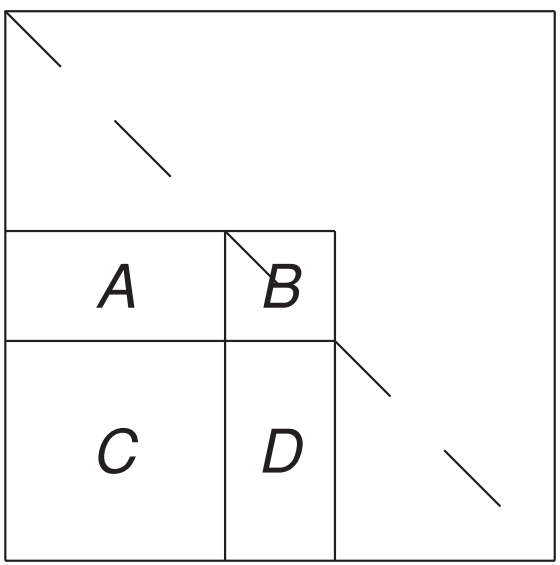


B starts at top left and moves to bottom right

1: $B = B - AA^T$
2: $D = D - CA^T$

Blocked Cholesky decomposition

Divide the matrix into blocks and update each block using a high performance linear algebra library (BLAS).

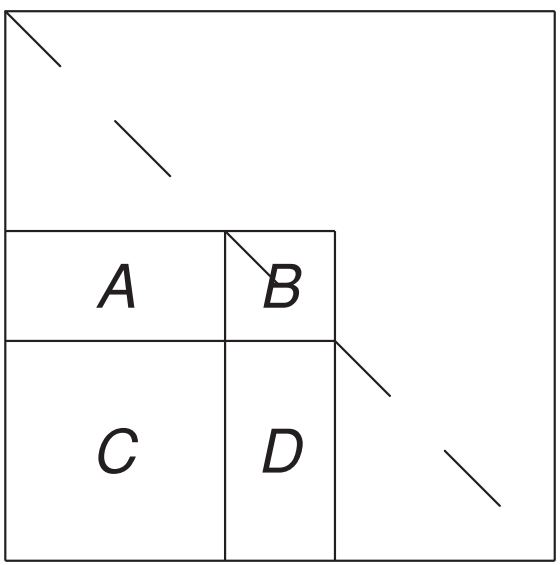


B starts at top left and moves to bottom right

1: $B = B - AA^T$
2: $D = D - CA^T$
3: $B = chol(B)$

Blocked Cholesky decomposition

Divide the matrix into blocks and update each block using a high performance linear algebra library (BLAS).

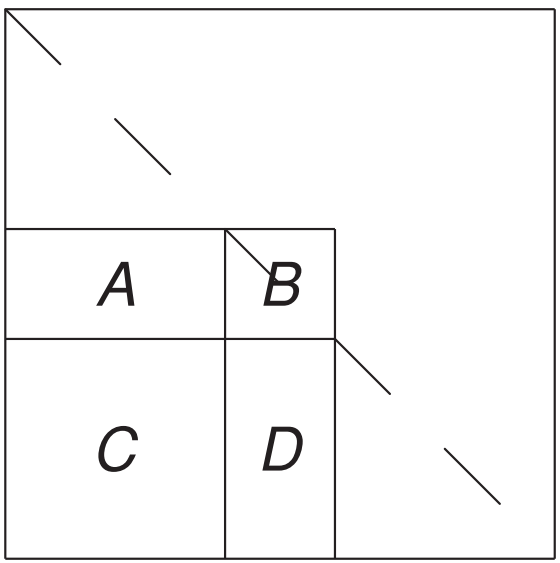


B starts at top left and moves to bottom right

1: $B = B - AA^T$
2: $D = D - CA^T$
3: $B = chol(B)$
4: $D = D(B^{-1})^T$

Blocked Cholesky decomposition

Divide the matrix into blocks and update each block using a high performance linear algebra library (BLAS).

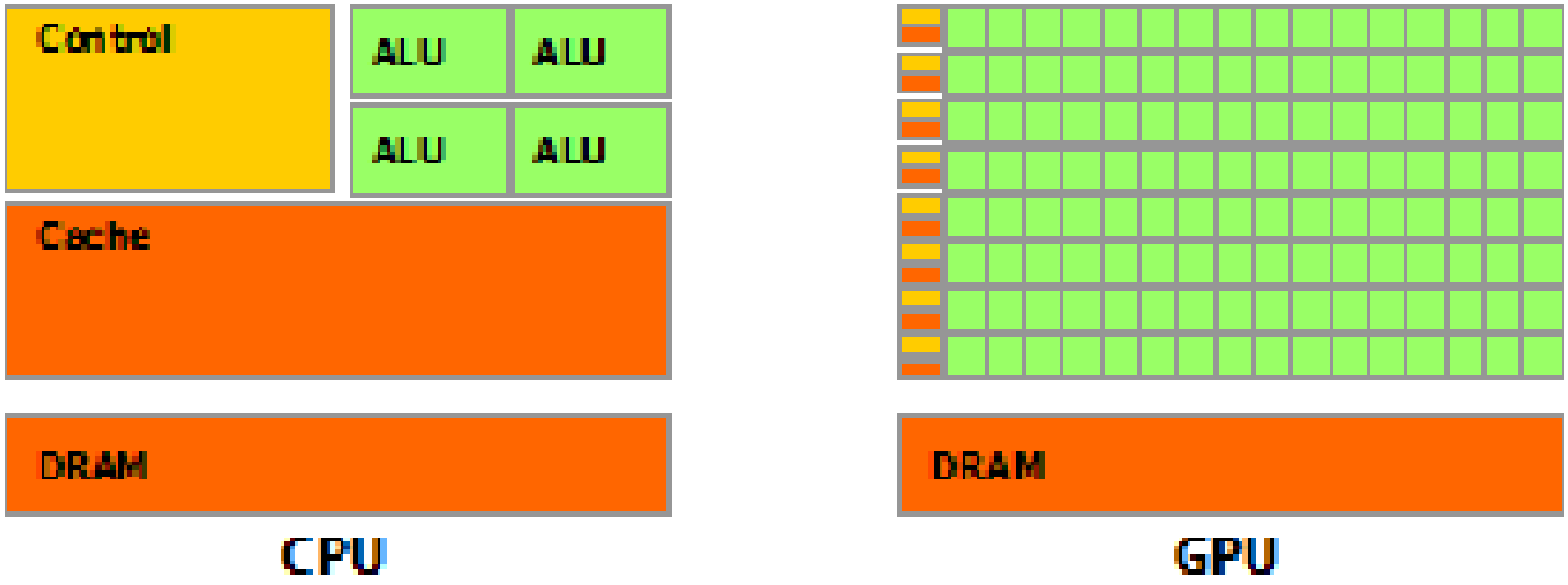


B starts at top left and moves to bottom right

1: $B = B - AA^T$
2: $D = D - CA^T$
3: $B = chol(B)$
4: $D = D(B^{-1})^T$

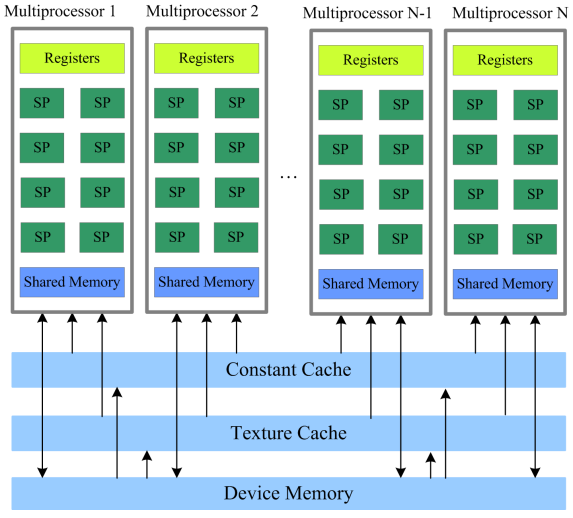
Step 2 is independent of steps 1 and 3.

GPU Hardware



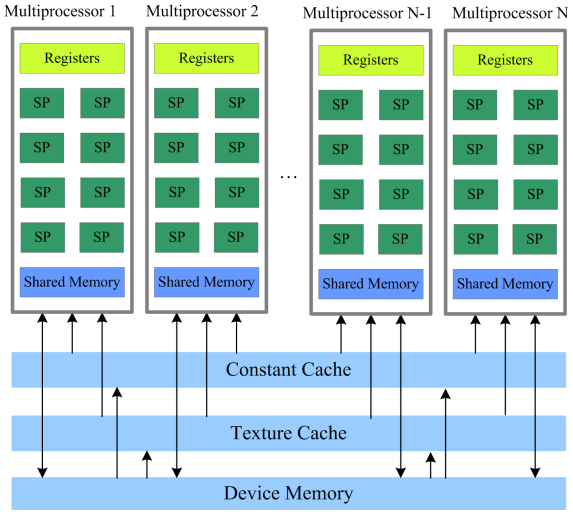
- GPU dedicates more die area to data processing
- CPU dedicates more die area to control flow and cache

GPU Hardware



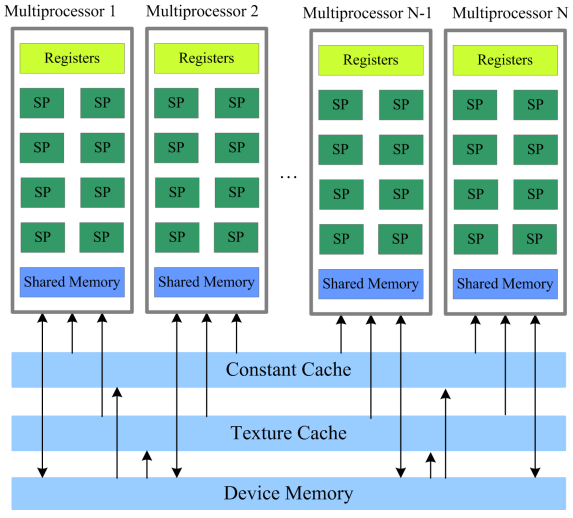
- A GPU is an array of Streaming Multiprocessors (SMs)

GPU Hardware



- A GPU is an array of Streaming Multiprocessors (SMs)
- Each SM has its own shared memory and registers

GPU Hardware



- A GPU is an array of Streaming Multiprocessors (SMs)
- Each SM has its own shared memory and registers
- Each SM is simpler than a CPU core
  - Better at simple arithmetic ($+$, $-$, $\times$)
  - Worse at complex arithmetic ($\div$, *sin*, *cos*, *tan*, *log*, *exp*,...)

GPU Programming

- GPUs execute kernel functions written in CUDA-C

```
template <typename T>
__global__ void scale(int n, T alpha, T * x, int incx) {
  const int i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i < n)
    x[i * incx] *= alpha;
}
```

- nVidia CUDA compiler converts CUDA-C code into GPU binary files
- CUDA runtime library provides an API to
  - transfer compiled code and data onto the GPU
  - launch kernel functions using a 3D grid of 3D thread blocks

GPU Thread hierarchy

- Each thread block runs on one SM
  - Each SM runs more than one thread block
- Within each thread block threads are multitasked in groups of 32 called warps
- Within a warp threads are SIMD
  - Run the same instruction at the same time
- Threads within a block can synchronize with each other
  - Ensures all threads in the block are at the same instruction

GPU Memory hierarchy

GPUs have three types of memory to store data:

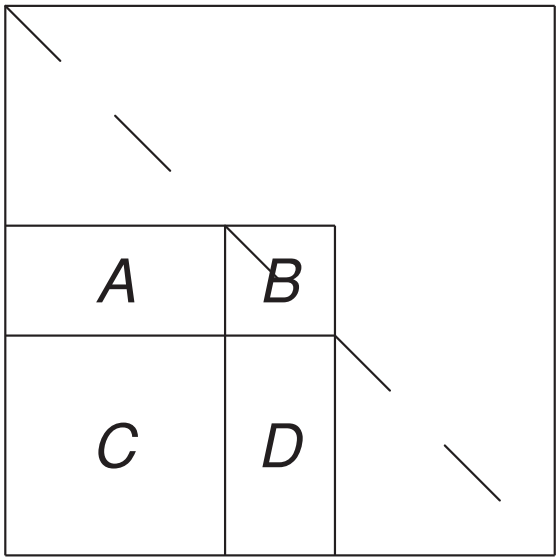
- Global memory - large, slow, accessible from all threads (and the CPU)
- Shared memory - small, fast, shared between threads in a block
- Registers - small, fastest, private to each thread

In global and shared memory highest bandwidth is obtained when consecutive threads access consecutive elements.

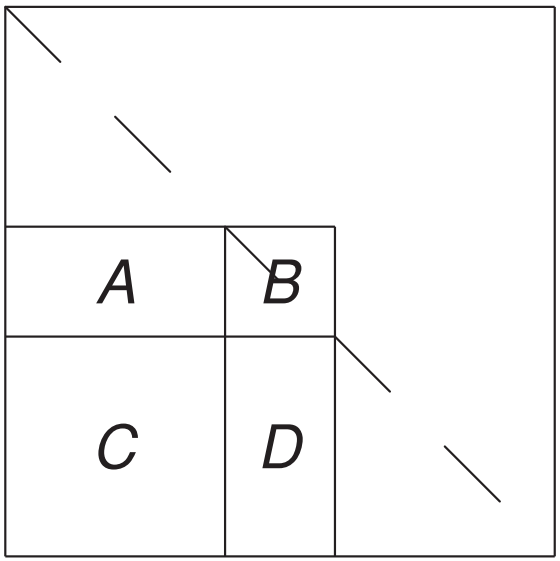
GPU vs CPU

- CPU better at
  - complex maths (like square root in Cholesky)
  - branching (if/then/else)
- GPU better at
  - simple maths (sums, multiplies)
  - executing operations in parallel

Hybrid Blocked Cholesky decomposition

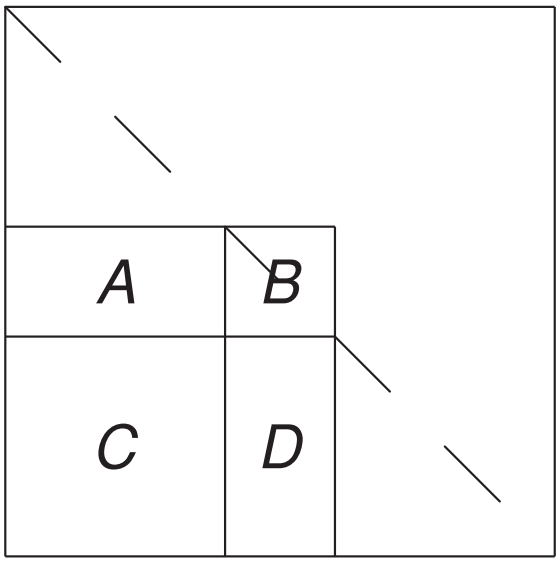


Hybrid Blocked Cholesky decomposition



B starts at top left and moves to bottom right

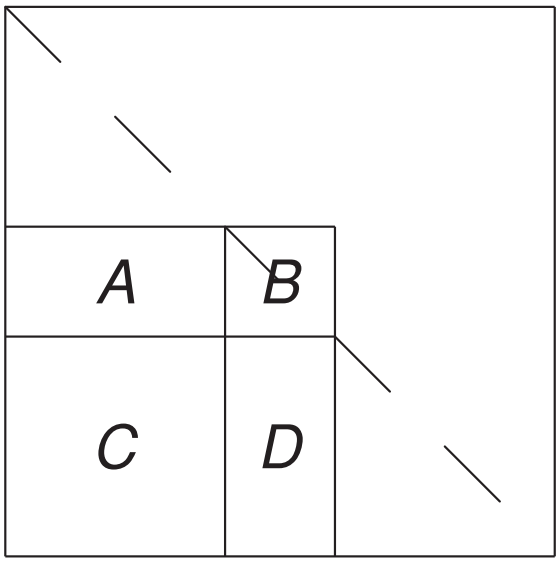
Hybrid Blocked Cholesky decomposition



B starts at top left and moves to bottom right

1: $B = B - AA^T$

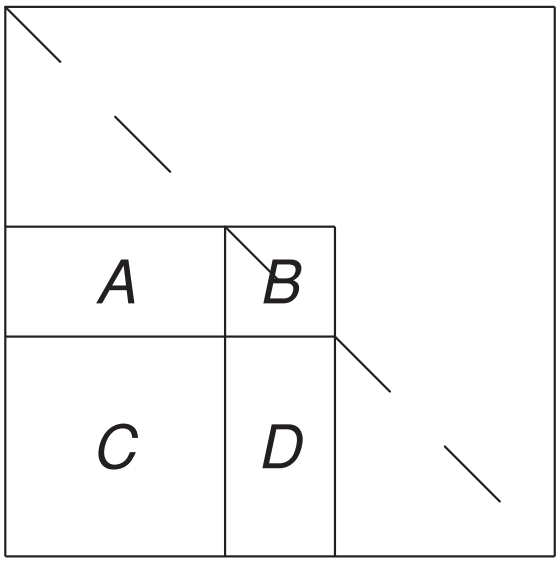
Hybrid Blocked Cholesky decomposition



B starts at top left and moves to bottom right

1: $B = B - AA^T$
2: $D = D - CA^T$

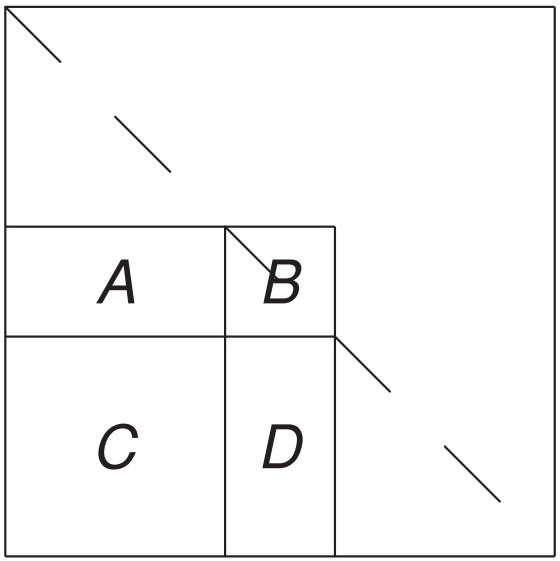
Hybrid Blocked Cholesky decomposition



B starts at top left and moves to bottom right

1: $B = B - AA^T$
2: $D = D - CA^T$
3: $B = chol(B)$

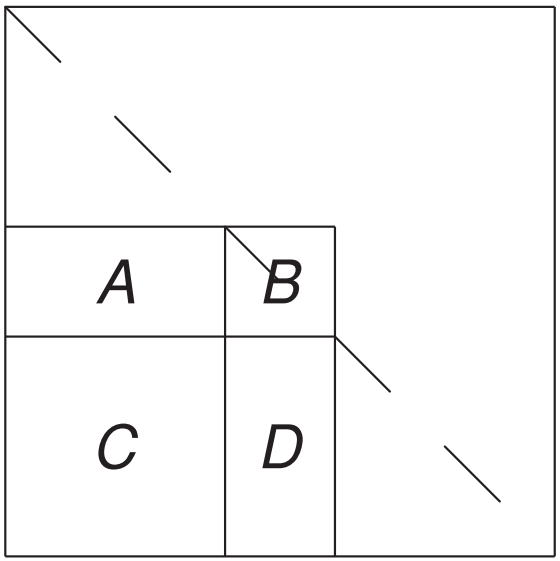
Hybrid Blocked Cholesky decomposition



B starts at top left and moves to bottom right

1: $B = B - AA^T$
2: $D = D - CA^T$
3: $B = chol(B)$
4: $D = D(B^{-1})^T$

Hybrid Blocked Cholesky decomposition



B starts at top left and moves to bottom right

1: $B = B - AA^T$
2: $D = D - CA^T$
3: $B = chol(B)$
4: $D = D(B^{-1})^T$

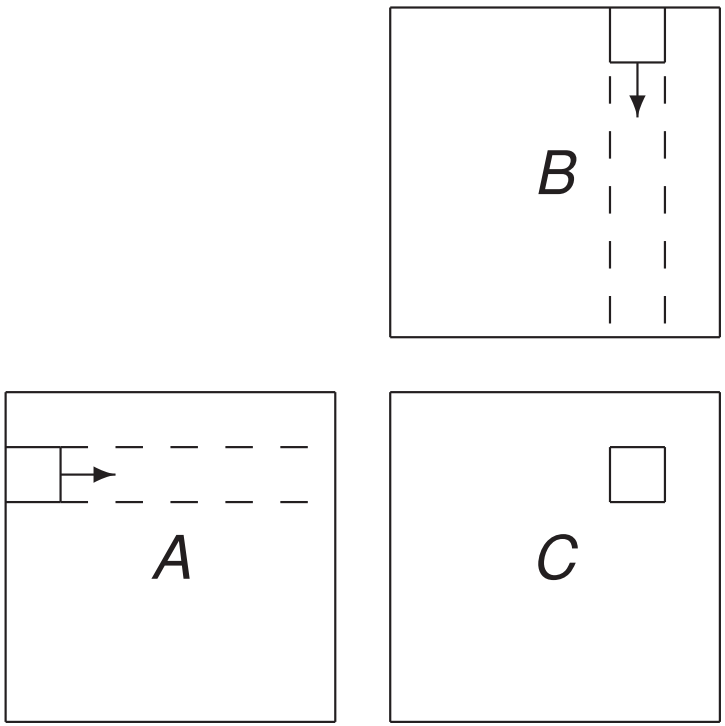
Have GPU perform BLAS operations in steps 1, 2 and 4 which contain lots of simple parallel operations while CPU performs smaller Cholesky decomposition in step 3 which contains more complex serial operations.

GPU Matrix Multiply

$$C = \alpha AB + \beta C$$

- Each element of C is independent
- Have a 2D grid of 2D thread blocks each running $C_{i,j} = \alpha \sum_{l=0}^{k} A_{i,l} B_{l,j} + \beta C_{i,j}$ in parallel
- Calculating one element of C requires reading k elements from $A_{i,0\rightarrow k}$ and k elements from $B_{0\rightarrow k,j}$
- Calculating all of C requires reading $2mnk$ elements from global memory
- Problem: reading elements of A and B from global memory is slow

Blocked GPU Matrix Multiply



- Divide A into blocks of $mb \times kb$, B into blocks of $kb \times nb$ and C into blocks of $mb \times nb$
- Store one block of C in registers and process a $1 \times nb$ row per thread using thread blocks of $mb \times 1$
- Read blocks of A and B into shared memory and access from all threads in the thread block

Bandwidth Reduction

- There are $\frac{m}{mb} \times \frac{n}{nb}$ blocks of C

Bandwidth Reduction

- There are $\frac{m}{mb} \times \frac{n}{nb}$ blocks of C
- Calculating all of C now requires reading

Bandwidth Reduction

- There are $\frac{m}{mb} \times \frac{n}{nb}$ blocks of C
- Calculating all of C now requires reading
  - $\frac{m}{mb} \times \frac{n}{nb} \times \frac{k}{kb} \times mb \times kb$ elements of A

Bandwidth Reduction

- There are $\frac{m}{mb} \times \frac{n}{nb}$ blocks of C
- Calculating all of C now requires reading
  - $\frac{m}{mb} \times \frac{n}{nb} \times \frac{k}{kb} \times mb \times kb$ elements of A and
  - $\frac{m}{mb} \times \frac{n}{nb} \times \frac{k}{kb} \times kb \times nb$ elements of B

Bandwidth Reduction

- There are $\frac{m}{mb} \times \frac{n}{nb}$ blocks of C
- Calculating all of C now requires reading
  - $\frac{m}{mb} \times \frac{n}{nb} \times \frac{k}{kb} \times mb \times kb$ elements of A and
  - $\frac{m}{mb} \times \frac{n}{nb} \times \frac{k}{kb} \times kb \times nb$ elements of B

  or

$$m \times n \times k \times (\frac{1}{mb} + \frac{1}{nb})$$

  elements in total

Bandwidth Reduction

- There are $\frac{m}{mb} \times \frac{n}{nb}$ blocks of C
- Calculating all of C now requires reading
  - $\frac{m}{mb} \times \frac{n}{nb} \times \frac{k}{kb} \times mb \times kb$ elements of A and
  - $\frac{m}{mb} \times \frac{n}{nb} \times \frac{k}{kb} \times kb \times nb$ elements of B

  or

$$m \times n \times k \times (\frac{1}{mb} + \frac{1}{nb})$$

  elements in total
- This is

$$\frac{2}{\frac{1}{mb} + \frac{1}{nb}}$$

  times less than when no blocking is used ($mb = 1$ and $nb = 1$)

Compute Bound Matrix Multiply

Theorem

If the bandwidth reduction is greater than the FLOP:word ratio then the algorithm will be compute bound.[1]

Compute Bound Matrix Multiply

Theorem

If the bandwidth reduction is greater than the FLOP:word ratio then the algorithm will be compute bound.[1]

- What is the FLOP:word ratio?

Compute Bound Matrix Multiply

Theorem

If the bandwidth reduction is greater than the FLOP:word ratio then the algorithm will be compute bound.[1]

- What is the FLOP:word ratio?
  - ratio of floating point operations (FLOPs) performed to memory bandwidth required expressed as a number of elements (words)

Compute Bound Matrix Multiply

Theorem

If the bandwidth reduction is greater than the FLOP:word ratio then the algorithm will be compute bound.[1]

- What is the FLOP:word ratio?
  - ratio of floating point operations (FLOPs) performed to memory bandwidth required expressed as a number of elements (words)
- Can be worked out using the GPU documentation (GTX 285)

Compute Bound Matrix Multiply

Theorem

If the bandwidth reduction is greater than the FLOP:word ratio then the algorithm will be compute bound.[1]

- What is the FLOP:word ratio?
  - ratio of floating point operations (FLOPs) performed to memory bandwidth required expressed as a number of elements (words)
- Can be worked out using the GPU documentation (GTX 285)
  - Throughput (FLOPs):
    30 SMs $\times$ 1.476GHz $\times$ 16 operations per clock cycle $=$
    708.48GFLOPs/s

Compute Bound Matrix Multiply

Theorem

If the bandwidth reduction is greater than the FLOP:word ratio then the algorithm will be compute bound.[1]

- What is the FLOP:word ratio?
  - ratio of floating point operations (FLOPs) performed to memory bandwidth required expressed as a number of elements (words)
- Can be worked out using the GPU documentation (GTX 285)
  - Throughput (FLOPs):
    30 SMs $\times$ 1.476GHz $\times$ 16 operations per clock cycle $=$
    708.48GFLOPs/s
  - Bandwidth (words):
    512bit memory interface $\times$ 2.484GHz$/$32bits per word $=$
    39.744×10^9words/s

Compute Bound Matrix Multiply

Theorem

If the bandwidth reduction is greater than the FLOP:word ratio then the algorithm will be compute bound.[1]

- What is the FLOP:word ratio?
  - ratio of floating point operations (FLOPs) performed to memory bandwidth required expressed as a number of elements (words)
- Can be worked out using the GPU documentation (GTX 285)
  - Throughput (FLOPs):
    30 SMs $\times$ 1.476GHz $\times$ 16 operations per clock cycle $=$
    708.48GFLOPs/s
  - Bandwidth (words):
    512bit memory interface $\times$ 2.484GHz$/$32bits per word $=$
    39.744×10^9words/s
  - FLOP:word ratio: $\frac{708.48}{39.744} = 17.82$
- Choosing $mb = 64$ and $nb = 16$ gives a bandwidth reduction of $25.6\times$

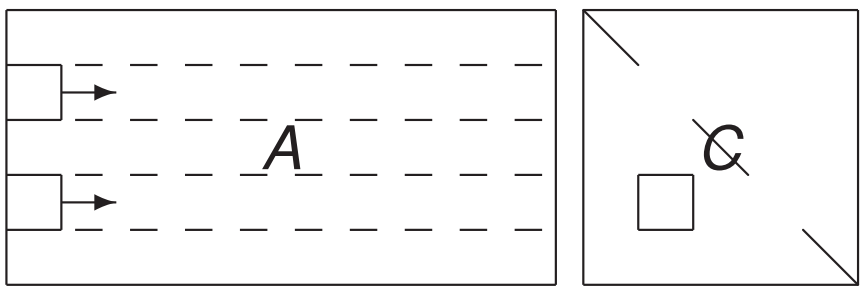
GPU Symmetric Rank-K Update

$$C = \alpha A A^T + \beta C$$

GPU Symmetric Rank-K Update

$$C = \alpha A A^T + \beta C$$

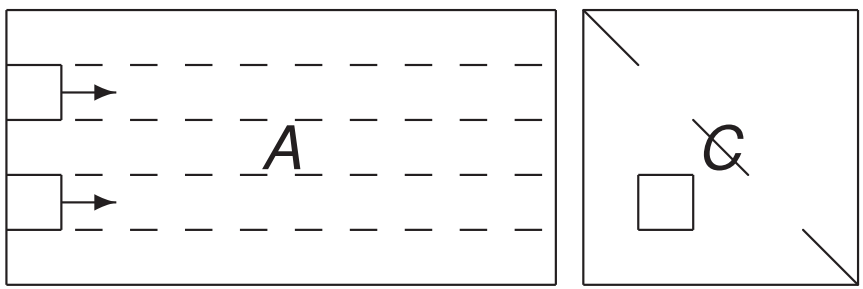
Similar to matrix multiplication with $B = A^T$ except only the lower half of C is written to



GPU Symmetric Rank-K Update

$$C = \alpha A A^T + \beta C$$

Similar to matrix multiplication with $B = A^T$ except only the lower half of C is written to



Can use the same code modified so that thread blocks strictly above the diagonal exit early and those on the diagonal only write to the lower half

GPU Triangular Solve

Solves $XA^T = \alpha B$ by calculating

$$B = \alpha B(A^{-1})^T$$

where X overwrites B

GPU Triangular Solve

Solves $XA^T = \alpha B$ by calculating

$$B = \alpha B(A^{-1})^T$$

where X overwrites B

$$B_{i,j} = \alpha B_{i,j} - \sum_{k=0}^{j} A_{k,i} B_{i,k}$$

GPU Triangular Solve

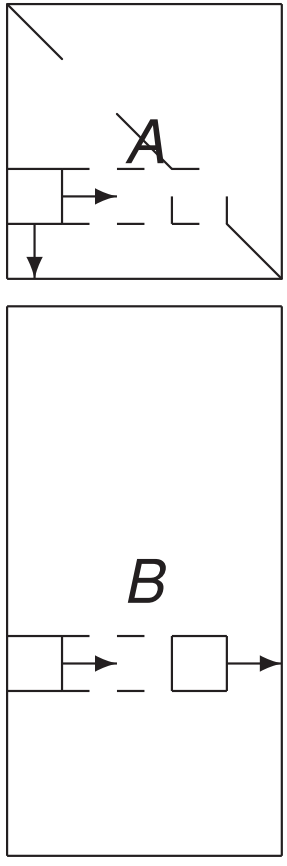
Solves $XA^T = \alpha B$ by calculating

$$B = \alpha B (A^{-1})^T$$

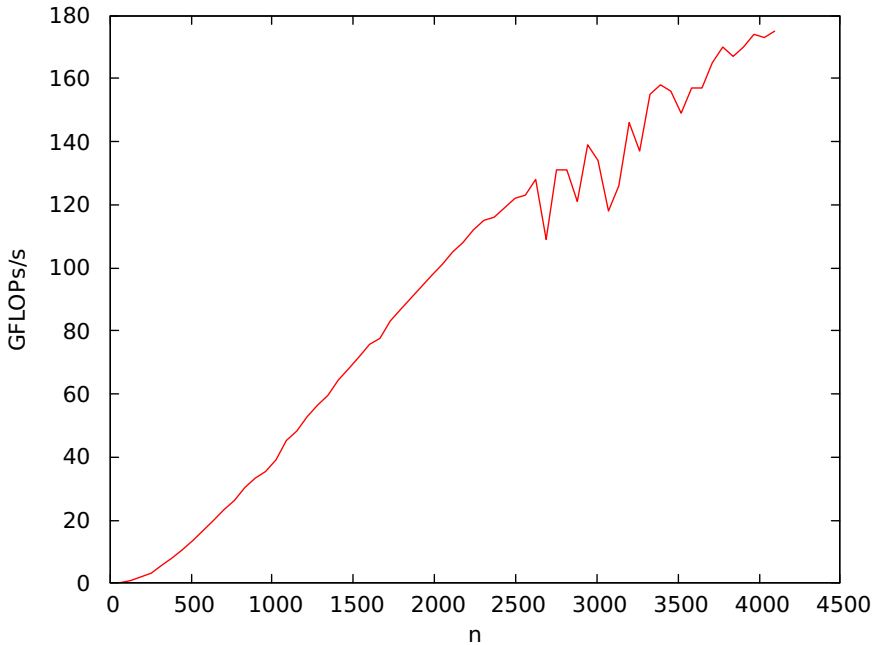
where X overwrites B

$$B_{i,j} = \alpha B_{i,j} - \sum_{k=0}^{j} A_{k,i} B_{i,k}$$

- Each row needs to be updated left-to-right
- Schedule column of thread blocks and use a loop to enforce ordering (slow)



Hybrid Cholesky Decomposition - Results



- Performance reaches 180GFLOPs/s

Replacing Triangular Solve

- Triangular Solve is slow

Replacing Triangular Solve

- Triangular Solve is slow

$$B = \alpha B(A^{-1})^T$$

- Contains inverse (slow) and matrix multiplication (fast)

Replacing Triangular Solve

- Triangular Solve is slow

$$B = \alpha B(A^{-1})^T$$

- Contains inverse (slow) and matrix multiplication (fast)
- Separate into $A = A^{-1}$ and $B = \alpha BA^T$

GPU Triangular Matrix Multiply

$$B = \alpha B A^T$$

- Implementation which updates B in place has similar dependencies to triangular solve (so similar performance)

$$B_{i,j} = \alpha \sum_{k=0}^{j} A_{k,i} B_{k,j}$$

- Elements of B which have **not** yet been calculated are used to update the current element

GPU Triangular Matrix Multiply

$$B = \alpha B A^T$$

- Implementation which updates B in place has similar dependencies to triangular solve (so similar performance)

$$B_{i,j} = \alpha \sum_{k=0}^{j} A_{k,i} B_{k,j}$$

- Elements of B which have **not** yet been calculated are used to update the current element
- In an "out of place" implementation each element is independent

$$X_{i,j} = \alpha \sum_{k=0}^{j} A_{k,i} B_{k,j}$$

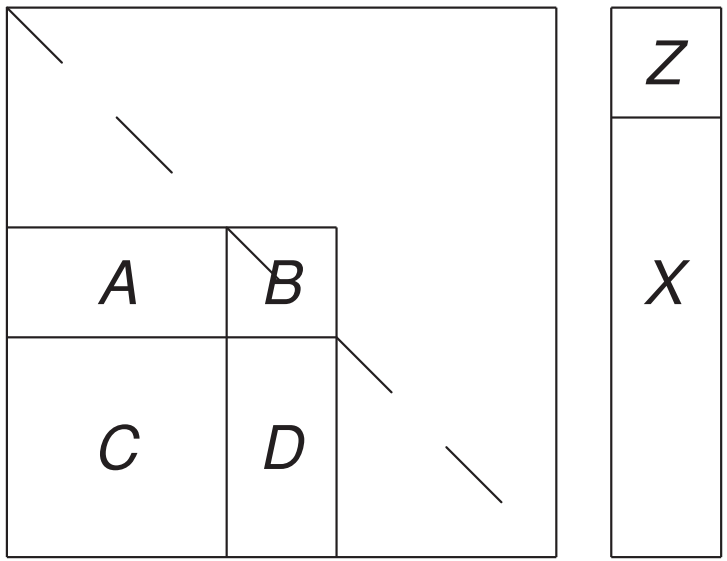
Calculating the Inverse

- Have replaced triangular solve $B = \alpha B(A^{-1})^T$ with triangular multiply $X = \alpha BA^T$
- Now need to form inverse of A

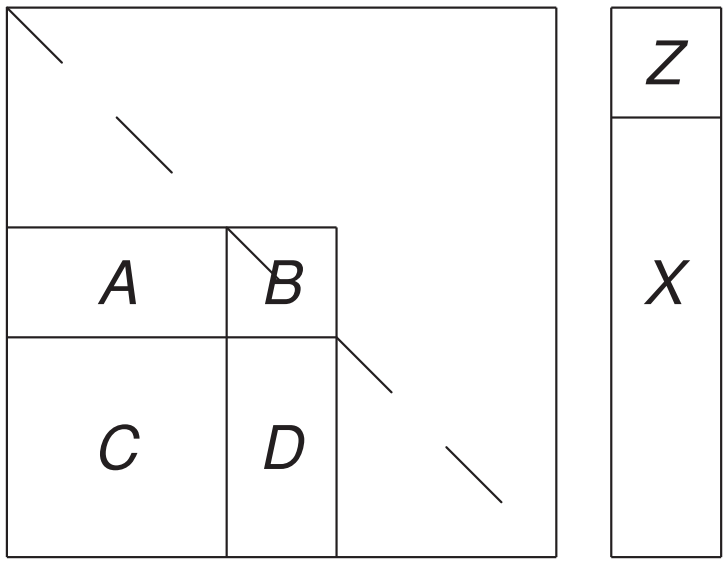
Calculating the Inverse

- Have replaced triangular solve $B = \alpha B(A^{-1})^T$ with triangular multiply $X = \alpha BA^T$
- Now need to form inverse of A
- A is diagonal block in blocked Cholesky decomposition
- Have just computed Cholesky decomposition of A using CPU
- Cholesky decomposition provides faster calculation of inverse
- Calculate inverse of diagonal block A on CPU and copy into temporary block on GPU.

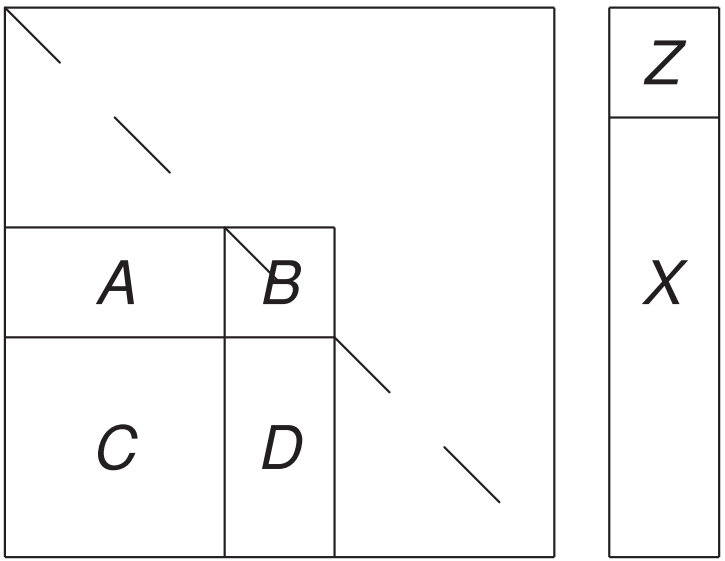
Hybrid Cholesky decomposition without triangular solve



Hybrid Cholesky decomposition without triangular solve

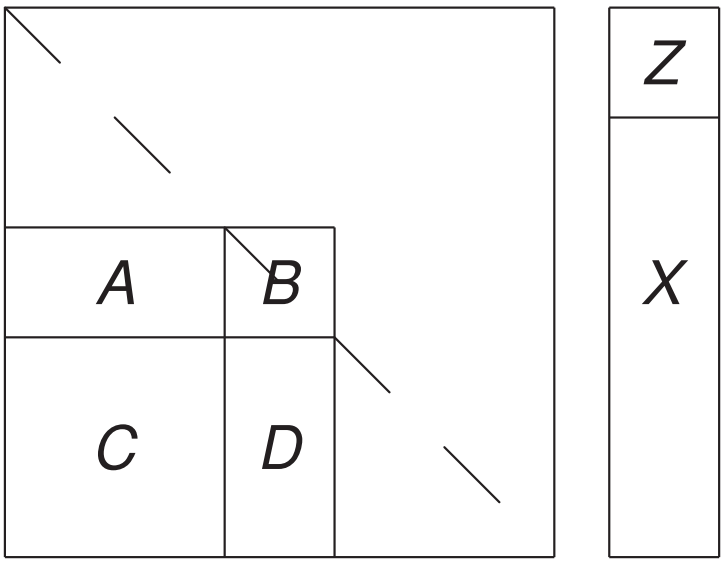


Hybrid Cholesky decomposition without triangular solve



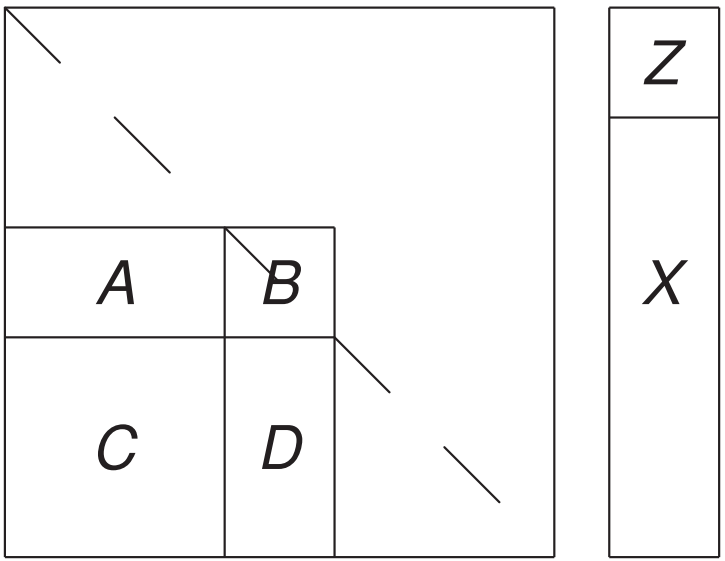
1: $B = B - AA^T$
2: $X = D - CA^T$

Hybrid Cholesky decomposition without triangular solve



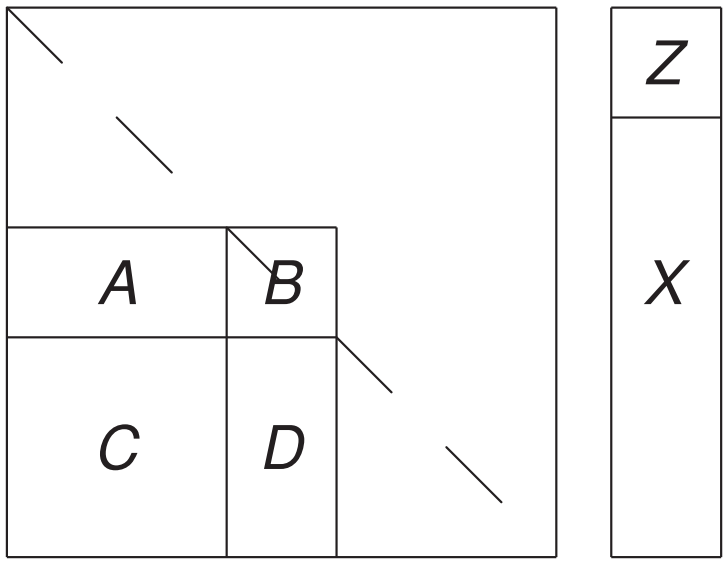
1: $B = B - AA^T$
2: $X = D - CA^T$
3: $B = chol(B)$
4: $Z = B^{-1}$

Hybrid Cholesky decomposition without triangular solve



1: $B = B - AA^T$
2: $X = D - CA^T$
3: $B = chol(B)$
4: $Z = B^{-1}$
5: $D = XZ^T$

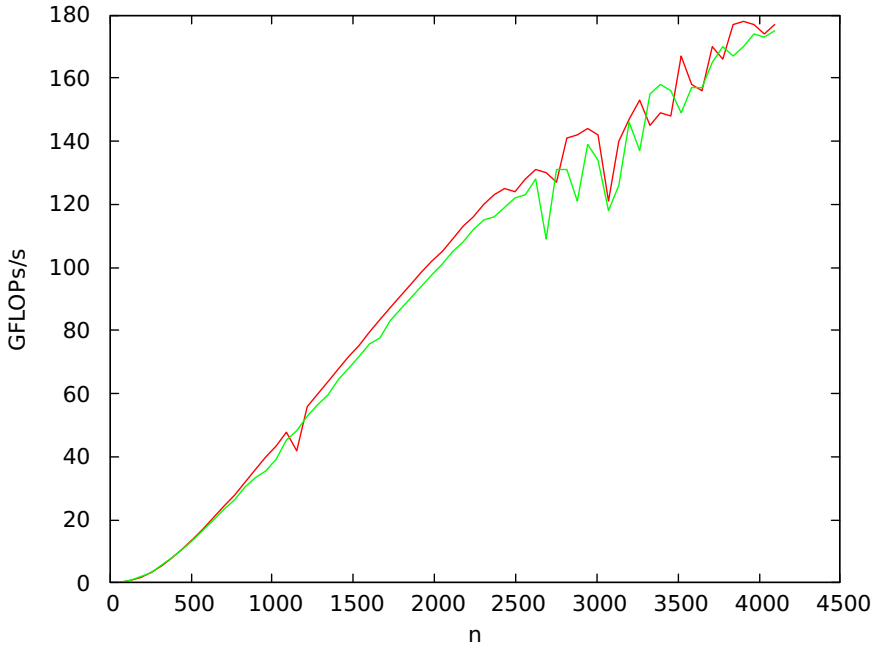
Hybrid Cholesky decomposition without triangular solve



1: $B = B - AA^T$
2: $X = D - CA^T$
3: $B = chol(B)$
4: $Z = B^{-1}$
5: $D = XZ^T$

Use out of place matrix multiply to populate X then triangular multiply to copy back to D

Hybrid Cholesky decomposition without triangular solve



Improving diagonal block transfer

- Each memory copy has overhead

Improving diagonal block transfer

- Each memory copy has overhead

$$t = \frac{n}{bandwidth} + overhead$$

Copying matrices

- Matrices are stored in memory as an array of n columns of m elements (“column major”)
- Each column is padded so that the next column is aligned on a memory boundary
- Submatrices share the same padding as the larger matrix

Copying matrices

- Matrices are stored in memory as an array of n columns of m elements ("column major")
- Each column is padded so that the next column is aligned on a memory boundary
- Submatrices share the same padding as the larger matrix
- Each column must be copied separately

$$t(m,n) = n \times (\frac{m}{bandwidth} + overhead)$$

Copying matrices

- Matrices are stored in memory as an array of n columns of m elements ("column major")
- Each column is padded so that the next column is aligned on a memory boundary
- Submatrices share the same padding as the larger matrix
- Each column must be copied separately

$$t(m,n) = n \times (\frac{m}{bandwidth} + overhead)$$

- If the number of rows is a multiple of the memory alignment there is no padding

$$t(m,n) = \frac{m \times n}{bandwidth} + overhead$$

Copying matrices

- Matrices are stored in memory as an array of n columns of m elements ("column major")
- Each column is padded so that the next column is aligned on a memory boundary
- Submatrices share the same padding as the larger matrix
- Each column must be copied separately

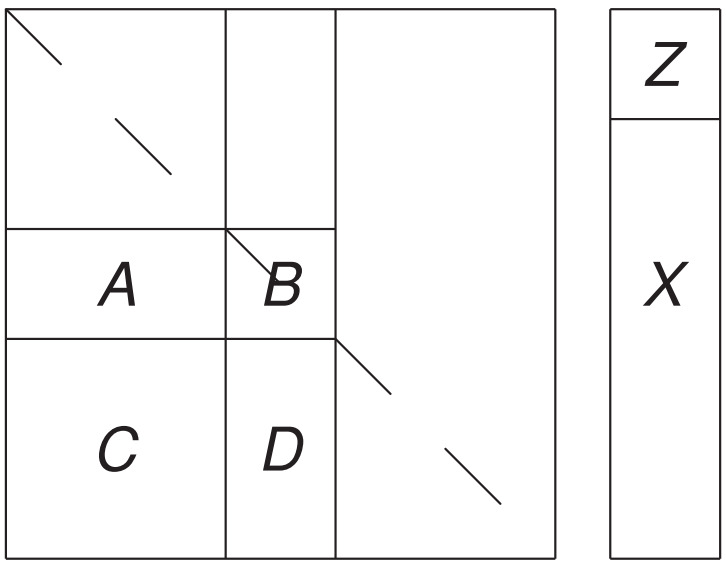
$$t(m,n) = n \times (\frac{m}{bandwidth} + overhead)$$

- If the number of rows is a multiple of the memory alignment there is no padding

$$t(m,n) = \frac{m \times n}{bandwidth} + overhead$$

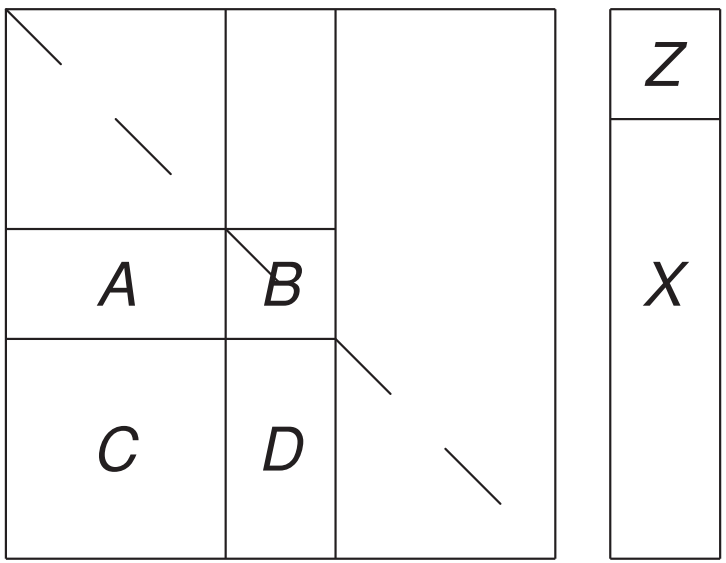
- But submatrices are always padded

Block Column Copy



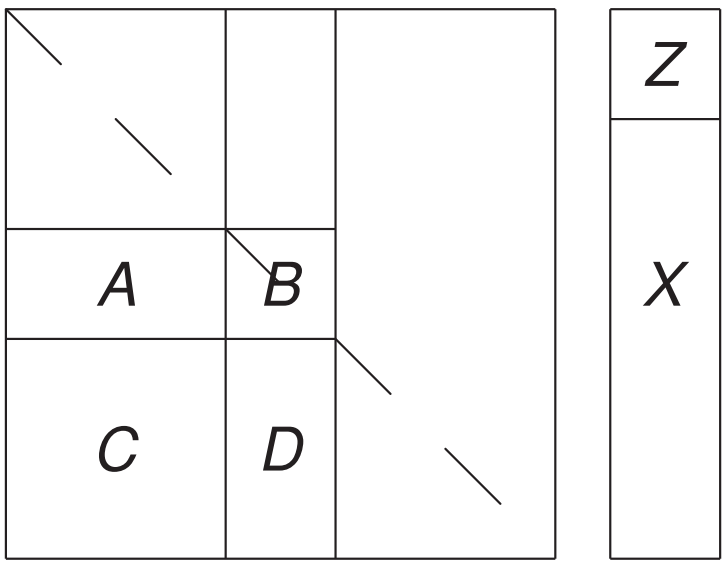
Block Column Copy

- Define column around B

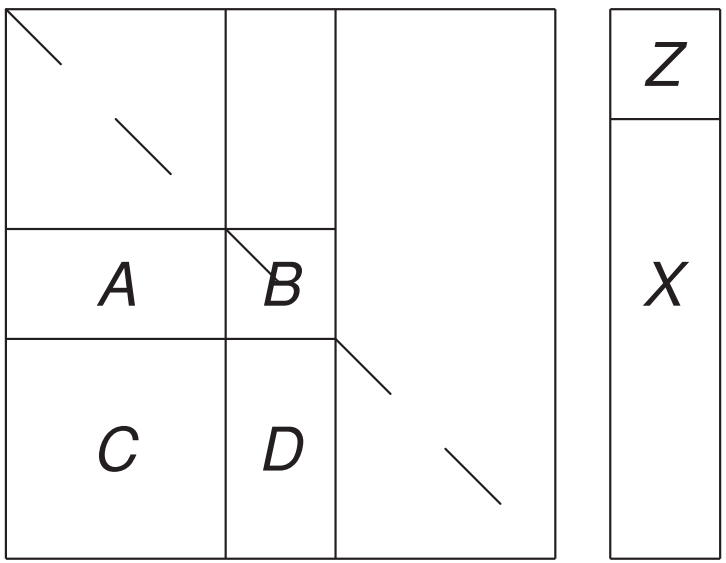


Block Column Copy

- Define column around B
  - No padding when n is a multiple of the memory alignment



Block Column Copy

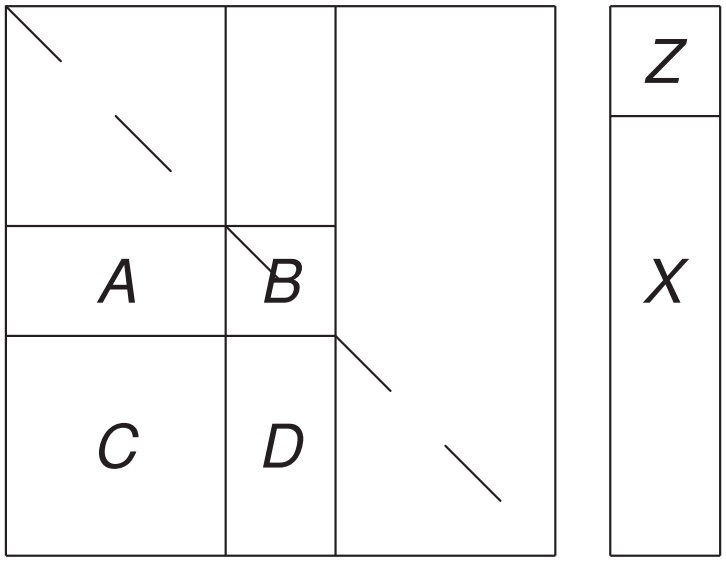


- Define column around B
  - No padding when n is a multiple of the memory alignment
- Faster to copy (larger) column when

$$\frac{n \times nb}{bandwidth} + overhead <$$

$$nb \times (\frac{nb}{bandwidth} + overhead)$$

Block Column Copy



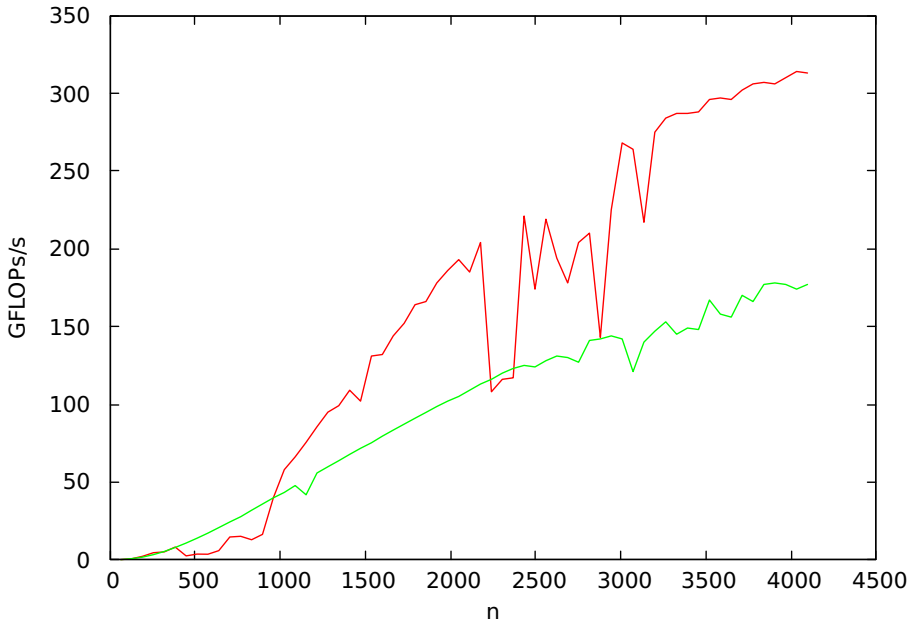
- Define column around B
  - No padding when n is a multiple of the memory alignment
- Faster to copy (larger) column when

$$\frac{n \times nb}{bandwidth} + overhead <$$

$$nb \times (\frac{nb}{bandwidth} + overhead)$$

- Don't have to worry about overwriting updated D as matrix multiply is now out of place

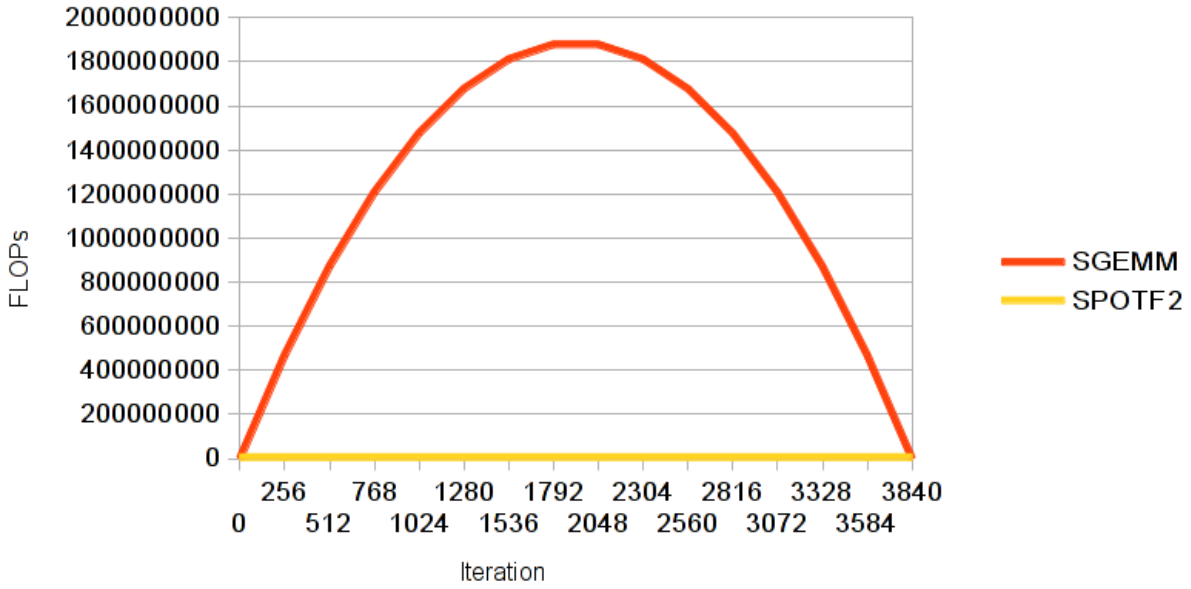
Block Column Copy - results



Tuning the block size

- For CPU blocked algorithms the block size is chosen so that blocks fit in the CPU cache
- Introduced an extra level of blocking for hybrid algorithm
- Aim to choose block size so that workload is balanced between computing devices

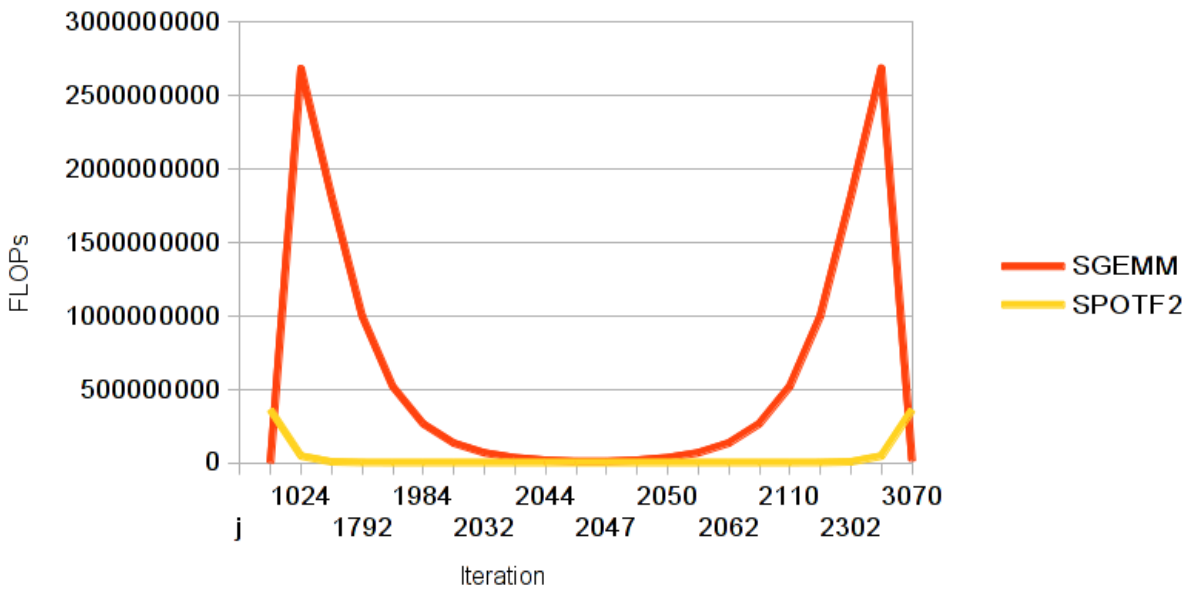
Static block size



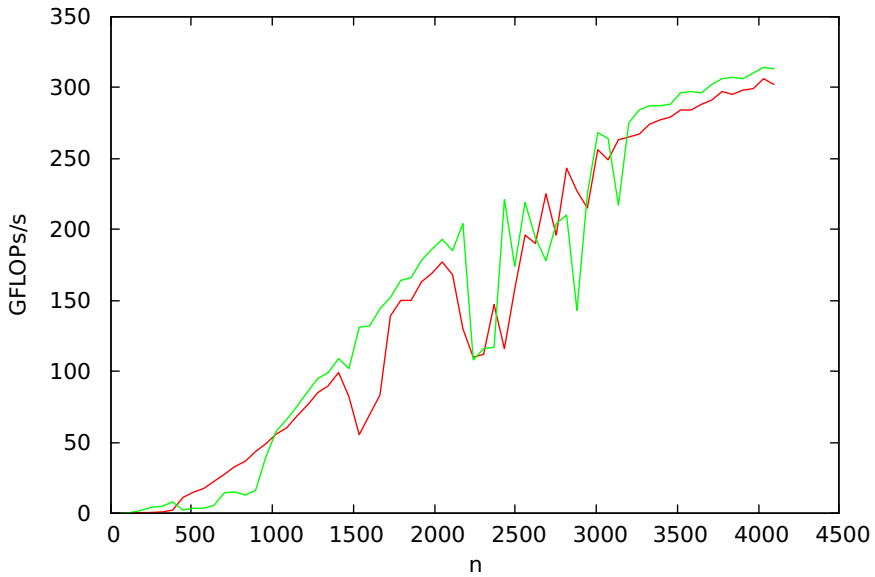
- Aim to minimise area between two curves

Dynamic block size

- Can change block size on each iteration and still have a correct algorithm



Dynamic block size - results



GPU Cholesky decomposition

- Now that the block size changes on each iteration it can get too small for the matrix multiply to sufficiently overlap the $2\times$ copy and Cholesky decomposition on the CPU

GPU Cholesky decomposition

- Now that the block size changes on each iteration it can get too small for the matrix multiply to sufficiently overlap the $2\times$ copy and Cholesky decomposition on the CPU
- Implement unblocked Cholesky decomposition for the GPU

GPU Cholesky decomposition

- Now that the block size changes on each iteration it can get too small for the matrix multiply to sufficiently overlap the $2\times$ copy and Cholesky decomposition on the CPU
- Implement unblocked Cholesky decomposition for the GPU
  - Due to data dependencies only runs on one SM so that threads can share results
  - Uses triangular packed storage to make most efficient use of shared memory and get large number of threads

GPU Cholesky decomposition

- Now that the block size changes on each iteration it can get too small for the matrix multiply to sufficiently overlap the $2\times$ copy and Cholesky decomposition on the CPU
- Implement unblocked Cholesky decomposition for the GPU
  - Due to data dependencies only runs on one SM so that threads can share results
  - Uses triangular packed storage to make most efficient use of shared memory and get large number of threads
- GPU is already performing matrix multiply

GPU Cholesky decomposition

- Now that the block size changes on each iteration it can get too small for the matrix multiply to sufficiently overlap the $2\times$ copy and Cholesky decomposition on the CPU
- Implement unblocked Cholesky decomposition for the GPU
  - Due to data dependencies only runs on one SM so that threads can share results
  - Uses triangular packed storage to make most efficient use of shared memory and get large number of threads
- GPU is already performing matrix multiply
  - Possible to overlap both on the GPU?

Multiple kernels

- nVidia CUDA Programming Guide (2008): *maximum performance occurs when no threads execute divergent branches*

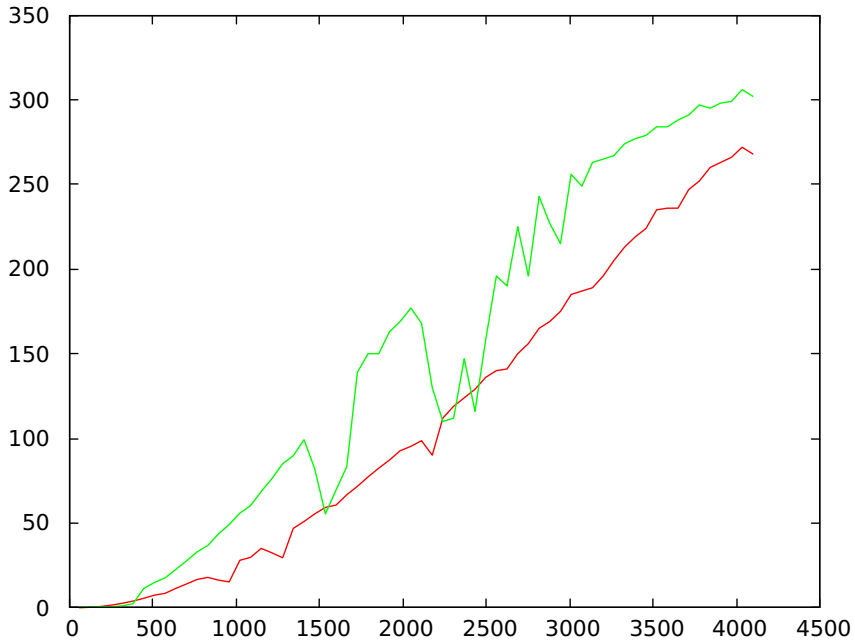
Multiple kernels

- nVidia CUDA Programming Guide (2008): *maximum performance occurs when no threads execute divergent branches*
- nVidia CUDA Programming Guide (2011): *maximum performance occurs when no threads within the same warp execute divergent branches*

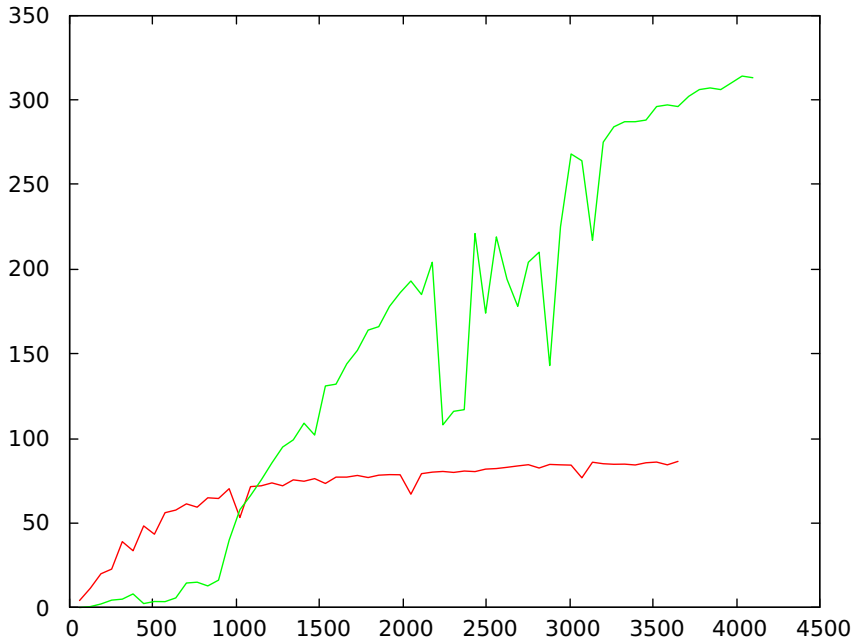
Multiple kernels

- nVidia CUDA Programming Guide (2008): *maximum performance occurs when no threads execute divergent branches*
- nVidia CUDA Programming Guide (2011): *maximum performance occurs when no threads within the same warp execute divergent branches*
- Write combined matrix multiply and unblocked Cholesky decomposition kernel
  - First $n - 1$ thread blocks perform matrix multiplication
  - Thread block n performs Cholesky decomposition

Multiple Kernels - results



Final Results



Conclusions

- Possible to get large speed improvements for inherently sequential algorithms such as the Cholesky decomposition by carefully considering the structure of the algorithm and the type of operations performed at each step.

Conclusions

- Possible to get large speed improvements for inherently sequential algorithms such as the Cholesky decomposition by carefully considering the structure of the algorithm and the type of operations performed at each step.
- Having a GPU available allows processing to overlap making maximum use of the available parallelism.

Conclusions

- Possible to get large speed improvements for inherently sequential algorithms such as the Cholesky decomposition by carefully considering the structure of the algorithm and the type of operations performed at each step.
- Having a GPU available allows processing to overlap making maximum use of the available parallelism.
- Different types of parallel workloads can be sent to the most appropriate device.

References

Vasily Volkov and James W. Demmel.
Benchmarking GPUs to tune dense linear algebra.
In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.