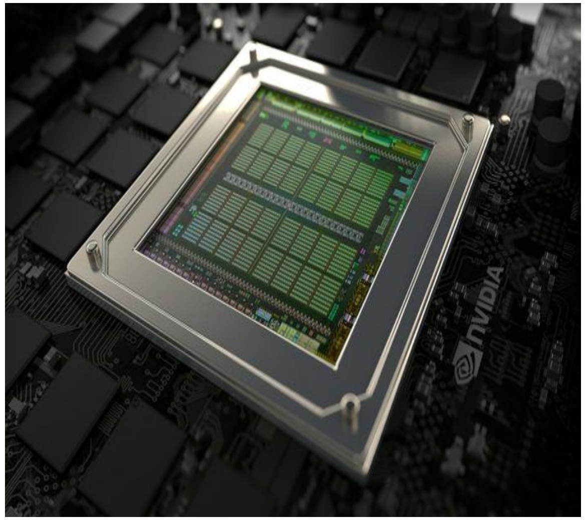
الجامعة اللبنانية
UNIVERSITE LIBANAISE

Computer Architecture II

Done By:
Mahdi Mahdi
Ali Tawbi

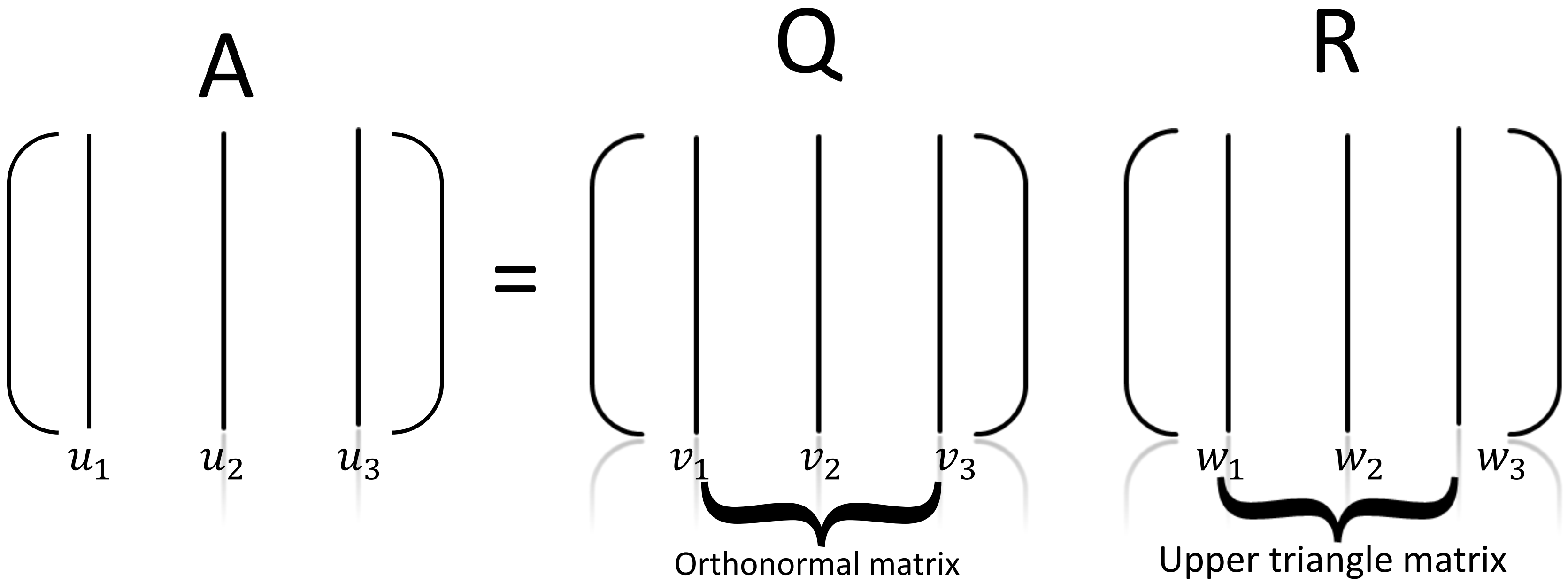
Instructors:
Dr. Hala Hijazi
Dr. Majed Sinan

CPU vs. GPU



Performance comparison for the Gram-Schmidt algorithm

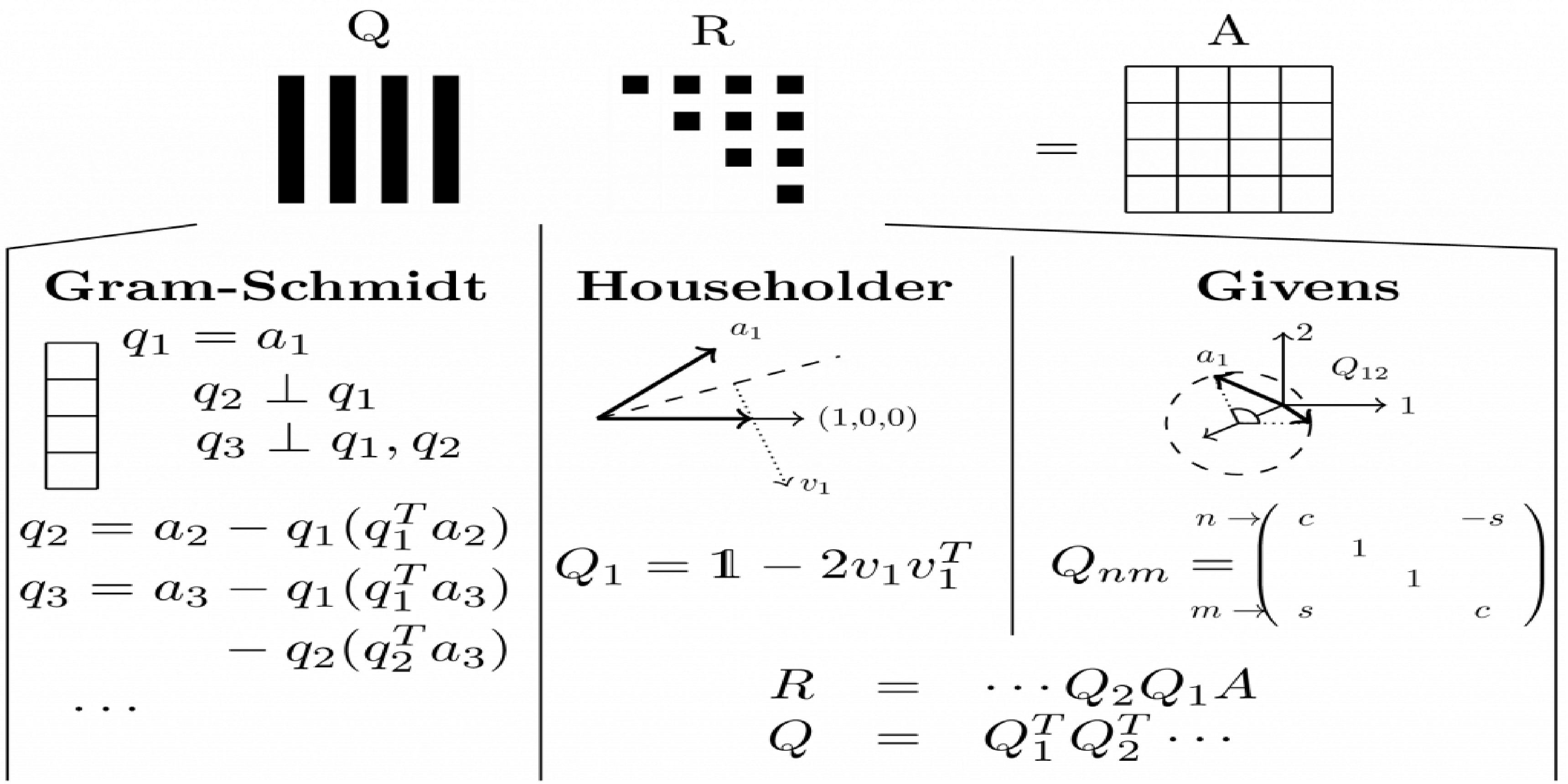
Introduction



Some uses of QR decompositions :

- solving linear systems of equations and linear least square problems.
- determine eigenvalues and vectors.

Methods for determine QR decomposition:



❖ Gram-Schmidt method.

- Operates on whole columns.
- Stability problems in correlated channels.

❖Householder reflections.
Column dash wise operation.
Higher stability in correlated channels

❖Givens rotations.
Works with small 2x2 rotation matrix.
Suitable for small MIMO systems

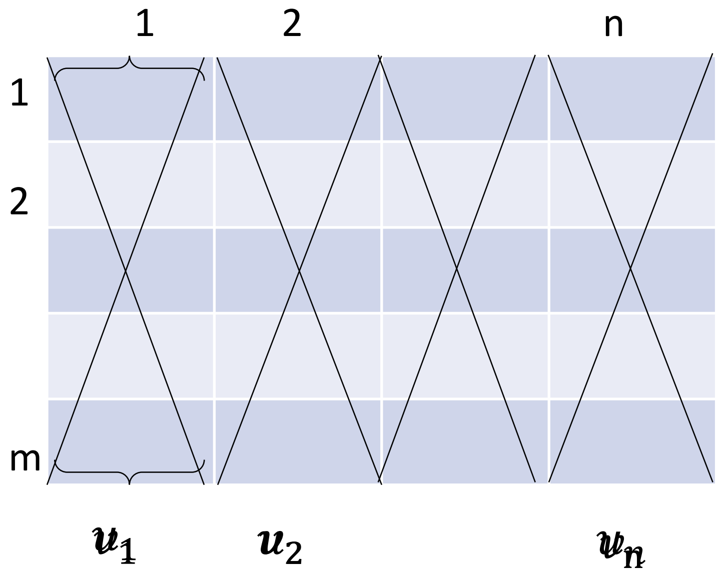
In this research we will investigate how the Gram-Schmidt method can be parallelized specifically to take advantage of recent multi-core and

GPU architectures, so we will compare:

- various implementations using OpenMP on the CPU.
- a native GPU implementation using NVIDIA’s Compute Unified Device Architecture (CUDA).
- and versions using routines from the basic linear algebra subprograms library (BLAS) both on the CPU and GPU.

And we will show how blocking techniques improve cache usage and how useful they are to reduce the memory traffic in the CPU and GPU versions.

Gram-Schmidt method in a nutshell:



$$v_1 = \frac{u_1}{\|u_1\|} - \sum_{k=2}^{n} proj_{u_k}\left(\frac{u_1}{\|u_1\|}\right)$$

$$v_2 = \frac{u_2}{\|u_2\|} - \sum_{k=3}^{n} proj_{u_k}\left(\frac{u_2}{\|u_2\|}\right)$$

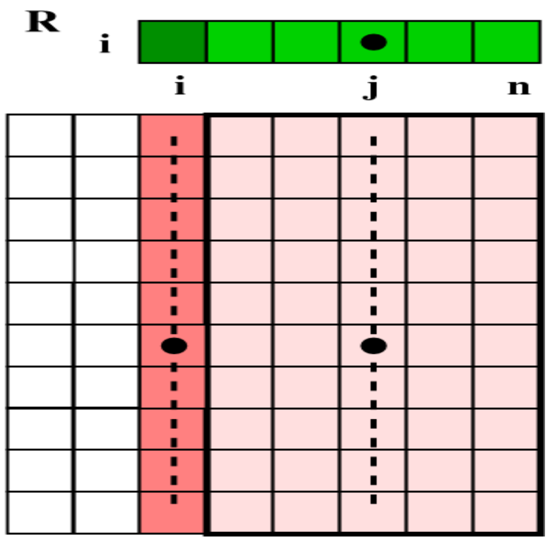
$$v_p = \frac{u_p}{\|u_p\|} - \sum_{k=p+1}^{n} proj_{u_k}\left(\frac{u_p}{\|u_p\|}\right) \quad \forall p, 1 \leq p \leq n$$

... they are both orthogonal to all the other columns.

	1	2		n
1	r_{11}	r_{12}		r_{1n}
2	0	r_{22}		
	0	0		
n	0	0	0	r_{nn}

R

Q



R is obtained as a by-product from the projection coefficients.

$$r_{ij} = \mathrm{col(i)} \cdot col(j)$$

Algorithm:

- Let a_{Kj} be the elements of A for k = 1,...,m and j = 1,...,n, and correspondingly q_{Kj} and r_{Kj} the elements of Q and R, respectively. Initially, we choose Q = A, i. e. q_{Kj} = a_{Kj} for all k, j. Then, the modified Gram-Schmidt method written in **pseudo code** looks as :

```
R = 0
for i = 1, . . . , n
    for j = i, . . . , n
        for k = 1, . . . , m
            r_ij = r_ij + q_ki q_kj
        end
    end
    r_ii = √r_ii
    for k = 1, . . . , m
        q_ki = q_ki / r_ii
    end
    for j = i + 1, . . . , n
        r_ij = r_ij / r_ii
    end
    for j = i + 1, . . . , n
        for k = 1, . . . , m
            q_kj = q_kj − q_ki r_ij
        end
    end
end
```

Algorithm in other notations :

Vector notations:

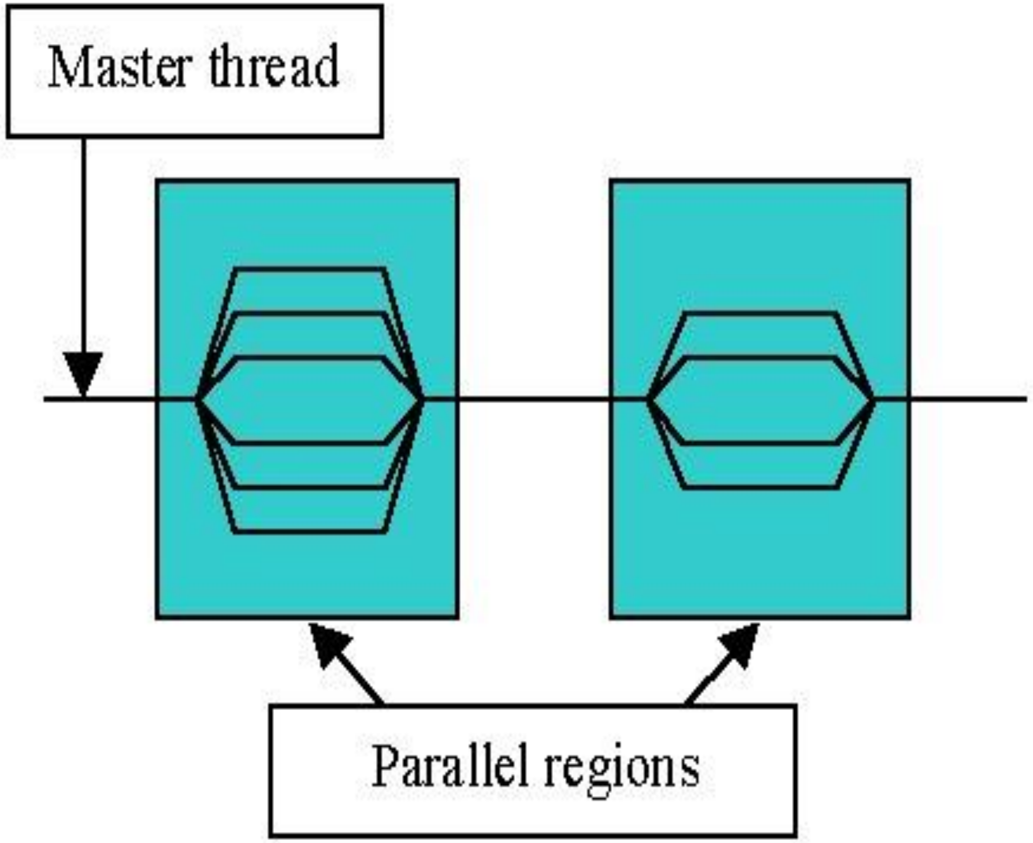
$$
\begin{array}{l}
\mathrm{R} = 0 \\
\textbf{for } i = 1, \dots, n \\
\quad ! \ \textit{dot products, matrix-vector mult} \\
\quad ! \ R_{i,i:n} = Q^{\mathrm{T}}_{1:m,i:n} \times Q_{1:m,i} \\
\quad \textbf{for } j = i, \dots, n \\
\qquad r_{ij} = Q_{1:m,i} \cdot Q_{1:m,j} \\
\quad \textbf{end} \\
\quad ! \ \textit{normalize column } i \textit{ of } Q \\
\quad s = \sqrt{r_{ii}} \ // \text{ norm of i-th column of Q} \\
\quad Q_{1:m,i} = Q_{1:m,i}/s \\
\\
\quad ! \ \textit{compute projection factors} \\
\quad R_{i,i:n} = R_{i,i:n}/s \\
\\
\quad ! \ \textit{orthogonalization, rank-1 update} \\
\quad ! \ Q_{1:m,i+1:n} -= Q_{1:m,i} \times R^{\mathrm{T}}_{i,i+1:n} \\
\quad \textbf{for } j = i+1, \dots, n \\
\qquad Q_{1:m,j} = Q_{1:m,j} - r_{ij} Q_{1:m,i} \\
\quad \textbf{end} \\
\textbf{end}
\end{array}
$$

using BLAS:

```
subroutine QR(q, r, m, n)
   integer i, m, n
   real q(m,n), r(m,n), S
   do i = 1, n
      ! R_{i,i:n} = 1.0 * Q^T_{1:m,i:n} × Q_{1:m,i} + 0.0 * R_{i,i:n}
      call sgemv(trans = 'T', m = m, n = n-i+1, alpha = 1.0, A = q(1,i),
                 lda = m, x = q(1,i), incx = 1, beta = 0.0, y = r(i,i), incy = n)
      S = 1.0 / sqrt(r(i,i))
      call sscal(n = m, alpha = S, x = q(1,i), incx = 1)
      call sscal(n = n-k+1, alpha = S, x = r(i,i), incx = n)
      ! Q_{1:m,i+1:n} = -1.0 * Q_{1:m,i} * R^T_{i,i+1:n} + Q_{1:m,i+1:n}
      call sger(m = m, n = n-i, alpha = -1.0, x = q(1,i), incx = 1,
                y = r(i,i+1), incy = n, A = q(1,i+1), lda = m)
   end do
end subroutine QR
```

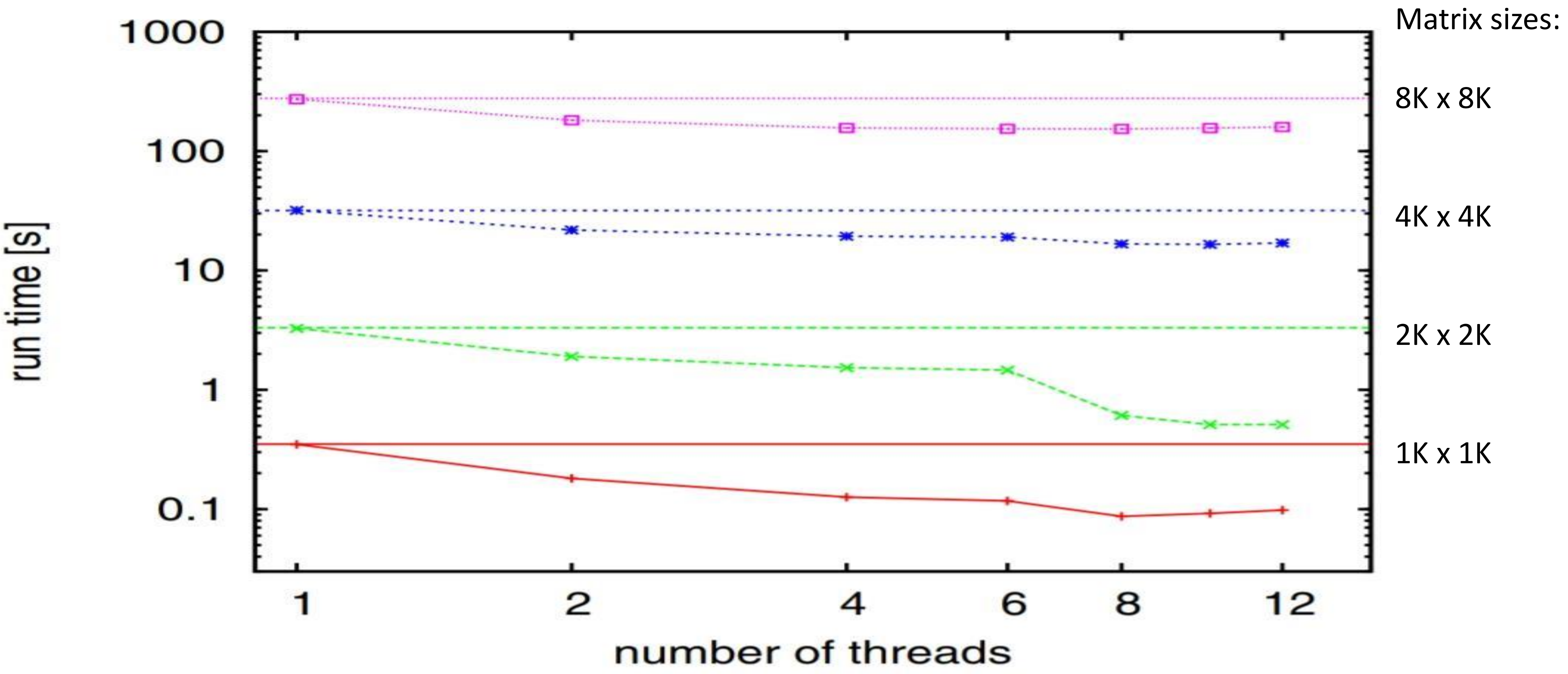
CPU parallelization and blocking:

OpenMP parallelization:



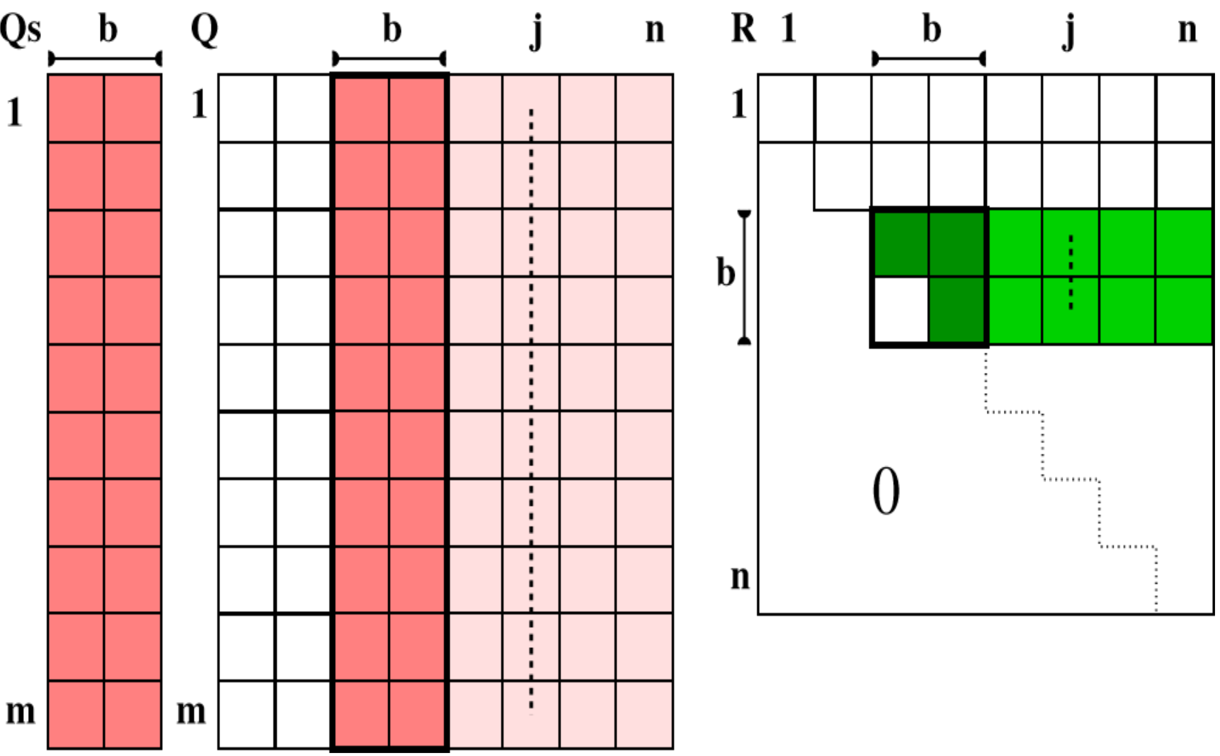
The outermost loop

$$
\begin{array}{l}
\mathrm{R} = 0 \\
\textbf{for } i = 1, \ldots, n \\
\quad \textbf{for } j = i, \ldots, n \\
\quad\quad \textbf{for } k = 1, \ldots, m \\
\quad\quad\quad r_{ij} = r_{ij} + q_{ki} q_{kj} \\
\quad\quad \textbf{end} \\
\quad \textbf{end} \\
\quad r_{ii} = \sqrt{r_{ii}} \\
\quad \textbf{for } k = 1, \ldots, m \\
\quad\quad q_{ki} = q_{ki} / r_{ii} \\
\quad \textbf{end} \\
\quad \textbf{for } j = i+1, \ldots, n \\
\quad\quad r_{ij} = r_{ij} / r_{ii} \\
\quad \textbf{end} \\
\quad \textbf{for } j = i+1, \ldots, n \\
\quad\quad \textbf{for } k = 1, \ldots, m \\
\quad\quad\quad q_{kj} = q_{kj} - q_{ki} r_{ij} \\
\quad\quad \textbf{end} \\
\quad \textbf{end} \\
\textbf{end}
\end{array}
$$



We measured the performance of the OpenMP implementation on one CPU node with two Intel Xeon X5650 CPUs (Westmere architecture, 2.67 GHz, 6 cores, 12 MB L3 cache)

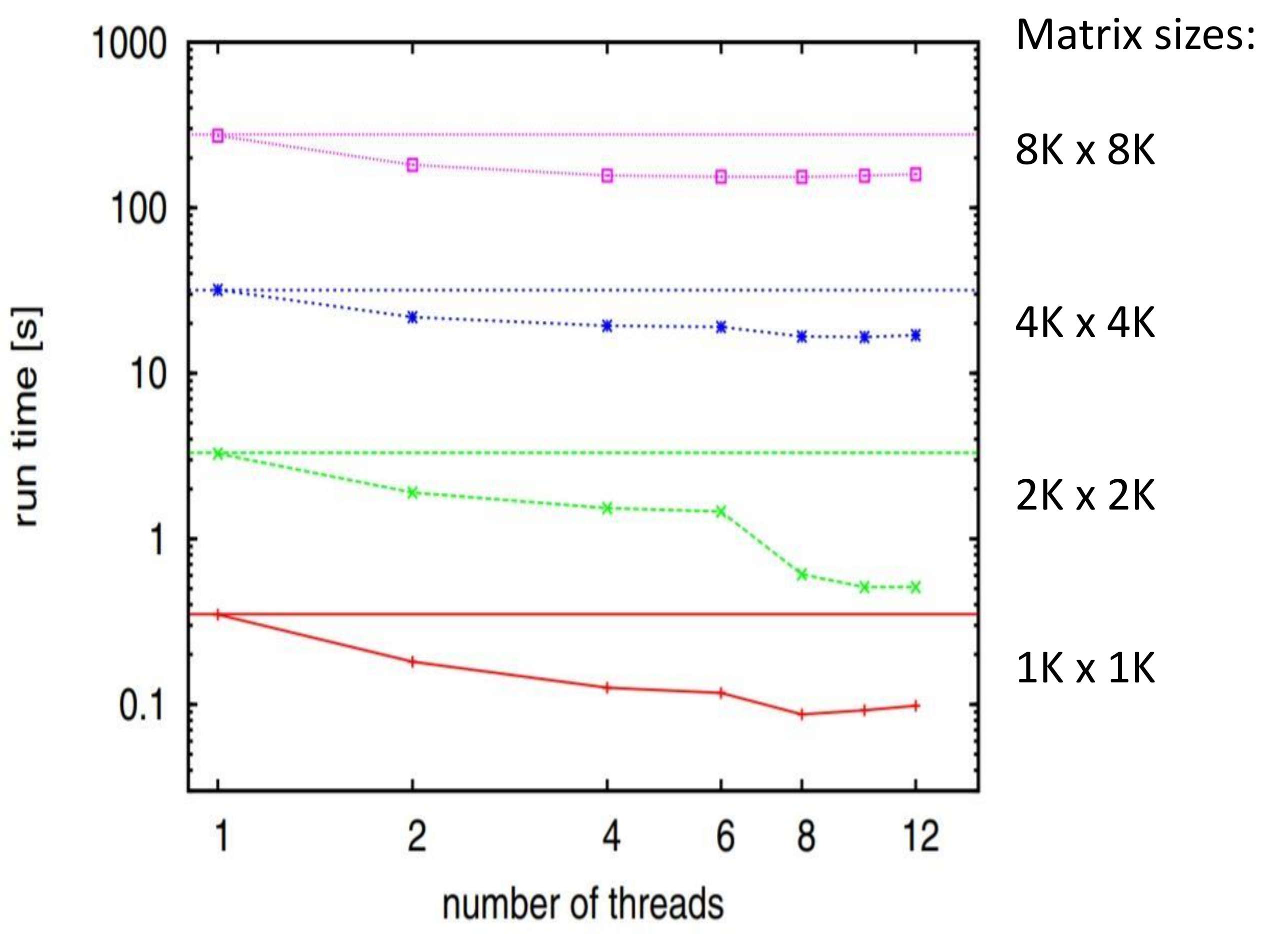
In order to reduce the required memory bandwidth we have implemented a blocked version of the code:

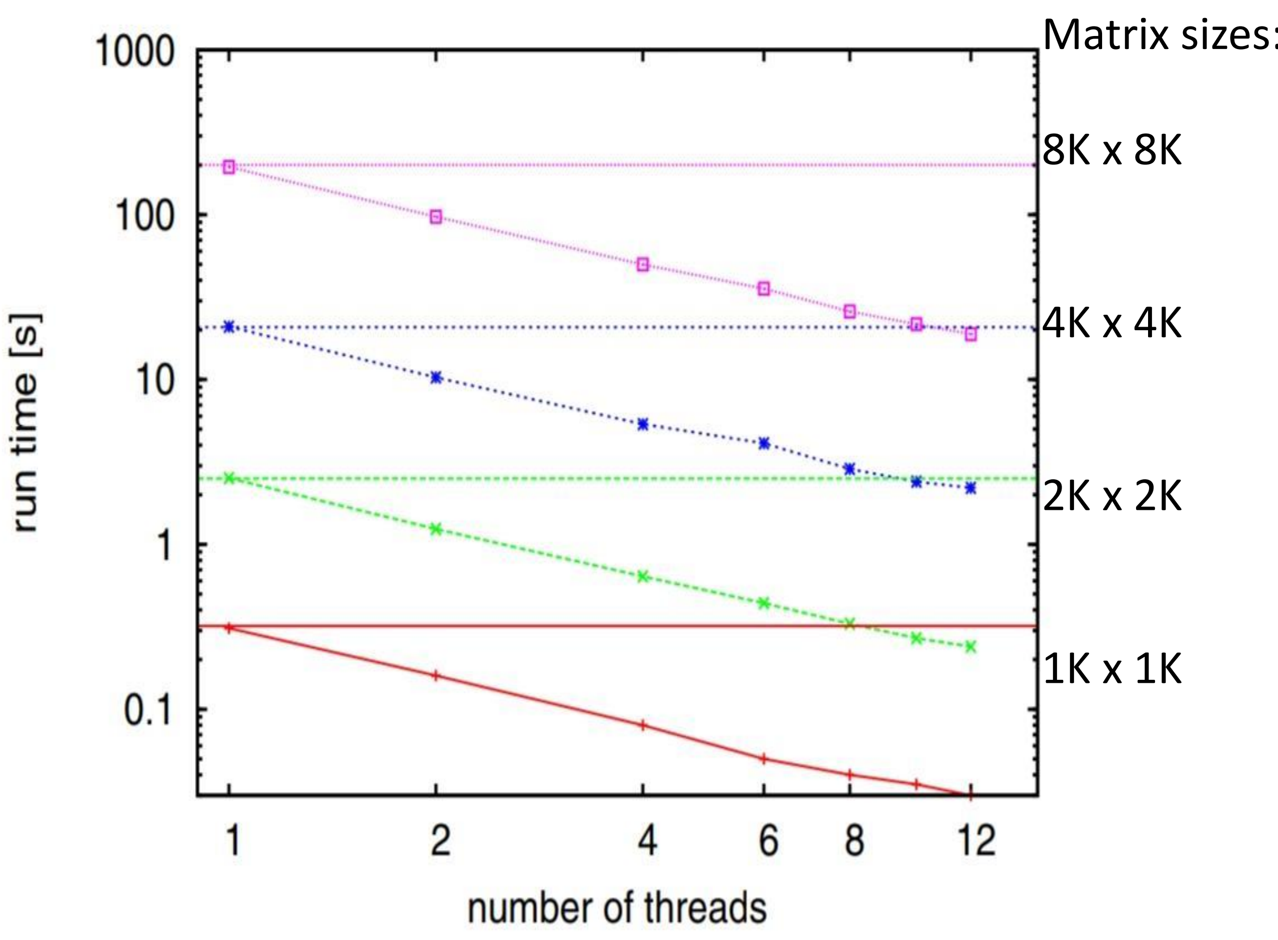


The blocked code, in which only the order of computations is changed, is as follows:

for $i1 = 1, b+1, 2*b+1, \ldots, n$
    $i2 = \min(i1 + b - 1, n)$
    $Q^s(1:m, 1:b) = Q(1:m, i1:i2)$
    call QR(Q(1:m,i1:i2), R(i1:i2,i1:i2))
    for $j = i1 + b, \ldots, n$ // parallel loop over remaining columns
        for $i = i1, \ldots, i2$
            $r_{ij} = Q^s_{1:m, i-i1+1} \cdot \mathbf{Q_{1:m,j}}$
            $r_{ij} = r_{ij} \,/\, r_{ii}$
            $\mathbf{Q_{1:m,j}} \mathrel{-=} r_{ij} * Q_{1:m,i}$
        end
    end
end

But How much should the size of b??

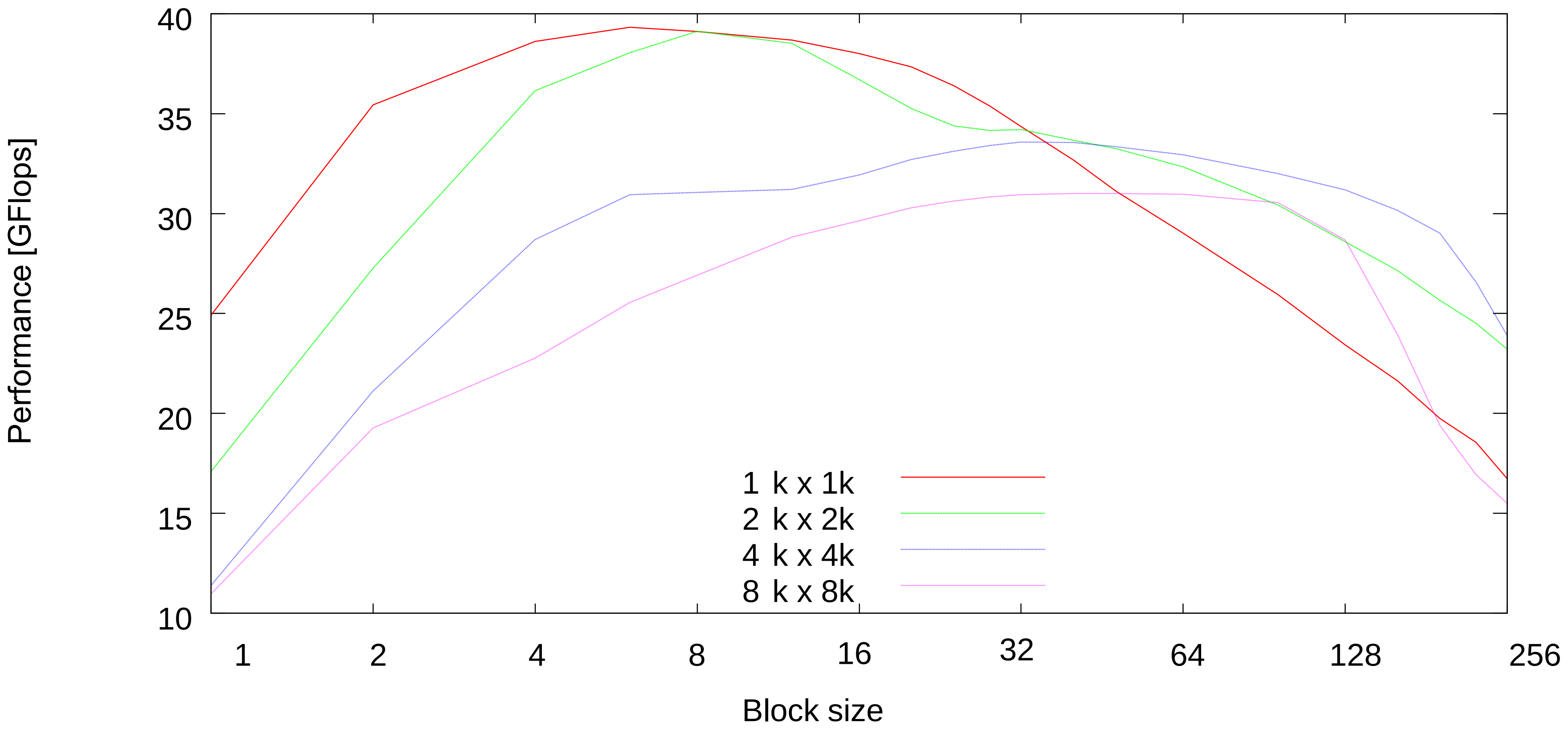




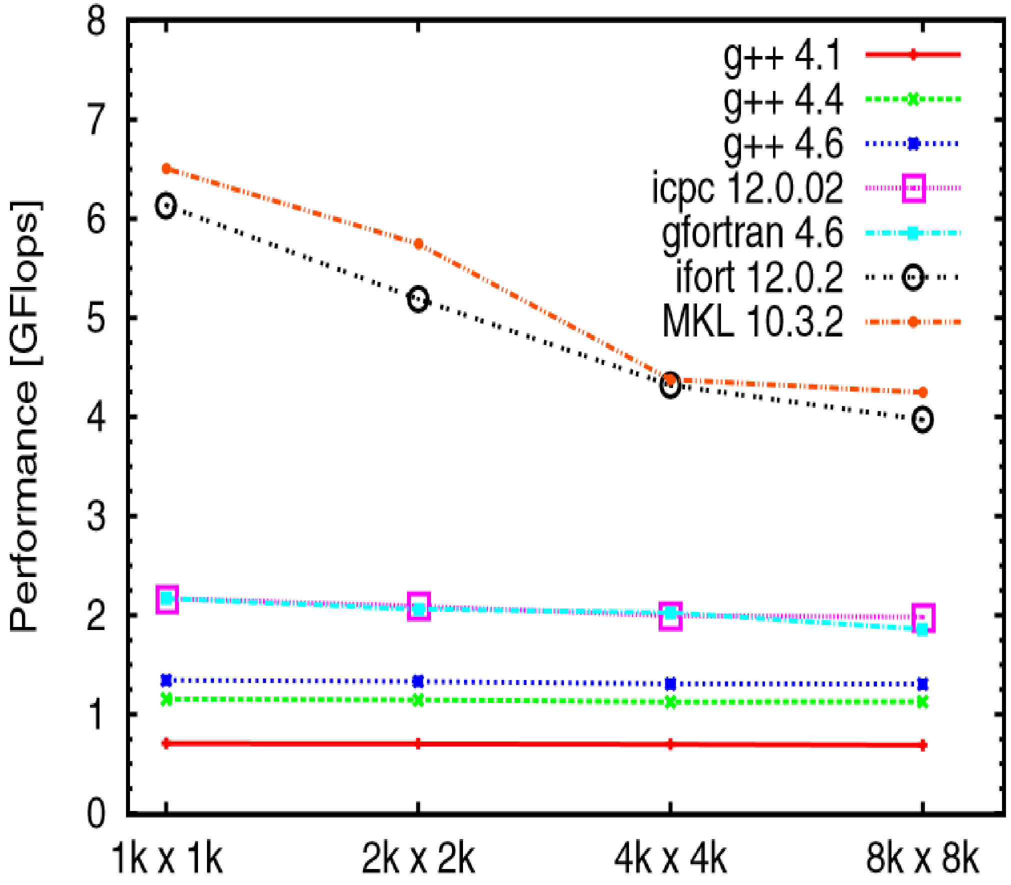
Without blocking.

With blocking : this graph shows a nearly ideal scaling with the number of threads.

The performance respecting to the size of b:



Language and Compiler dependence:



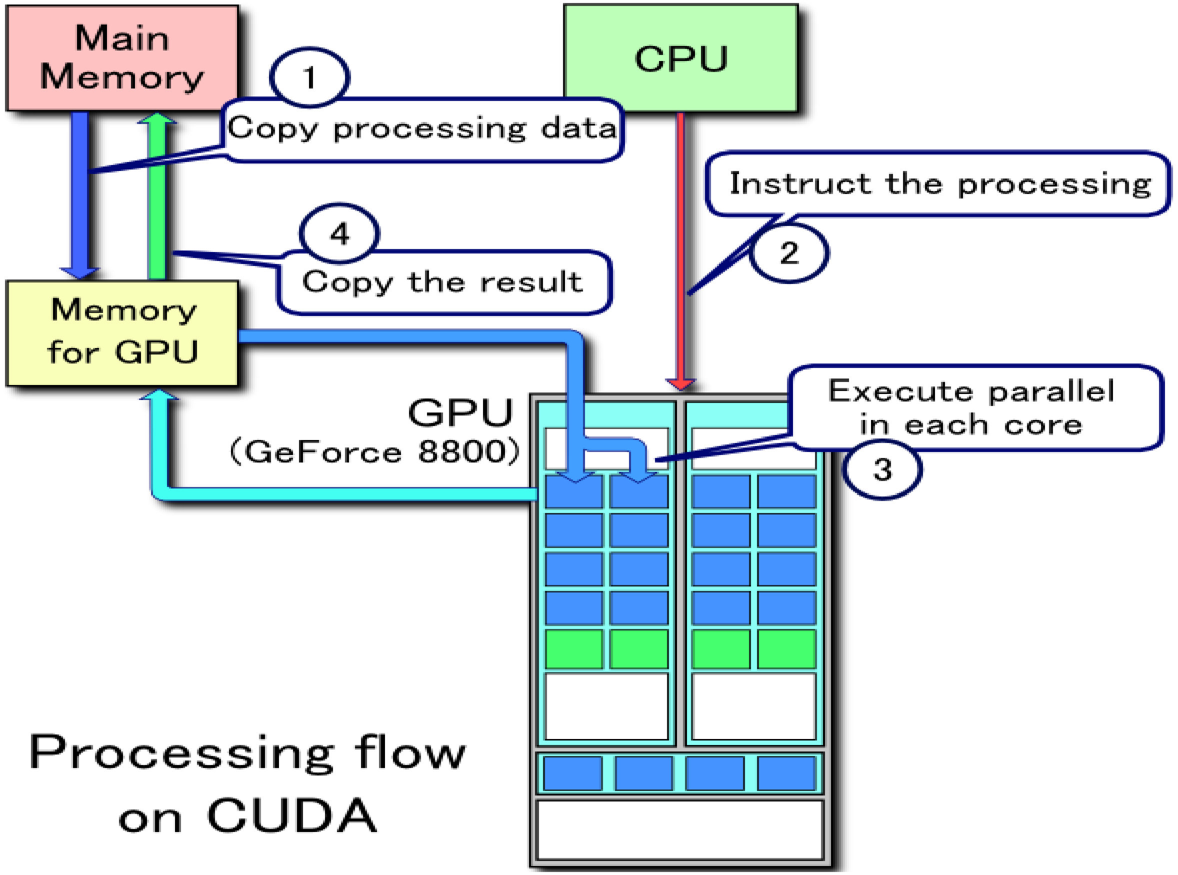
Performance in Gigaflops of a serial CPU version compiled using various compilers on a Intel Xeon X5650 @ 2.67 GHz CPU, depending on the matrix size.

compiler	g++	g++	icpc	gfortran	ifort	MKL
version	4.1	4.6	12.0.02	4.6	12.0.2	
flags	-O5	-O5	-fast	-O5	-fast	
unblocked, serial	1057.41	878.15	505.58	538.47	276.81	258.78
unblocked, 6 threads	192.81	175.06	156.05	162.00	153.84	
unblocked, 12 threads	158.62	190.33	157.54	200.02	159.01	
blocked, serial	1025.72	841.56	507.16	498.20	195.33	
blocked, 6 threads	178.84	145.15	88.05	110.11	35.55	
blocked, 12 threads	95.33	76.28	45.35	56.58	18.86	

Execution times (in seconds) of the unblocked and blocked version for problem size 8k × 8k. Results are given for the serial and for the parallel version running on one (6 threads) or two CPUs (12 threads).

Native GPU implementation:

- NVIDIA CUDA framework processing:



Orthogonalization on GPU:
Algorithm:

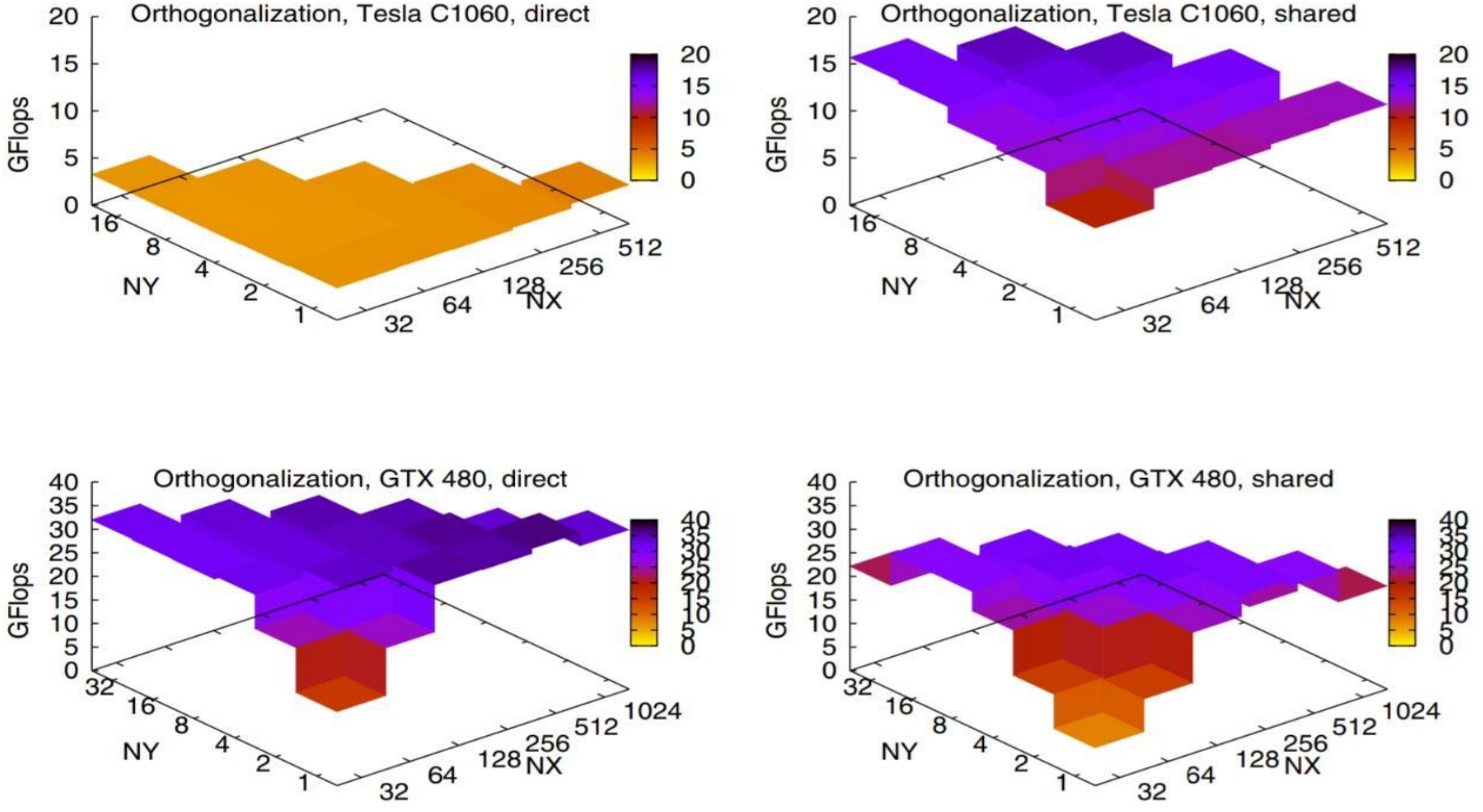
```
__global__
void Orthogonalization(Q, R, m, n, i)
{

    tx = threadIx.x;
    ty = threadIx.y;
    j  = blockId * NY + ty + i + 1;

    for (k=tx+1; k <= m; k+=NX) {


        q_kj = q_kj - r_ij q_ki;
    }
}
```

Using global memory

```
...
...
{
    __shared__ ri[NY], qi[NX];
    ...
    ...
    ...
    if (tx == 0) ri[ty] = r_ij;
    for (k=tx+1; k <= m; k+=NX) {
        if (ty==0) qi[tx] = q_ki;
        __syncthreads();
        q_kj = q_kj - ri[ty] * qi[tx];
    }
}
```

Using shared memory

Comparison of the performance between Tesla C1060 and GTX 480 using direct and shared memory:



Dot product on GPU:

Algorithm:

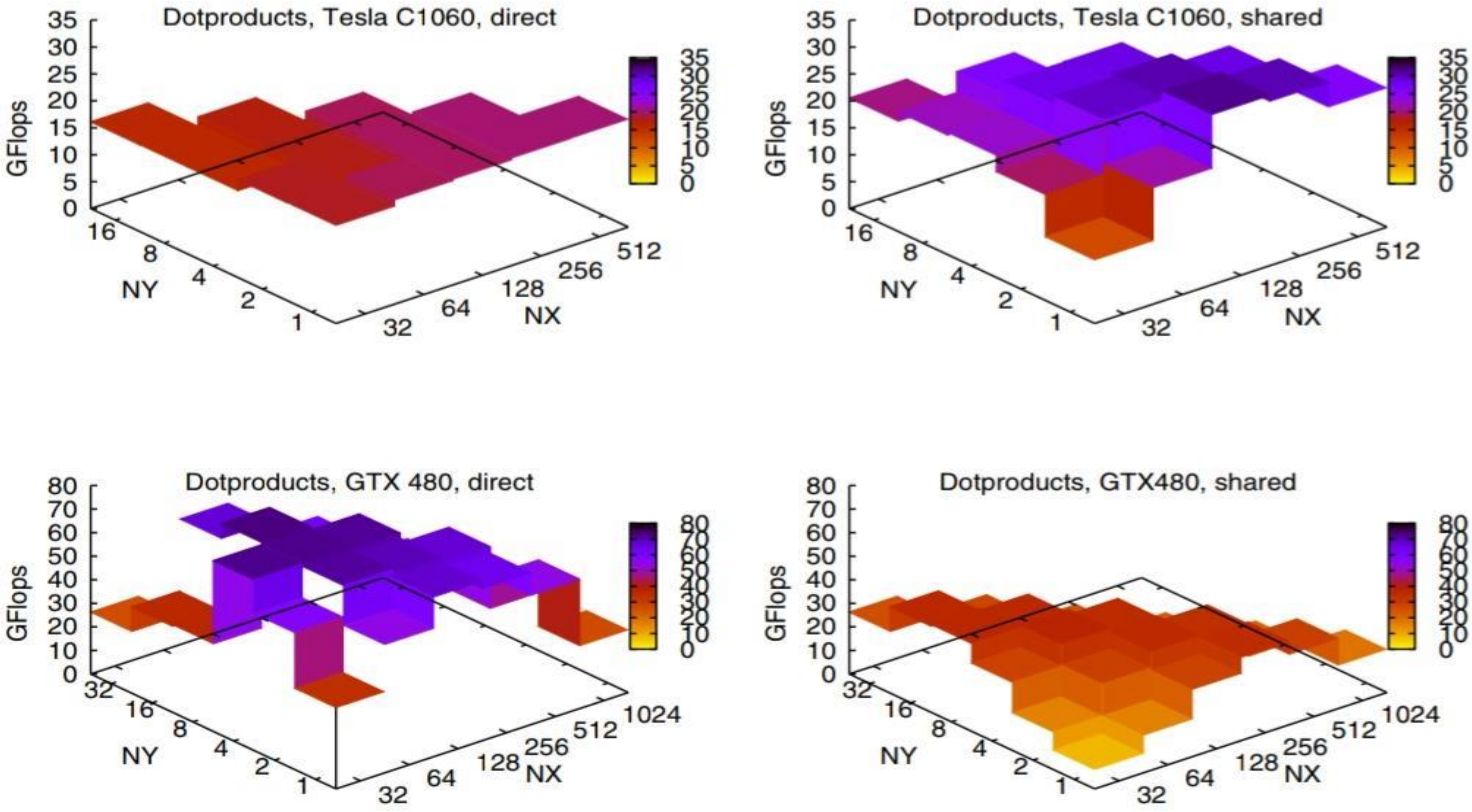
Using global memory

```
__global__
void Dotproducts(Q, R, m, n, i)
{
    __shared__   RS[NY][NX];

    tx = threadIx.x;
    ty = threadIx.x;
    j  = blockId * NY + ty + i;
    sum = 0;
    for (k=tx+1; k<=m; k+=NX) {


        sum += q_ki q_kj;
    }
    // reduction: r_ij += sum;
    RS[ty][tx] = sum;
    NT = NX;
    while (NT > 1) {
        __syncthreads();
        NT = NT / 2;
        if (tx < NT)
            RS[ty][tx] += RS[ty][tx+NT];

    }
    if (tx==0) r_i,j = RS[ty][0];
}
```

Using shared memory

```
...
...
{
    ...
    __shared__  qi[NX];
    ...
    ...
    ...
    ...
    for (...) {
        if (ty==0) qi[tx] = q_ki;
        __syncthreads();
        sum += qi[tx] * q_kj
    }
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
    ...
}
```

Comparison of the performance between Tesla C1060 and GTX 480 using direct and shared memory:



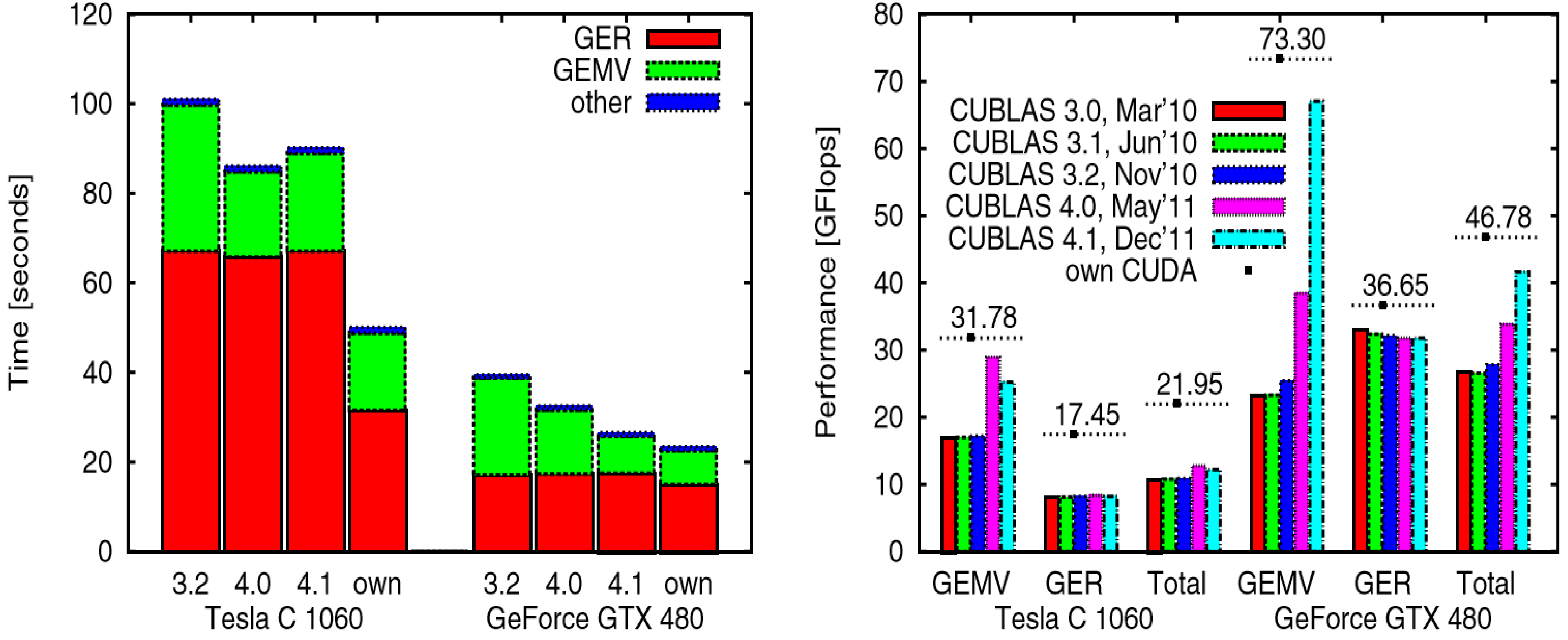
What is CUBLAS Library?

- BLAS
  - Basic Linear Algebra Subprogram
  - A library to perform basic linear algebra
  - Divided into three levels
  - Such as MKL BLAS,CUBLAS, C++ AMP BLAS……

- CUBLAS
  - An high level implementation of BLAS on top of the NVIDIA CUDA runtime
  - Single GPU or Multiple GPUs
  - Support CUDA Stream

Why we need CUBLAS?

- CUBLAS
  - Full support for all 152 standard BLAS routines
  - Support single-precision, double-precision, complex and double complex number data types
  - Support for CUDA steams
  - Fortran bindings
  - Support for multiple GPUs and concurrent kernels
  - Very efficient



Comparison of our CUDA version with the CUBLAS version using different CUBLAS versions.
Left graph shows cumulative execution time, right graph the performance.

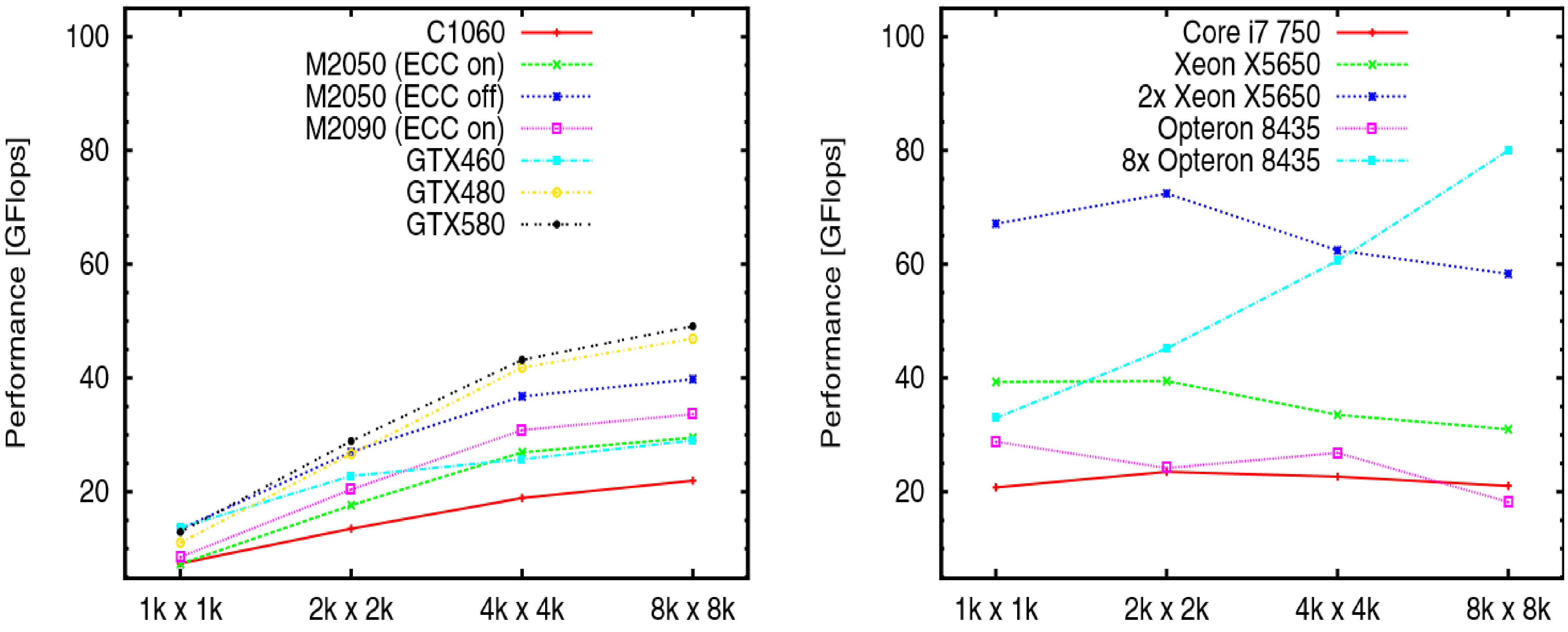
Comparison of CPU and GPU :

Name	Compute		Cores	CUDA	Proc	Mem	Mem	Global
	Capability	M P	/ MP	Cores	Clock	Clock	Width	Mem
					MHz	MHz	bit	GB
C1060	1.3	30	8	240	600	1600	512	4
M2090	2.0	16	32	512	650	1850	384	6
M2050	2.0	14	32	448	573	1546	384	6
GTX460	2.1	7	48	336	675	1800	256	1
GTX480	2.0	15	32	480	700	1848	384	1.5
GTX580	2.0	16	32	512	772	2004	384	3

GPU

Name	CPU	Cores	L2-Cache	L3-Cache
	MHz		kB	MB
Intel Xeon X5650	2666	6	6 x 256	12
Intel i5 750	2660	4	4 x 256	8
AMD Opteron 8435	2660	6	6 x 512	6

CPU



Performance of Gram-Schmidt implementation using different processors (left: GPUs, right: CPUs)