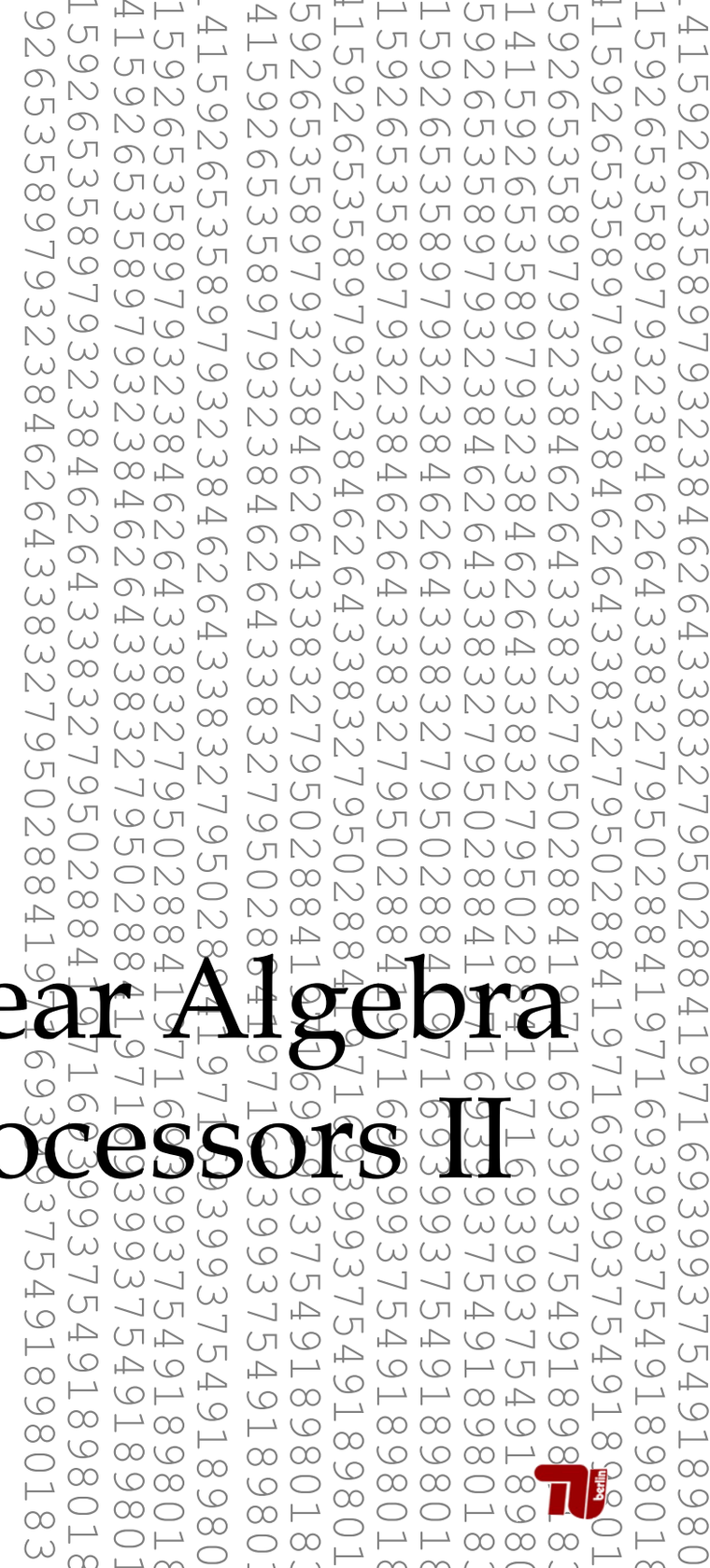
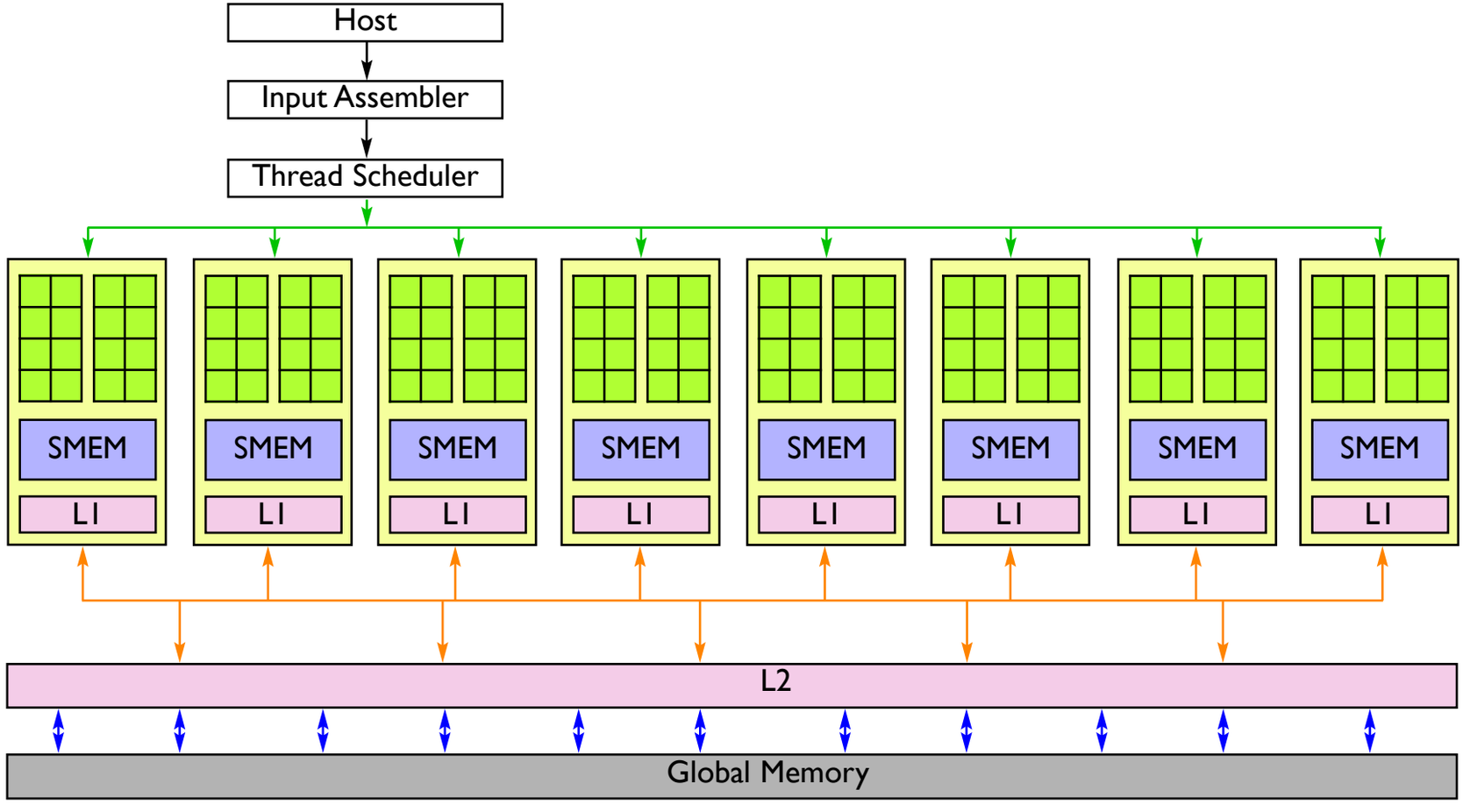
High Performance Linear Algebra on Data Parallel Co-Processors II



UNIVERSITY of TORONTO

Data parallel co-processors



Memory latency

GPU	8800GTX	8600GTS	9800GTX	GTX280
at pins, GB/s	86	32	70	141
aligned copy	89%	83%	85%	89%
misaligned	9%	10%	9%	51%
stride-2	9%	10%	9%	45%
stride-10	10%	10%	9%	10%
stride-1000	0.9%	2.1%	1.1%	1.1%

Volkov, V., and J. W. Demmel. *Benchmarking GPUs to Tune Dense Linear Algebra.* 2008.

Memory latency

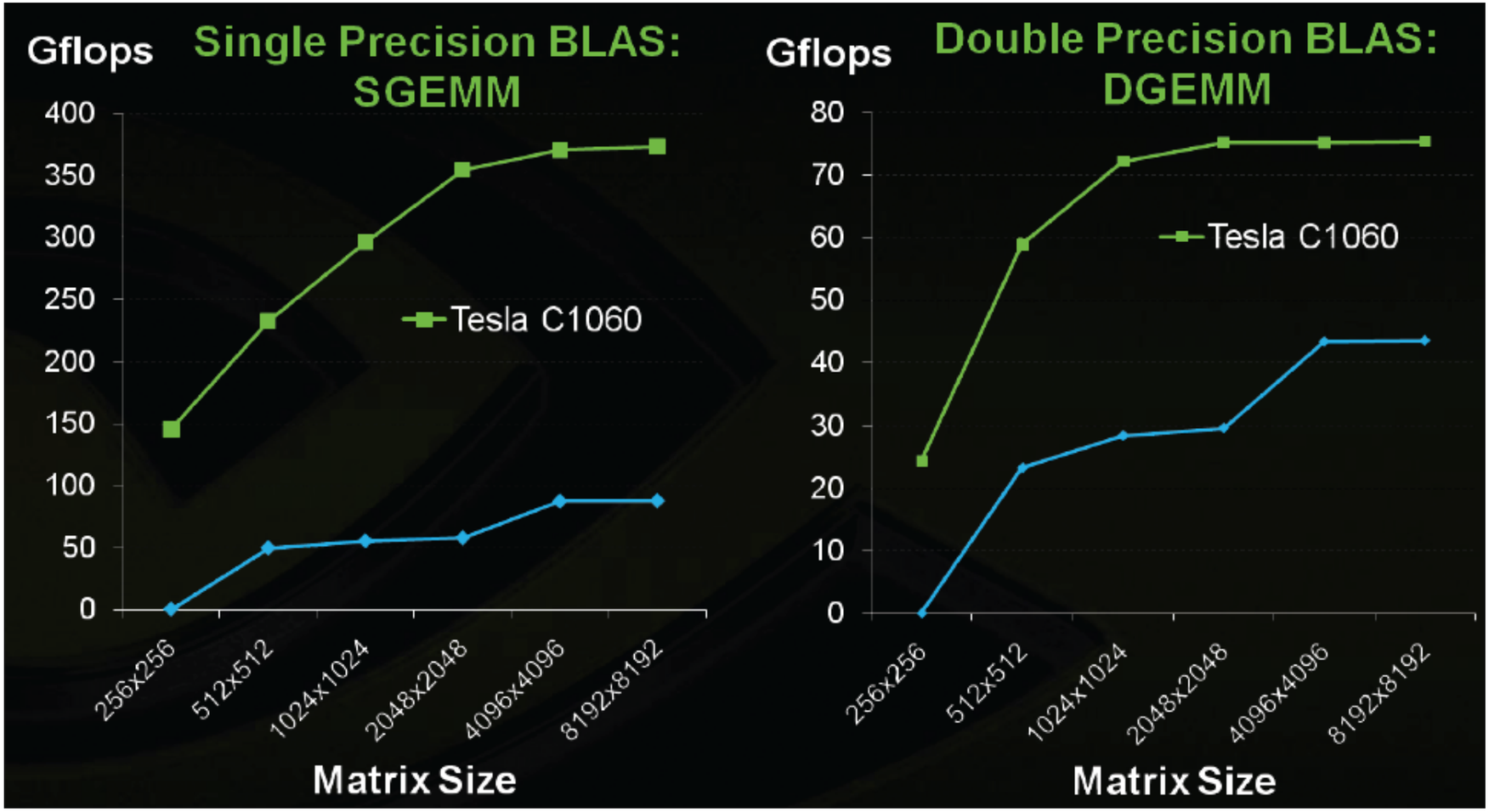
GPU	8800GTX	8600GTS	9800GTX	GTX280
at pins, GB/s	86	32	70	141
aligned copy	89%	83%	85%	89%
misaligned	9%	10%	9%	51%
stride-2	9%	10%	9%	45%
stride-10	10%	10%	9%	10%
stride-1000	0.9%	2.1%	1.1%	1.1%

Volkov, V., and J. W. Demmel. *Benchmarking GPUs to Tune Dense Linear Algebra*. 2008.

Linear algebra libraries

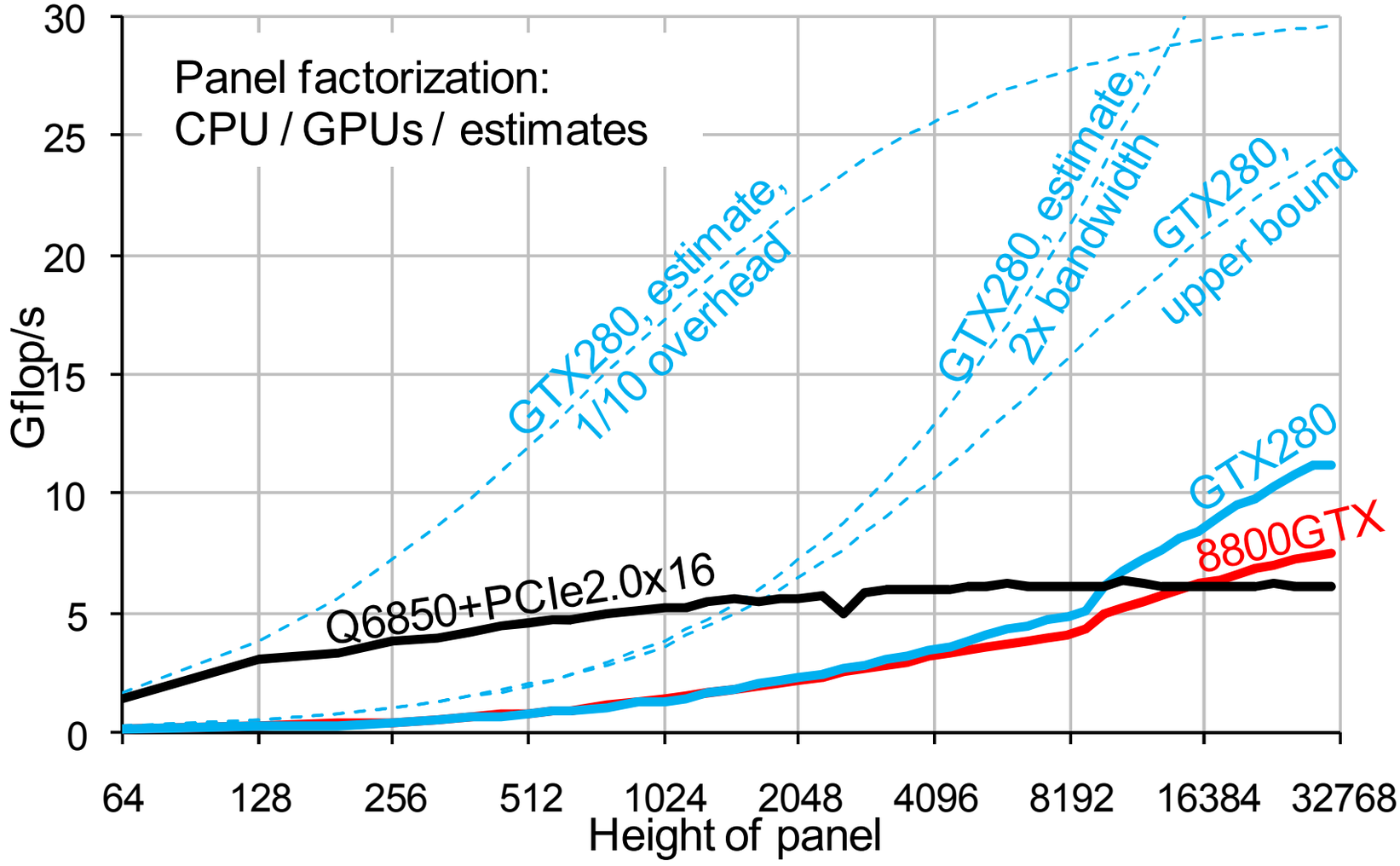
- **CUBLAS**: BLAS implementation for CUDA.
- **CUSparse**: Sparse matrix operations.
- **CUFFT**: FFT implementation for CUDA.
- **CULA**: Lapack implementation for CUDA.
- **CURand**: (Quasi) random number generation.
- **NAG GPU library**: NAG for GPUs.
- **Thrust**: STL style library for common functionality (sort, reduce, ...).

BLAS performance



http://gpgpu.org/wp/wp-content/uploads/2009/11/SC09_CUDA_Tools_Cohen.pdf

LU factorization using BLAS

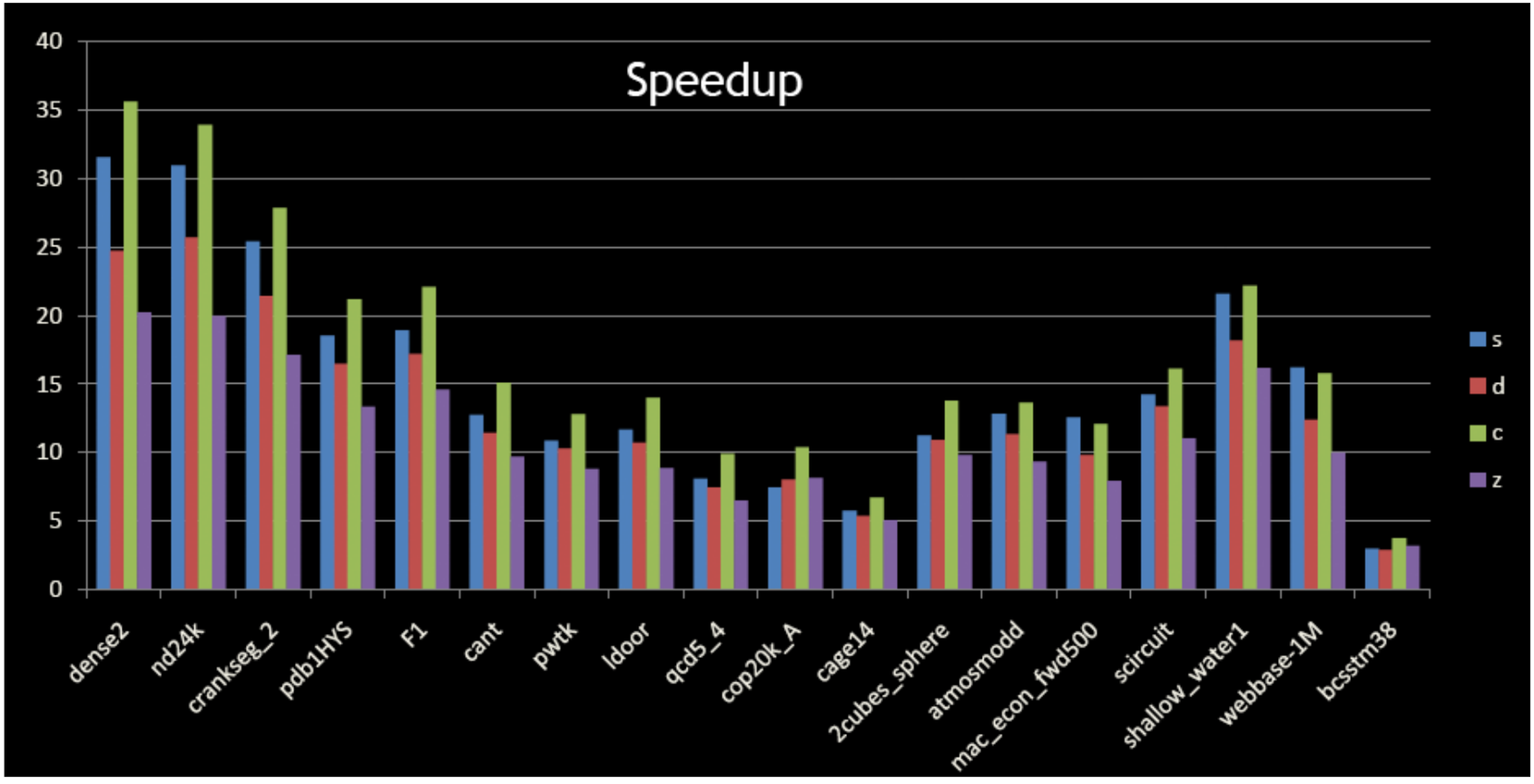


Volkov, V., and J. W. Demmel. *Benchmarking GPUs to Tune Dense Linear Algebra*. 2008.

CUFFT performance

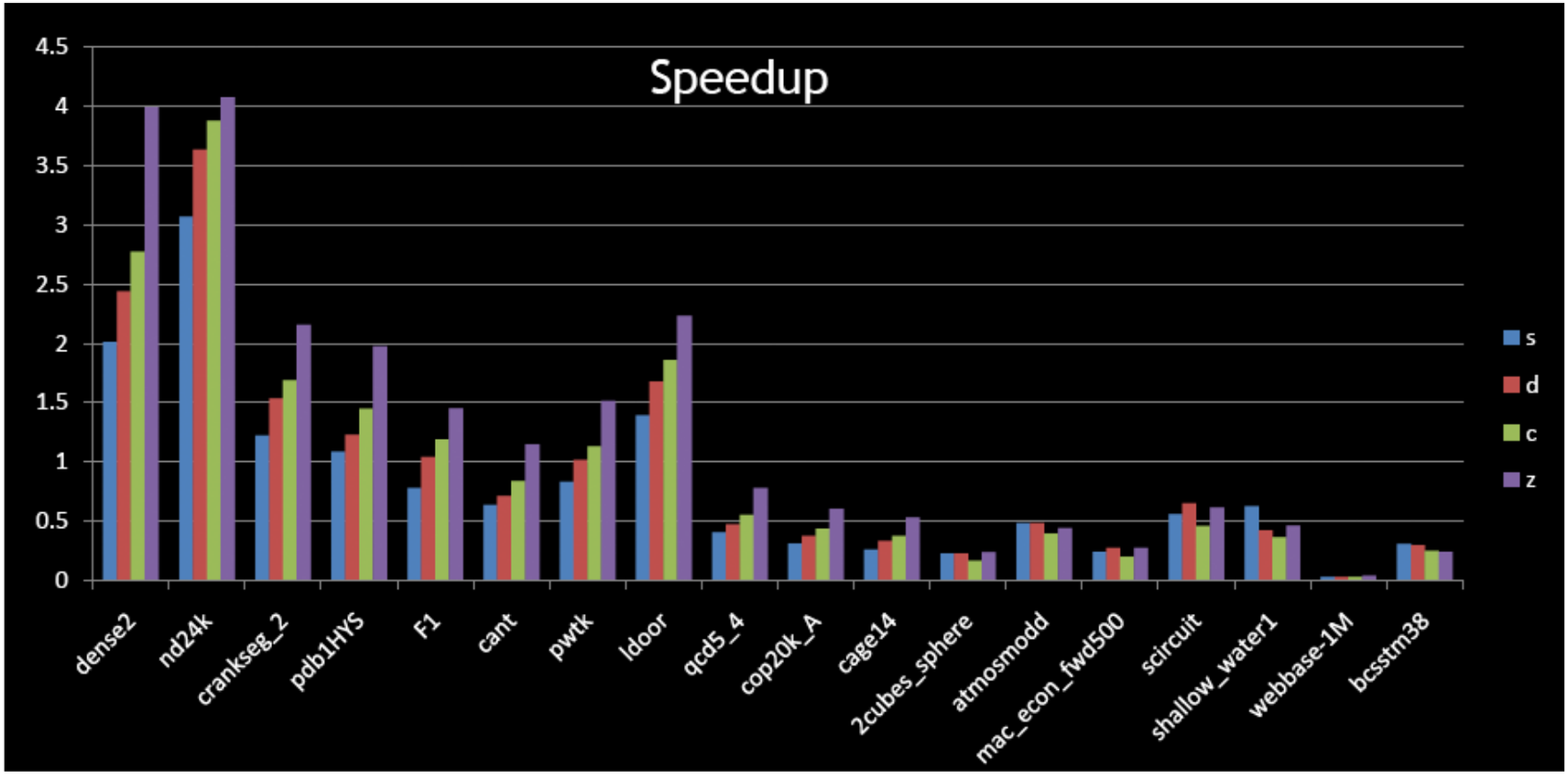
http://gpgpu.org/wp/wp-content/uploads/2009/11/SC09_CUDA_Tools_Cohen.pdf

CUSparse performance



http://www.nvidia.com/content/GTC-2010/pdfs/2070_GTC2010.pdf

CUSparse performance



http://www.nvidia.com/content/GTC-2010/pdfs/2070_GTC2010.pdf

GEMM

- General weighted matrix-matrix multiplication

GEMM

- General weighted matrix-matrix multiplication

$$\mathbf{C} = \alpha\,\mathbf{A}\,\mathbf{B} + \beta\,\mathbf{C}$$

GEMM

- General weighted matrix-matrix multiplication

$$\mathbf{C} = \alpha\,\mathbf{A}\,\mathbf{B} + \beta\,\mathbf{C}$$

- Algorithm has to be **blocked** to avoid being memory bandwidth limited.

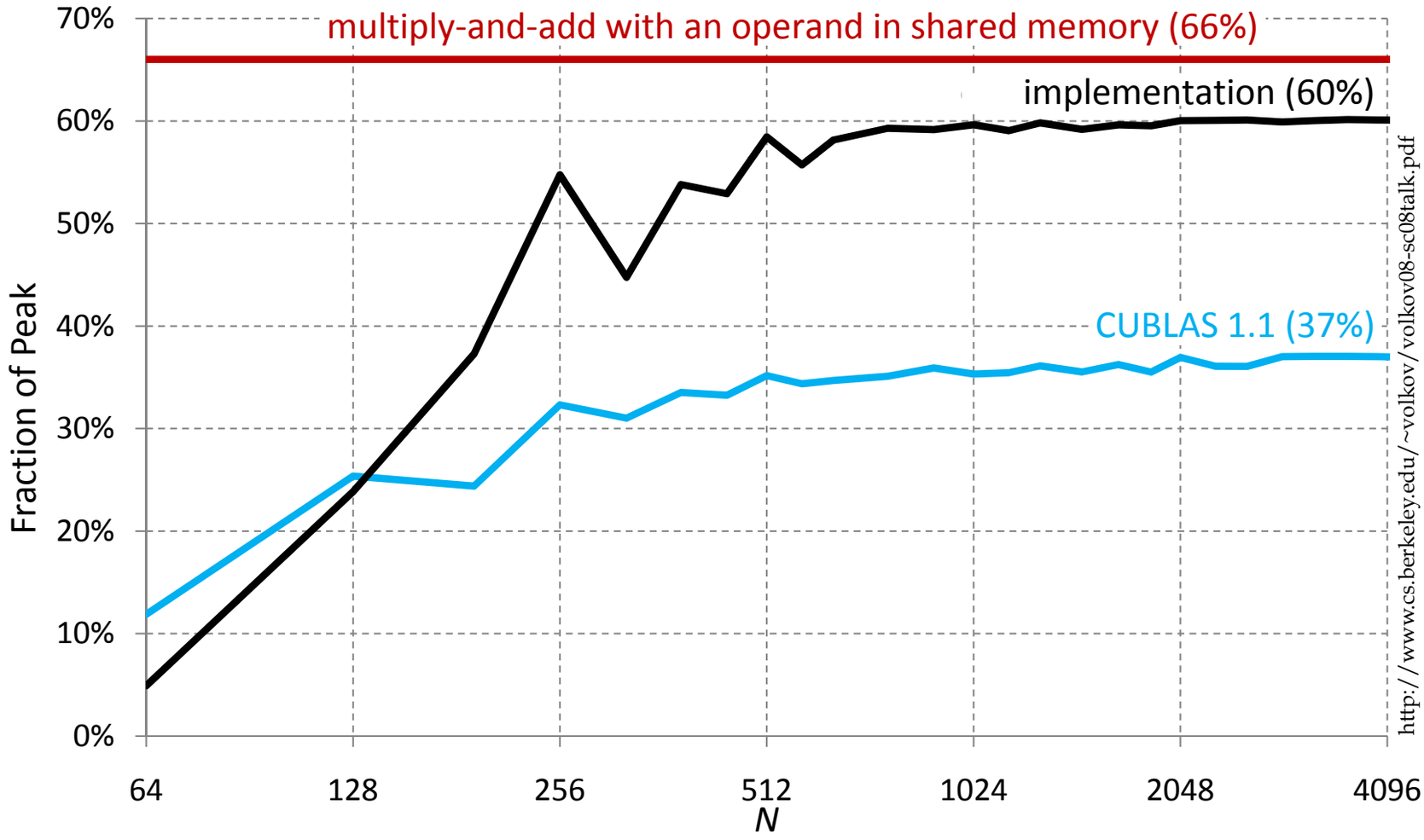
GEMM

- General weighted matrix-matrix multiplication

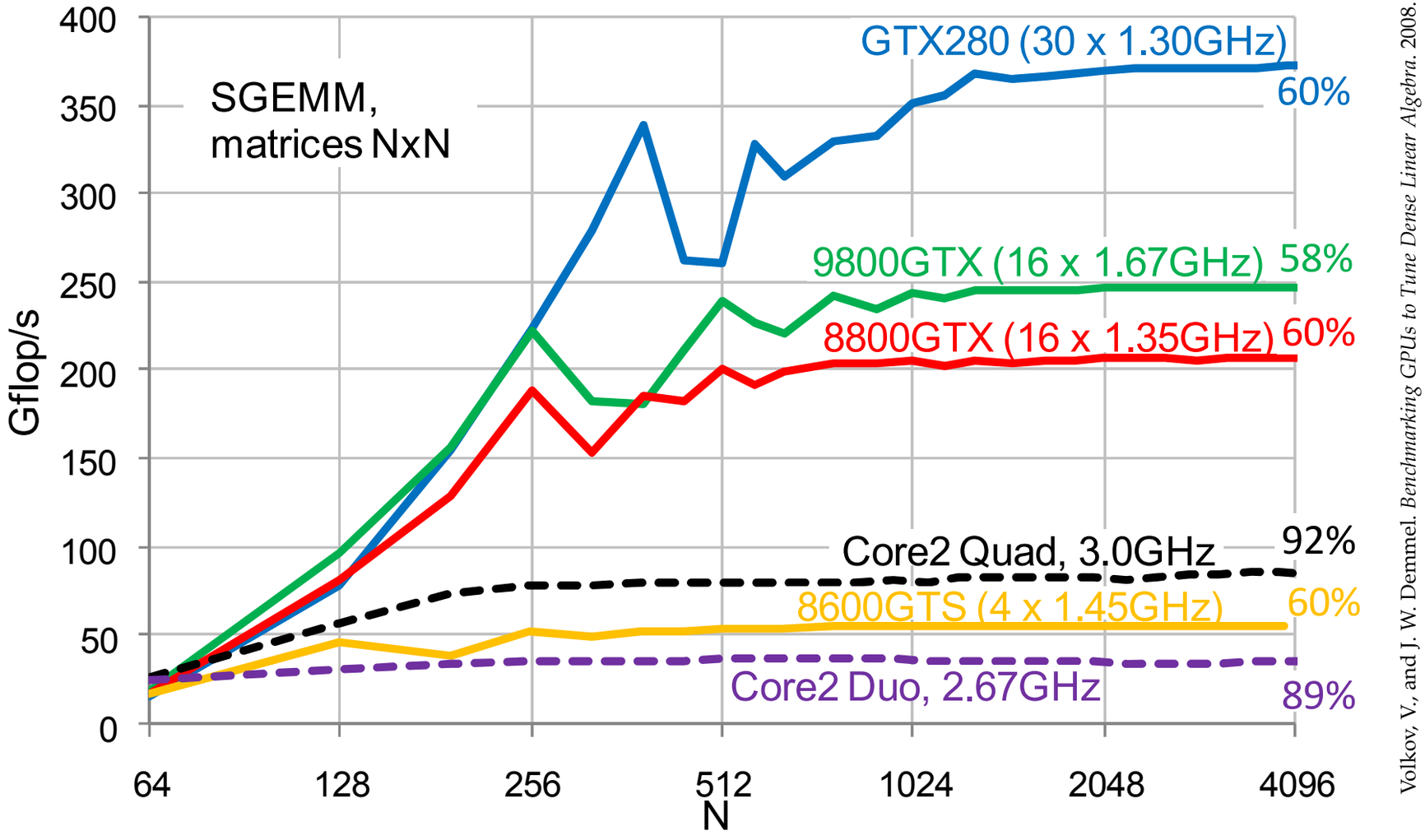
$$\mathbf{C} = \alpha\,\mathbf{A}\,\mathbf{B} + \beta\,\mathbf{C}$$

- Algorithm has to be **blocked** to avoid being memory bandwidth limited.
- Many implementation details have to be aligned with the details of the hardware architecture ...

GEMM


multiply-and-add with an operand in shared memory (66%)
implementation (60%)
CUBLAS 1.1 (37%)
Fraction of Peak
70%
60%
50%
40%
30%
20%
10%
0%
64
128
256
512
1024
2048
4096
N
http://www.cs.berkeley.edu/~volkov/volkov08-sc08talk.pdf

GEMM



Volkov, V., and J. W. Demmel. *Benchmarking GPUs to Tune Dense Linear Algebra*. 2008.

Bisection for eigenvalue spectrum

- **Sought**: all eigenvalues of tridiagonal, symmetric input matrix T.

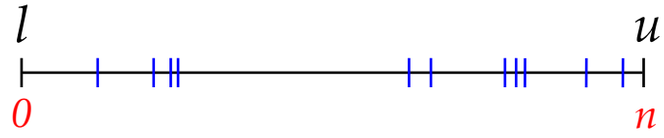
Bisection for eigenvalue spectrum

- **Sought**: all eigenvalues of tridiagonal, symmetric input matrix T.
- **Ingredients**: Gerschgorin interval $G(\mathbf{T}) = [l, u]$ and Sturm counts $C(y) = k$.

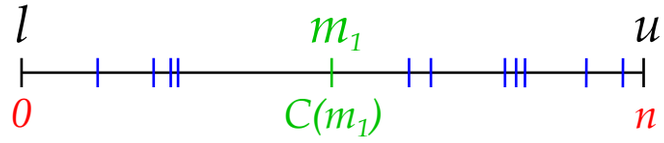
Bisection for eigenvalue spectrum

- **Sought**: all eigenvalues of tridiagonal, symmetric input matrix T.
- **Ingredients**: Gerschgorin interval $G(\mathbf{T}) = [l, u]$ and Sturm counts $C(y) = k$.
- **Recipe**: Bracket eigenvalues starting from the Gerschgorin interval using Sturm counts.

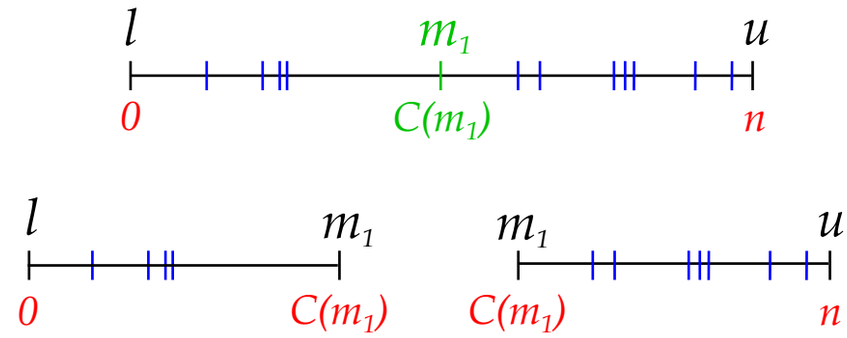
Bisection for eigenvalue spectrum



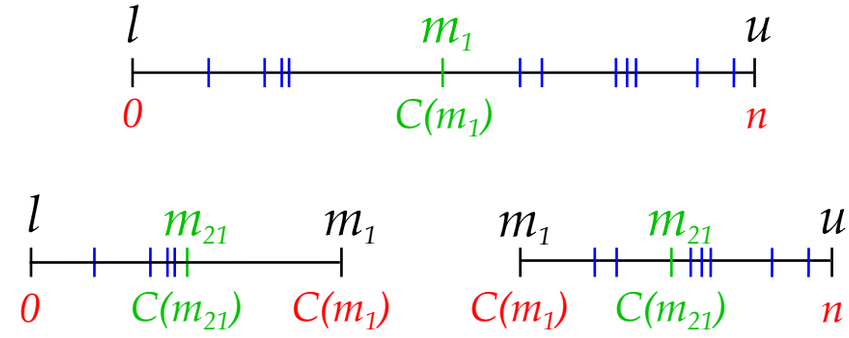
Bisection for eigenvalue spectrum



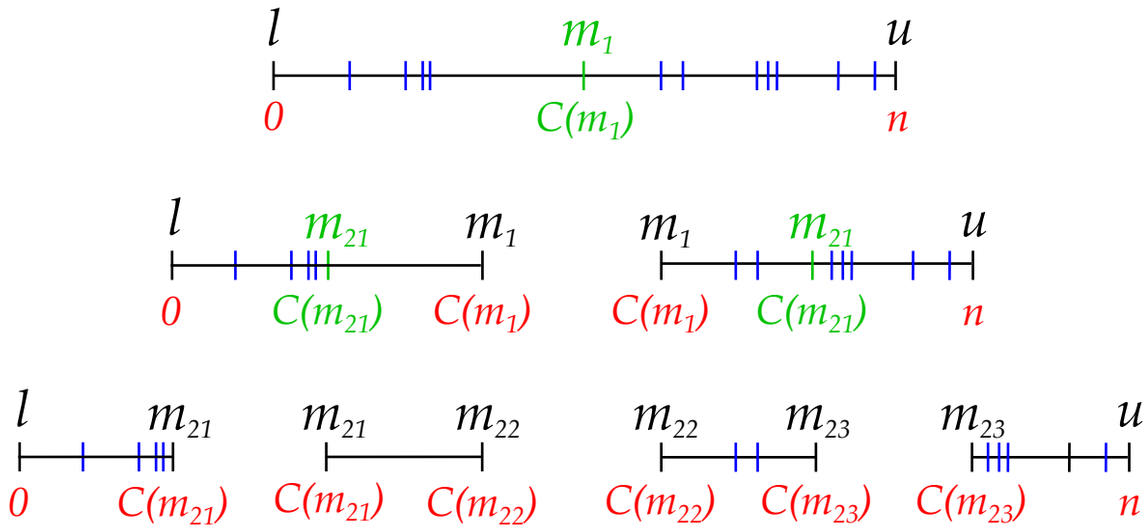
Bisection for eigenvalue spectrum



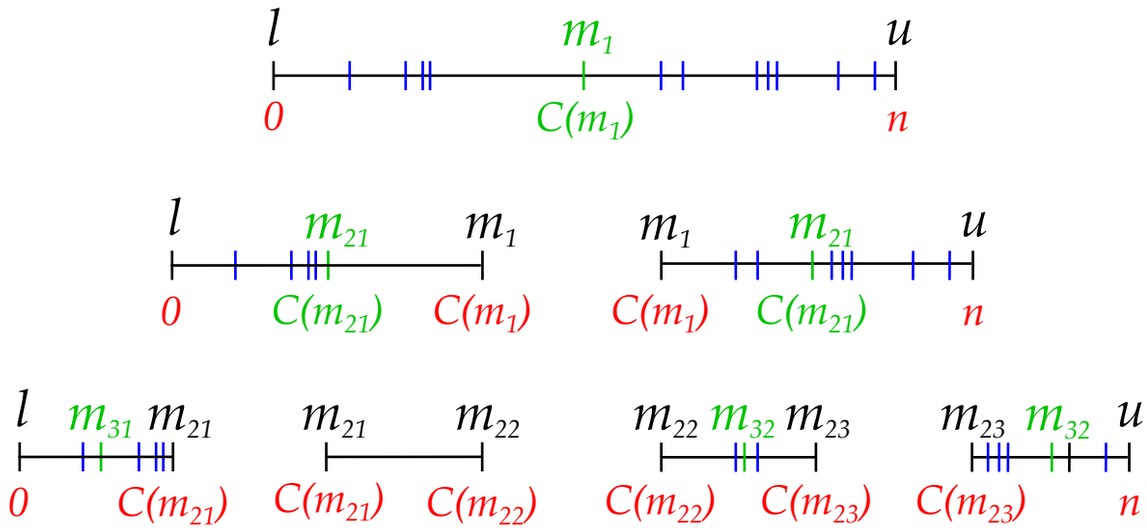
Bisection for eigenvalue spectrum



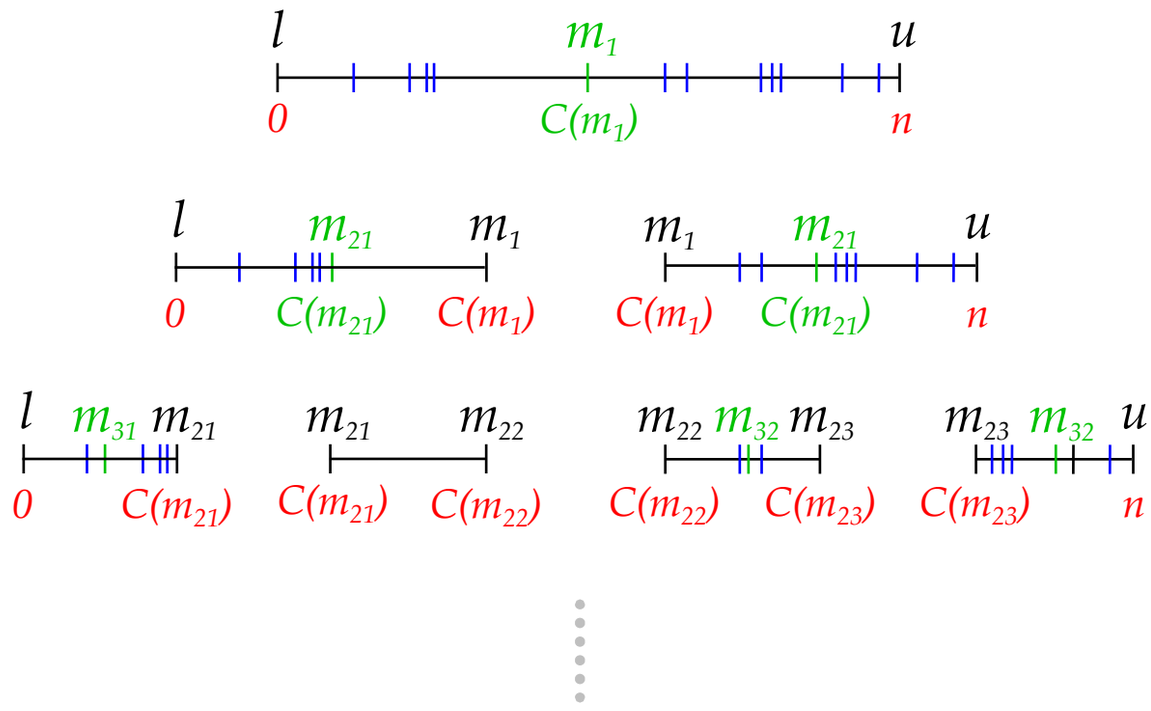
Bisection for eigenvalue spectrum



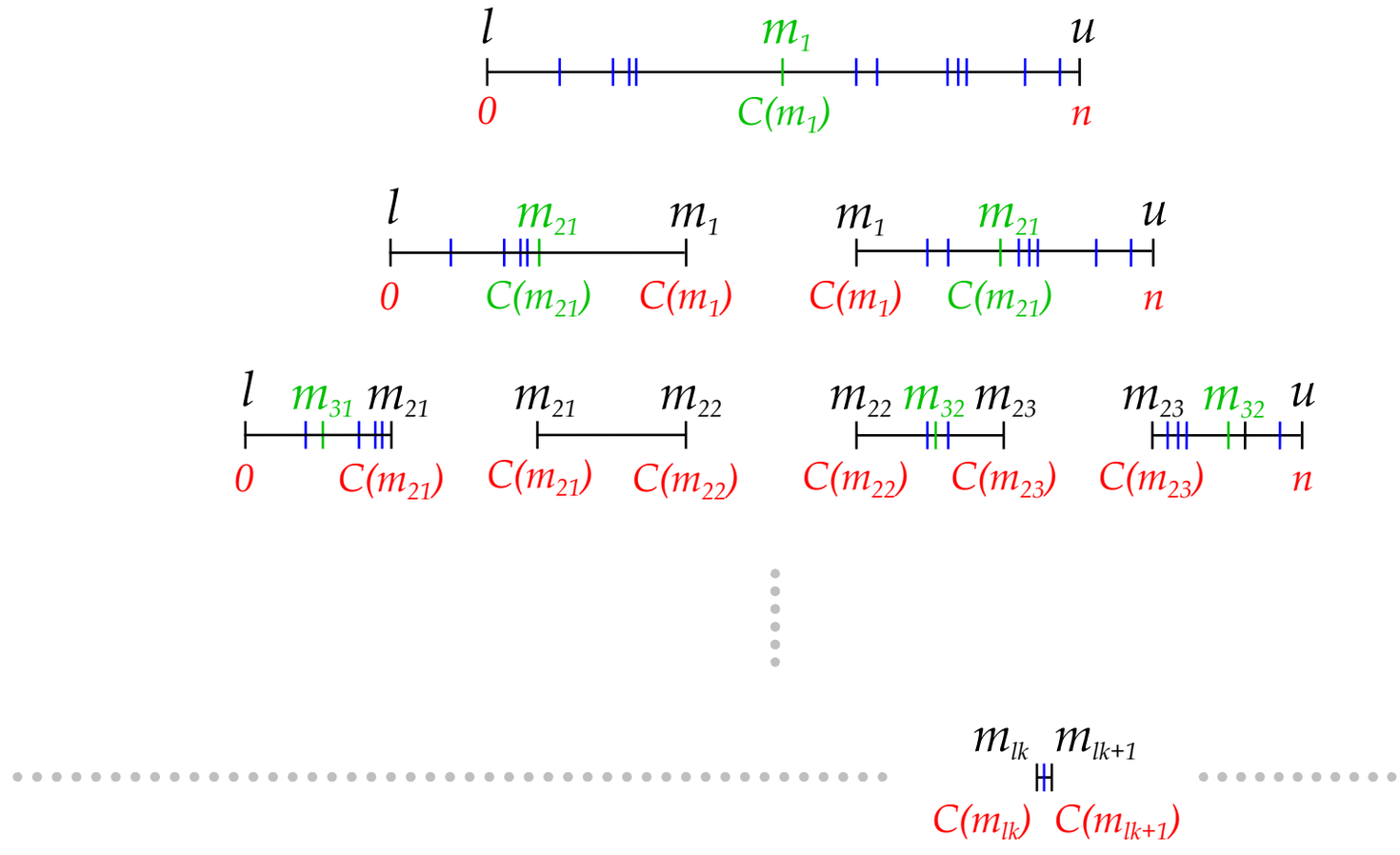
Bisection for eigenvalue spectrum



Bisection for eigenvalue spectrum



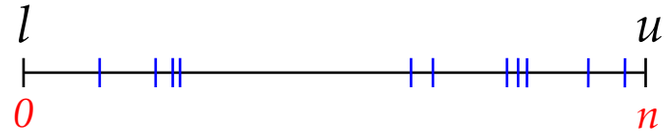
Bisection for eigenvalue spectrum



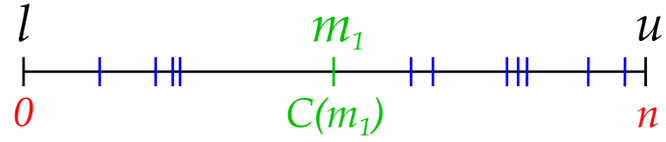
Bisection for eigenvalue spectrum

```
Count = 0;
d = 1;
for i = 1 to n
   d = a_i - x - (b_{i-1}*b_{i-1})/d;
   if d < 0
      Count += 1;
   endif
endfor
```

Bisection for eigenvalue spectrum

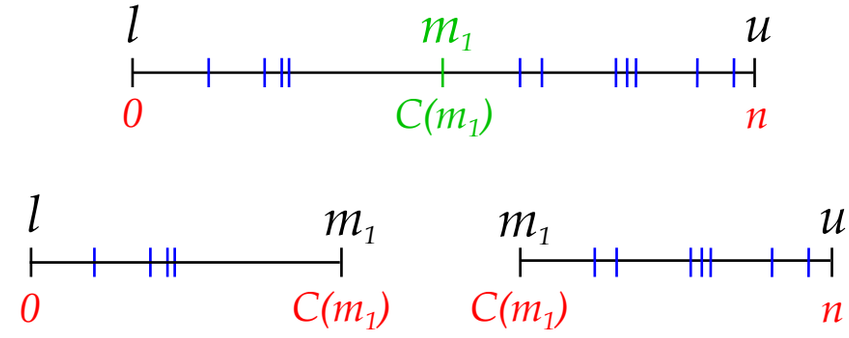


Bisection for eigenvalue spectrum



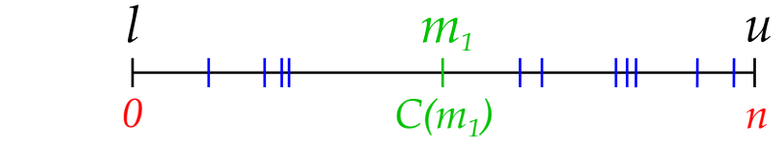
1X

Bisection for eigenvalue spectrum

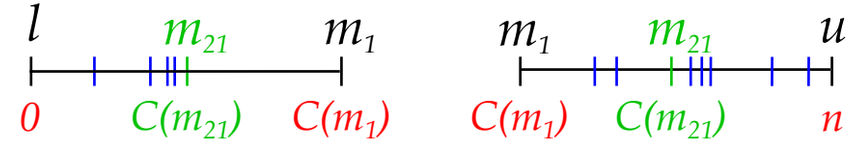


1X

Bisection for eigenvalue spectrum

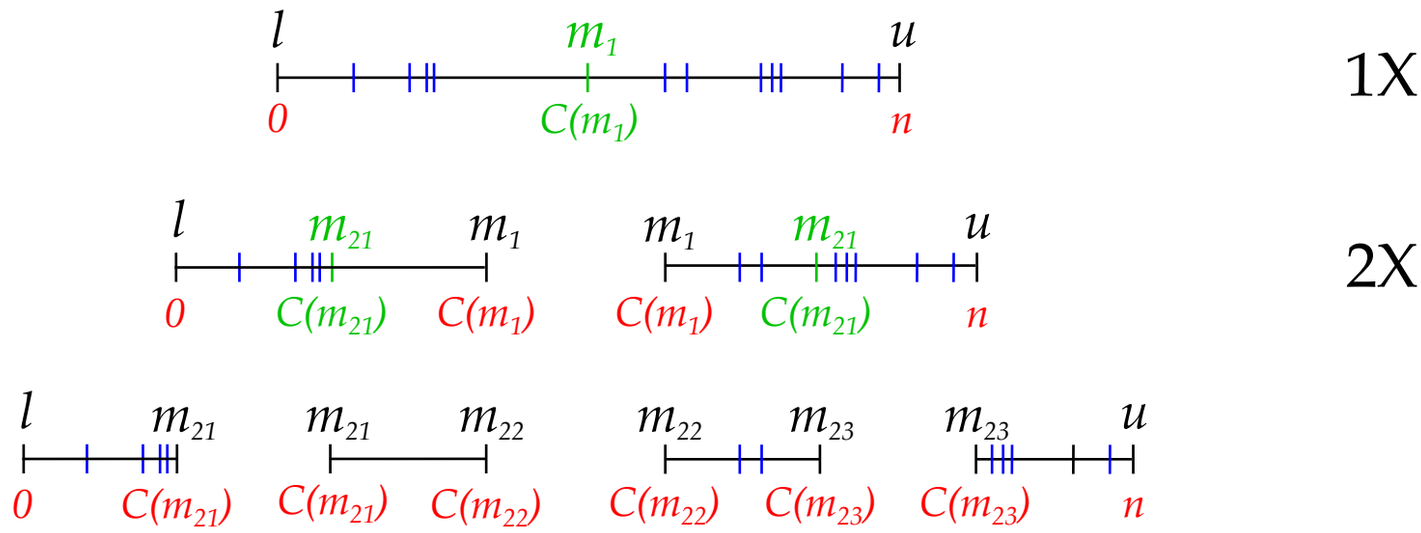


1X

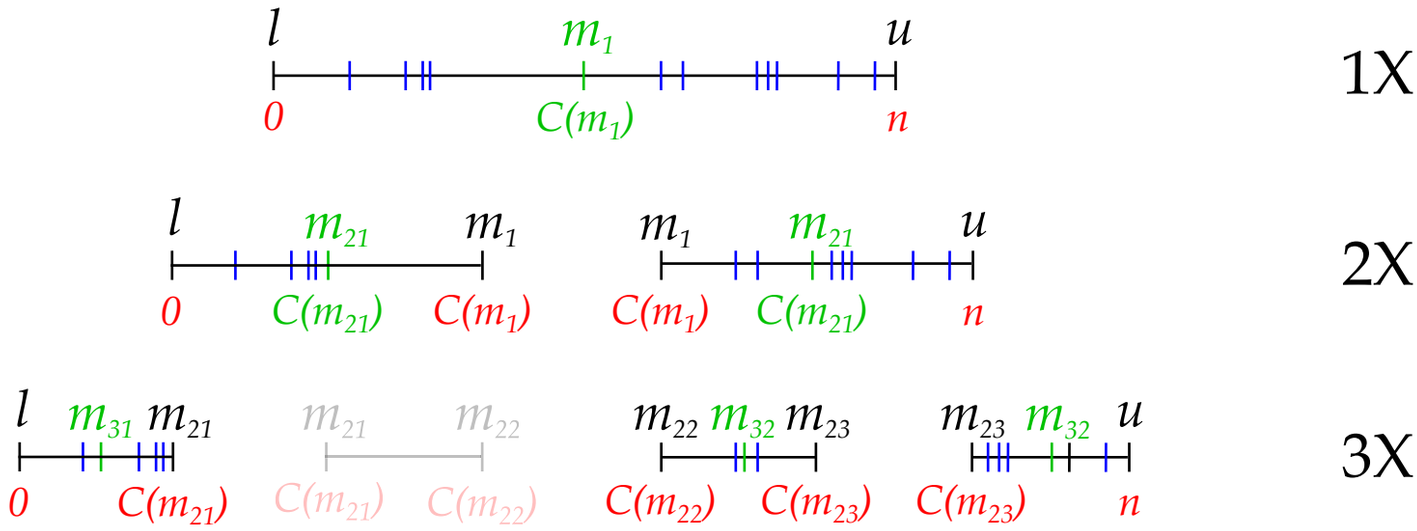


2X

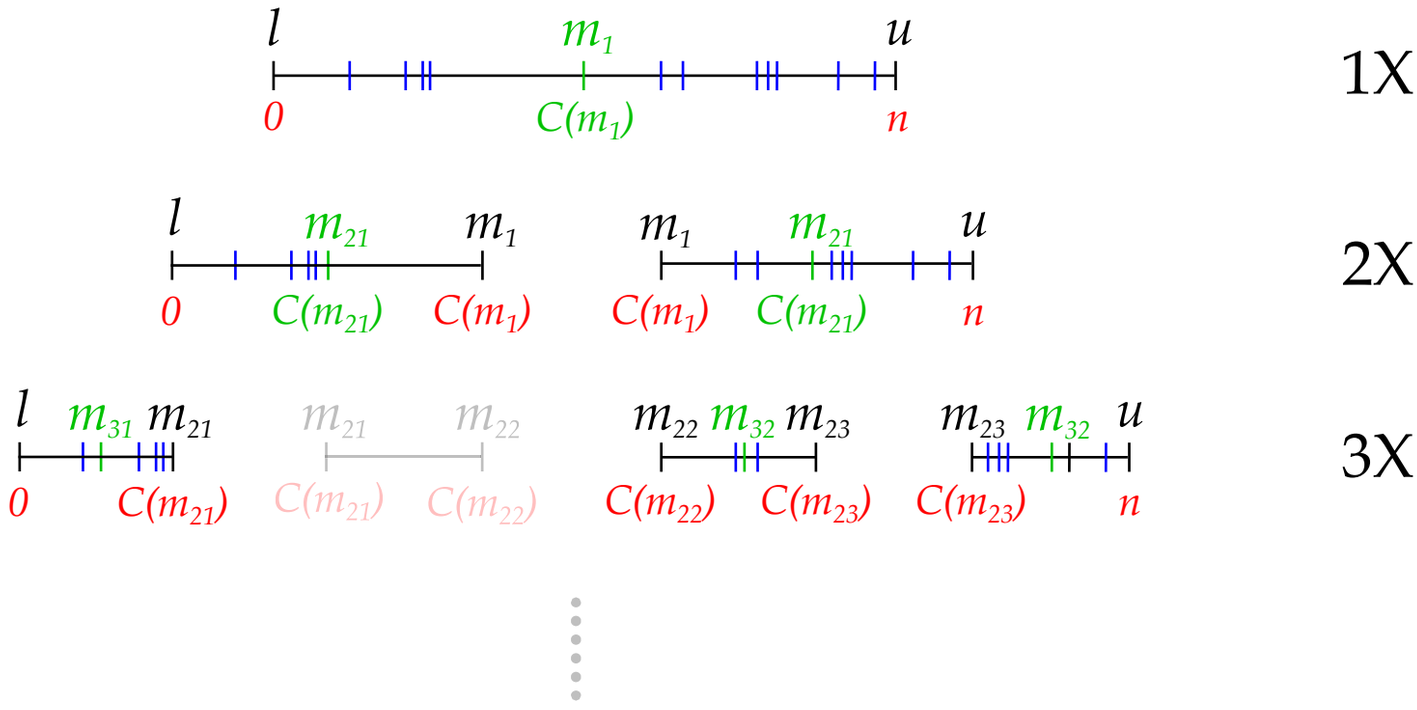
Bisection for eigenvalue spectrum



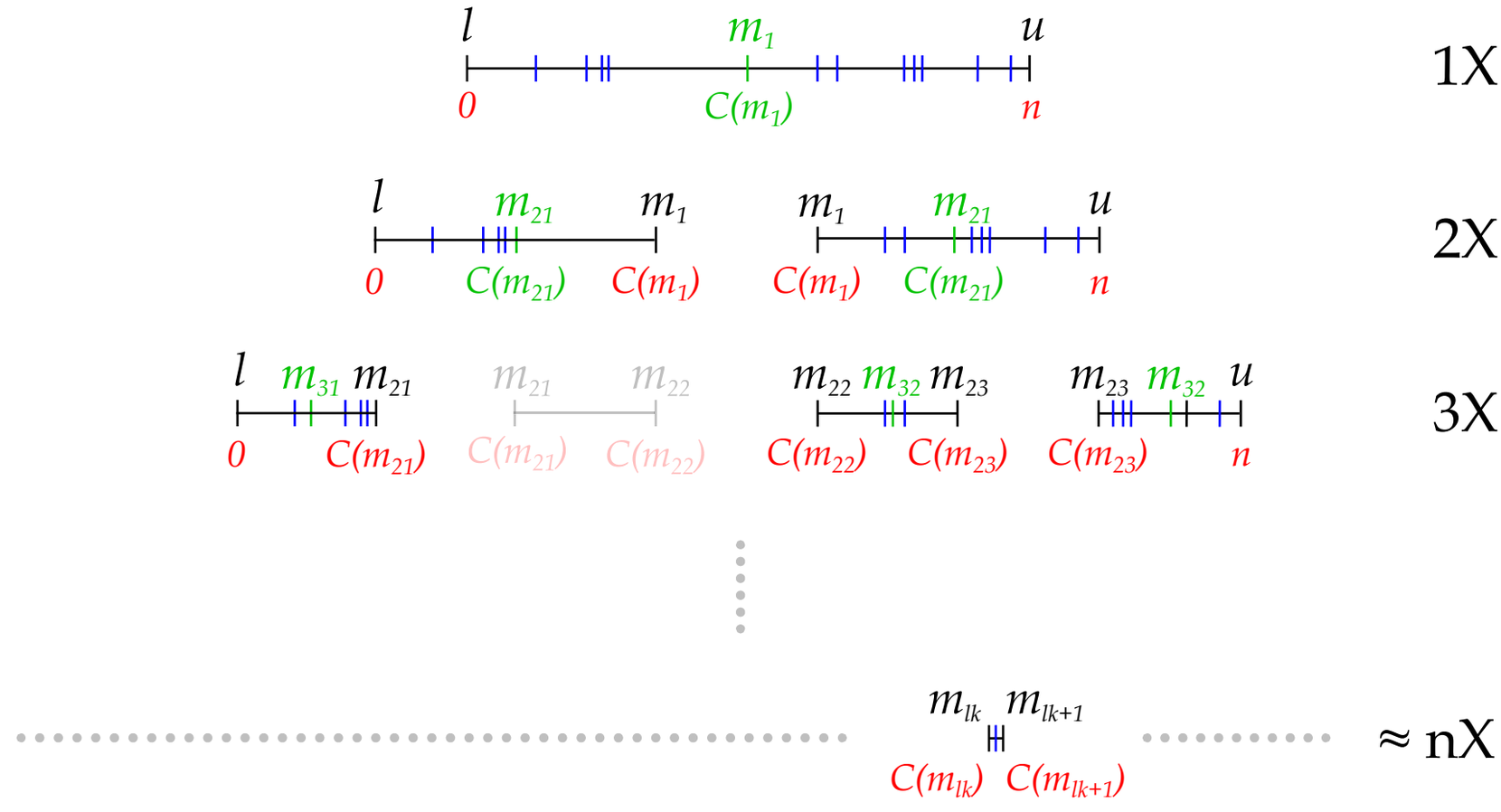
Bisection for eigenvalue spectrum



Bisection for eigenvalue spectrum



Bisection for eigenvalue spectrum



Bisection for eigenvalue spectrum

- **Strategy**: one thread per tree node (or interval).
  - Tree is processed level by level.

Bisection for eigenvalue spectrum

- **Strategy**: one thread per tree node (or interval).
  - Tree is processed level by level.
- **Implementation "details"**?
  - Memory management? Can and should we use shared memory?
  - How can we handle large matrices?

Bisection for eigenvalue spectrum

1. Compute Gerschgorin interval.

Bisection for eigenvalue spectrum

1. Compute Gerschgorin interval.
2. Move data to co-processor.
   - Matrix data (a_i, b_i).
   - Gerschgorin interval.

Bisection for eigenvalue spectrum

1. Compute Gerschgorin interval.
2. Move data to co-processor.
   - Matrix data (a_i, b_i).
   - Gerschgorin interval.
3. Launch kernel.

Bisection for eigenvalue spectrum

```
// move matrix data to shared mem

__shared__ float a[MAX_MAT_SIZE];
__shared__ float b[MAX_MAT_SIZE];

for( i=0; i<(mat_size/num_threads); ++i) {
  a[tid] = a_g[tid];
  b[tid] = b_g[tid];
}

__synchthreads();
```

Bisection for eigenvalue spectrum

```
struct SharedMem {

   float left[MAX_THREADS_BLOCK];
   float right[MAX_THREADS_BLOCK];

   short left_count[MAX_THREADS_BLOCK];
   short right_count[MAX_THREADS_BLOCK];

  short compact[MAX_THREADS_BLOCK+1];
};
```

Bisection for eigenvalue spectrum

```
// move Gerschgorin data to shared mem

__shared__ SharedMem smem;

if( 0 == tid) {
   smem.left[0] = lg;
   smem.right[0] = ug;
   smem.left_count[0] = 0;
   smem.right_count[0] = mat_size+1;
}
```

Bisection for eigenvalue spectrum

```
    // compute midpoint of interval

    // compute Sturm count for midpoint

    // write intervals to shared memory
    smem.left[tid] = left;
    smem.right[tid] = mid;
    smem.left[tid+1] = mid;
    smem.right[tid+1] = right;
    ...
}
```

Bisection for eigenvalue spectrum

```
if( tid < num_active threads) {

   float left = smem.left[tid];
   float right = smem.right[tid];
   ...

   // compute midpoint of interval
   float mid = midpoint( left, right);

   // compute Sturm count for midpoint
   mid_count = sturm_count( left, right, mid);
```

Bisection for eigenvalue spectrum

```
    // compute Sturm count for midpoint
    mid_count = sturm_count( left, right, mid);

    // write intervals to shared memory
    __syncthreads();
    smem.left[2*tid] = left;
    smem.right[2*tid] = mid;
    smem.left[2*tid+1] = mid;
    smem.right[2*tid+1] = right;

    num_intervals *= 2;
}
```

Interlude: prefix sum (scan)

- Swiss army knife of data-parallel programming.
- Sort, max, min, reduce, ...

Interlude: prefix sum (scan)

- Swiss army knife of data-parallel programming.
- Sort, max, min, reduce, ...
- Divide-and-Conquer strategy.
  - Traverse tree first bottom up and then top down.
  - Bottom up: gather information.
  - Top down: distribute information.

Bisection for eigenvalue spectrum

```
    // write intervals to shared memory
    __syncthreads();
    smem.left[2*tid] = left;
    smem.right[2*tid] = mid;
    smem.left[2*tid+1] = mid;
    smem.right[2*tid+1] = right;

    // compact intervals in shared memory
    ...

    num_intervals = smem.compact[num_intervals];
}
```

Bisection for eigenvalue spectrum

1. Read data from global to shared memory.
2. Until all intervals are converged:
   - Read interval into registers.
   - Bisect and compute Sturm count.
   - Write to shared memory.
   - Compact intervals.
3. Write eigenvalues to global memory.

Bisection for eigenvalue spectrum

```
for( j=0; j<(mat_size/blockDim.x); ++j){

  // read data with all threads in the block
  // make sure the shared memory is not used
  __syncthreads();

  // read data for current block
  if( addr < mat_size) {
    d_data[tid] = g_d[addr];
    lld_data[tid] = g_lld[addr];
  }
  __syncthreads();
```

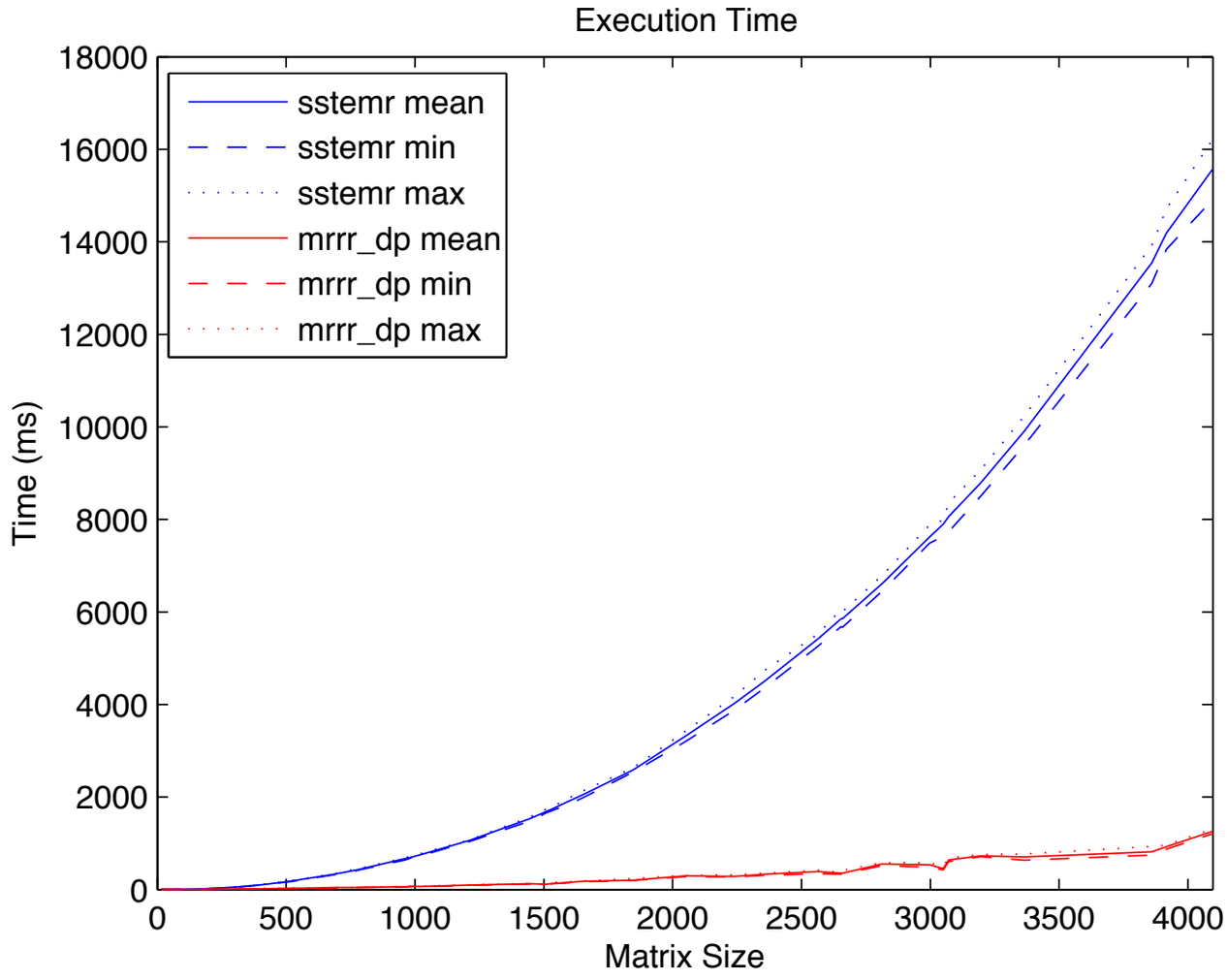
Bisection for eigenvalue spectrum

```
// read data for current block
if( addr < mat_size) {
  a[tid] = a_g[addr];
  b[tid] = b_g[addr];
}
__syncthreads();

// for all active intervals
if(tid < num_intervals) {

  // Sturm count computation for current block
  ...
```

Bisection for eigenvalue spectrum



How to get started?

- http://developer.nvidia.com/cuda/
  - Cuda 3.0 has device emulation mode.
  - SDK has many, many code examples.
  - NVIDIA linear algebra libraries.
- http://www.cs.berkeley.edu/~volkov/
  - Performance analysis of linear algebra on data parallel co-processors.

Conclusion

- Linear algebra on data parallel co-processors can be significantly more efficient than on CPUs.
- **But** requires architecture specific design and optimizations.
- For many algorithms still unclear how to efficiently implement them on a data parallel co-processor.