



# Computer Architecture II

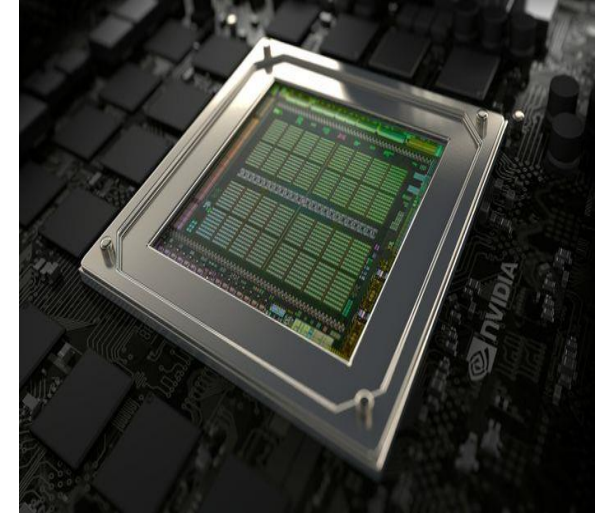


Done By:  
Mahdi Mahdi  
Ali Tawbi

Instructors:  
Dr. Hala Hijazi  
Dr. Majed Sinan



# CPU vs. GPU



Performance comparison for the Gram-Schmidt algorithm

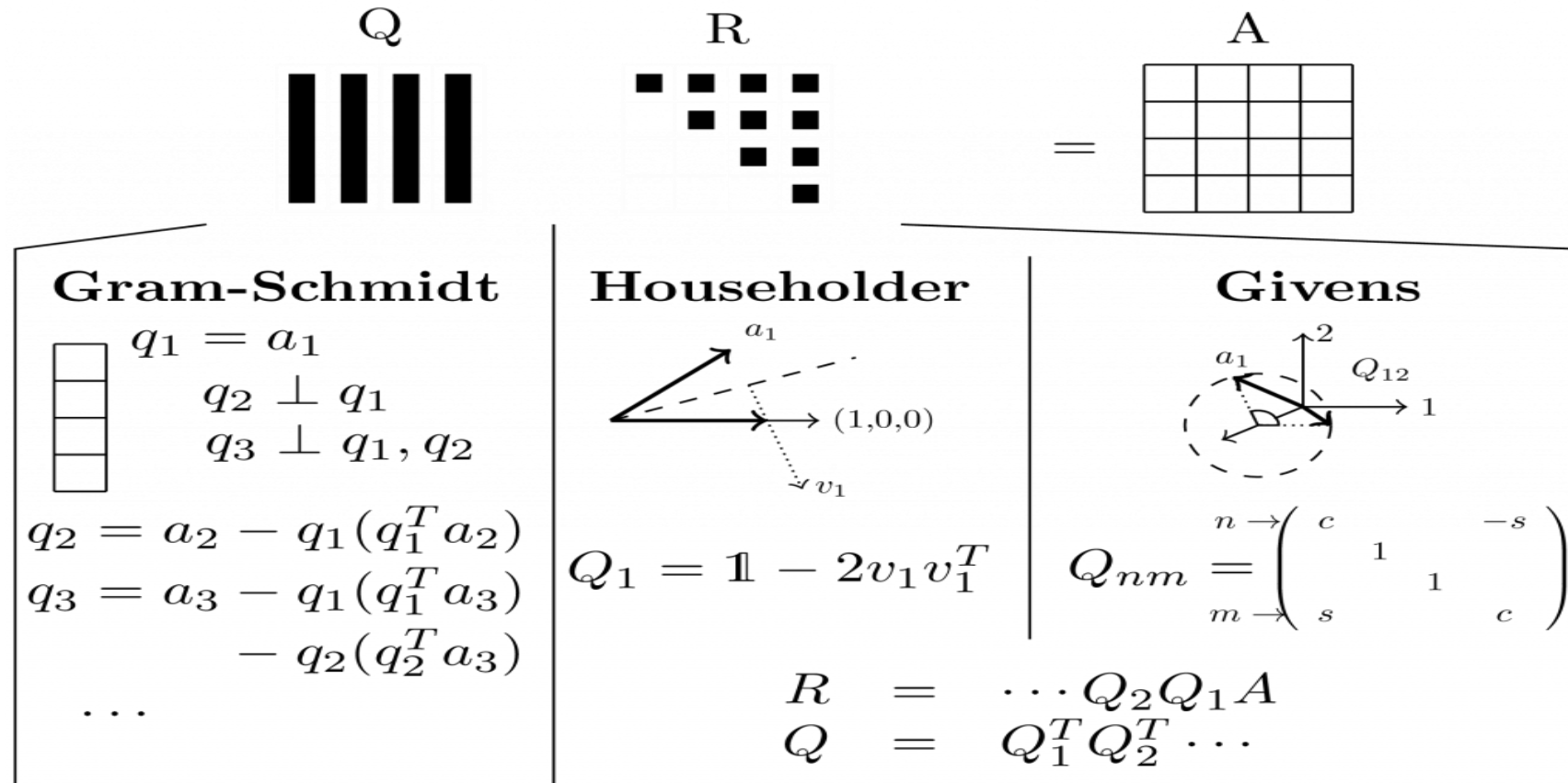
# Introduction

$$\begin{array}{c} \text{A} \\ \left( \begin{array}{c|c|c} & & \\ \hline & & \\ & & \end{array} \right) \\ \begin{array}{ccc} u_1 & u_2 & u_3 \end{array} \end{array} = \begin{array}{c} \text{Q} \\ \left( \begin{array}{c|c|c} & & \\ \hline & & \\ & & \end{array} \right) \\ \begin{array}{ccc} v_1 & v_2 & v_3 \end{array} \\ \underbrace{\hspace{10em}}_{\text{Orthonormal matrix}} \end{array} \begin{array}{c} \text{R} \\ \left( \begin{array}{c|c|c} & & \\ \hline & & \\ & & \end{array} \right) \\ \begin{array}{ccc} w_1 & w_2 & w_3 \end{array} \\ \underbrace{\hspace{10em}}_{\text{Upper triangle matrix}} \end{array}$$

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.

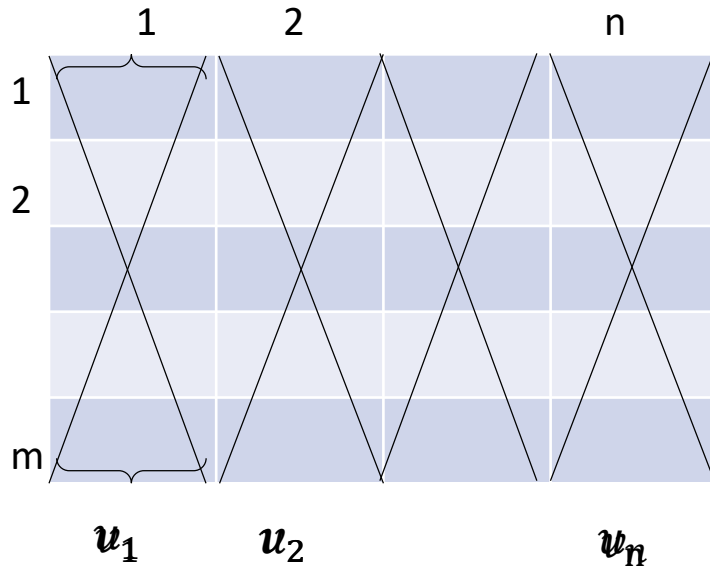
Suitable for small MIMO systems

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:

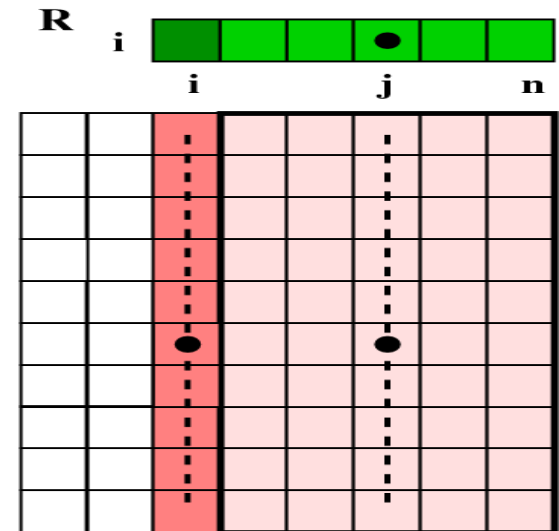


$$v_p = \frac{u_p}{\|u_p\|} - \sum_{k=1}^{p-1} \text{proj}_{u_k} \left( \frac{u_p}{\|u_p\|} \right) \quad \forall p, 1 \leq p \leq n$$

This step is repeated for all columns. The first column is normalized to have unit length. The second column is orthogonalized against the first column. The third column is orthogonalized against the first two columns. The fourth column is orthogonalized against the first three columns. The fifth column is orthogonalized against the first four columns. The sixth column is orthogonalized against the first five columns. The seventh column is orthogonalized against the first six columns. The eighth column is orthogonalized against the first seven columns. The ninth column is orthogonalized against the first eight columns. The tenth column is orthogonalized against the first nine columns. The eleventh column is orthogonalized against the first ten columns. The twelfth column is orthogonalized against the first eleven columns. The thirteenth column is orthogonalized against the first twelve columns. The fourteenth column is orthogonalized against the first thirteen columns. The fifteenth column is orthogonalized against the first fourteen columns. The sixteenth column is orthogonalized against the first fifteen columns. The seventeenth column is orthogonalized against the first sixteen columns. The eighteenth column is orthogonalized against the first seventeen columns. The nineteenth column is orthogonalized against the first eighteen columns. The twentieth column is orthogonalized against the first nineteen columns. The twenty-first column is orthogonalized against the first twenty columns. The twenty-second column is orthogonalized against the first twenty-one columns. The twenty-third column is orthogonalized against the first twenty-two columns. The twenty-fourth column is orthogonalized against the first twenty-three columns. The twenty-fifth column is orthogonalized against the first twenty-four columns. The twenty-sixth column is orthogonalized against the first twenty-five columns. The twenty-seventh column is orthogonalized against the first twenty-six columns. The twenty-eighth column is orthogonalized against the first twenty-seven columns. The twenty-ninth column is orthogonalized against the first twenty-eight columns. The thirtieth column is orthogonalized against the first twenty-nine columns. The thirty-first column is orthogonalized against the first thirty columns. The thirty-second column is orthogonalized against the first thirty-one columns. The thirty-third column is orthogonalized against the first thirty-two columns. The thirty-fourth column is orthogonalized against the first thirty-three columns. The thirty-fifth column is orthogonalized against the first thirty-four columns. The thirty-sixth column is orthogonalized against the first thirty-five columns. The thirty-seventh column is orthogonalized against the first thirty-six columns. The thirty-eighth column is orthogonalized against the first thirty-seven columns. The thirty-ninth column is orthogonalized against the first thirty-eight columns. The fortieth column is orthogonalized against the first thirty-nine columns. The forty-first column is orthogonalized against the first forty columns. The forty-second column is orthogonalized against the first forty-one columns. The forty-third column is orthogonalized against the first forty-two columns. The forty-fourth column is orthogonalized against the first forty-three columns. The forty-fifth column is orthogonalized against the first forty-four columns. The forty-sixth column is orthogonalized against the first forty-five columns. The forty-seventh column is orthogonalized against the first forty-six columns. The forty-eighth column is orthogonalized against the first forty-seven columns. The forty-ninth column is orthogonalized against the first forty-eight columns. The fiftieth column is orthogonalized against the first forty-nine columns. The fifty-first column is orthogonalized against the first fifty columns. The fifty-second column is orthogonalized against the first fifty-one columns. The fifty-third column is orthogonalized against the first fifty-two columns. The fifty-fourth column is orthogonalized against the first fifty-three columns. The fifty-fifth column is orthogonalized against the first fifty-four columns. The fifty-sixth column is orthogonalized against the first fifty-five columns. The fifty-seventh column is orthogonalized against the first fifty-six columns. The fifty-eighth column is orthogonalized against the first fifty-seven columns. The fifty-ninth column is orthogonalized against the first fifty-eight columns. The sixtieth column is orthogonalized against the first fifty-nine columns. The sixty-first column is orthogonalized against the first sixty columns. The sixty-second column is orthogonalized against the first sixty-one columns. The sixty-third column is orthogonalized against the first sixty-two columns. The sixty-fourth column is orthogonalized against the first sixty-three columns. The sixty-fifth column is orthogonalized against the first sixty-four columns. The sixty-sixth column is orthogonalized against the first sixty-five columns. The sixty-seventh column is orthogonalized against the first sixty-six columns. The sixty-eighth column is orthogonalized against the first sixty-seven columns. The sixty-ninth column is orthogonalized against the first sixty-eight columns. The seventieth column is orthogonalized against the first sixty-nine columns. The seventy-first column is orthogonalized against the first seventy columns. The seventy-second column is orthogonalized against the first seventy-one columns. The seventy-third column is orthogonalized against the first seventy-two columns. The seventy-fourth column is orthogonalized against the first seventy-three columns. The seventy-fifth column is orthogonalized against the first seventy-four columns. The seventy-sixth column is orthogonalized against the first seventy-five columns. The seventy-seventh column is orthogonalized against the first seventy-six columns. The seventy-eighth column is orthogonalized against the first seventy-seven columns. The seventy-ninth column is orthogonalized against the first seventy-eight columns. The eightieth column is orthogonalized against the first seventy-nine columns. The eighty-first column is orthogonalized against the first eighty columns. The eighty-second column is orthogonalized against the first eighty-one columns. The eighty-third column is orthogonalized against the first eighty-two columns. The eighty-fourth column is orthogonalized against the first eighty-three columns. The eighty-fifth column is orthogonalized against the first eighty-four columns. The eighty-sixth column is orthogonalized against the first eighty-five columns. The eighty-seventh column is orthogonalized against the first eighty-six columns. The eighty-eighth column is orthogonalized against the first eighty-seven columns. The eighty-ninth column is orthogonalized against the first eighty-eight columns. The ninetieth column is orthogonalized against the first eighty-nine columns. The ninety-first column is orthogonalized against the first ninety columns. The ninety-second column is orthogonalized against the first ninety-one columns. The ninety-third column is orthogonalized against the first ninety-two columns. The ninety-fourth column is orthogonalized against the first ninety-three columns. The ninety-fifth column is orthogonalized against the first ninety-four columns. The ninety-sixth column is orthogonalized against the first ninety-five columns. The ninety-seventh column is orthogonalized against the first ninety-six columns. The ninety-eighth column is orthogonalized against the first ninety-seven columns. The ninety-ninth column is orthogonalized against the first ninety-eight columns. The hundredth column is orthogonalized against the first ninety-nine 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.

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

# Algorithm:

- Let  $a_{Kj}$  be the elements of A for  $k = 1, \dots, m$  and  $j = 1, \dots, n$ , 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
   $r_{ii} = \sqrt{r_{ii}}$ 
  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
end
```



# Algorithm in other notations :

## Vector notations:

```
R = 0
for i = 1, ..., n
    ! dot products, matrix-vector mult
    !  $R_{i,i:n} = Q_{1:m,i:n}^T \times Q_{1:m,i}$ 
    for j = i, ..., n
         $r_{ij} = Q_{1:m,i} \cdot Q_{1:m,j}$ 
    end
    ! normalize column i of Q
     $s = \sqrt{r_{ii}}$  // norm of i-th column of Q
     $Q_{1:m,i} = Q_{1:m,i} / s$ 

    ! compute projection factors
     $R_{i,i:n} = R_{i,i:n} / s$ 

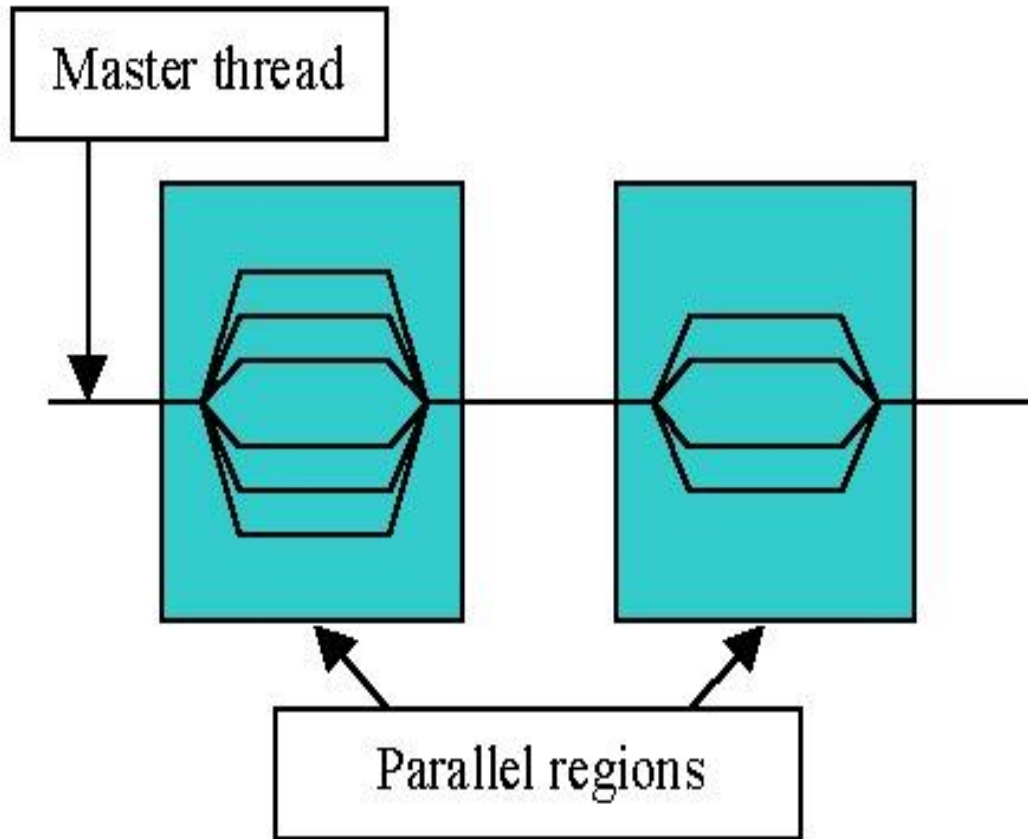
    ! orthogonalization, rank-1 update
    !  $Q_{1:m,i+1:n} = Q_{1:m,i} \times R_{i,i+1:n}^T$ 
    for j = i + 1, ..., n
         $Q_{1:m,j} = Q_{1:m,j} - r_{ij} Q_{1:m,i}$ 
    end
end
```

## using BLAS:

```
subroutine QR(q, r, m, n)
    integer i, m, n
    real q(m,n), r(m,n), S
    do i = 1, n
        !  $R_{i,i:n} = 1.0 * Q_{1:m,i:n}^T \times Q_{1:m,i} + 0.0 * R_{i,i:n}$ 
        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-i+1, alpha = S, x = r(i,i), incx = n)
        !  $Q_{1:m,i+1:n} = -1.0 * Q_{1:m,i} * R_{i,i+1:n}^T + Q_{1:m,i+1:n}$ 
        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
```

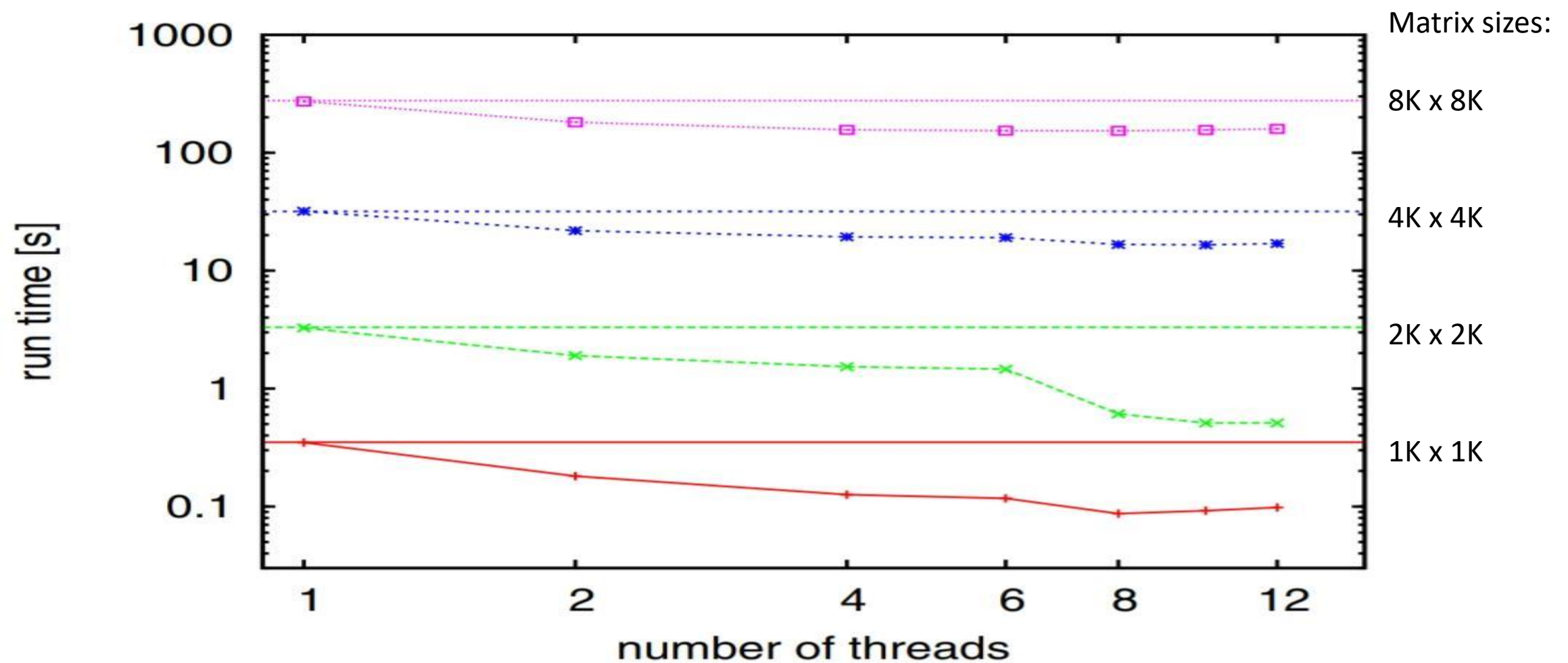
# CPU parallelization and blocking:

OpenMP parallelization:



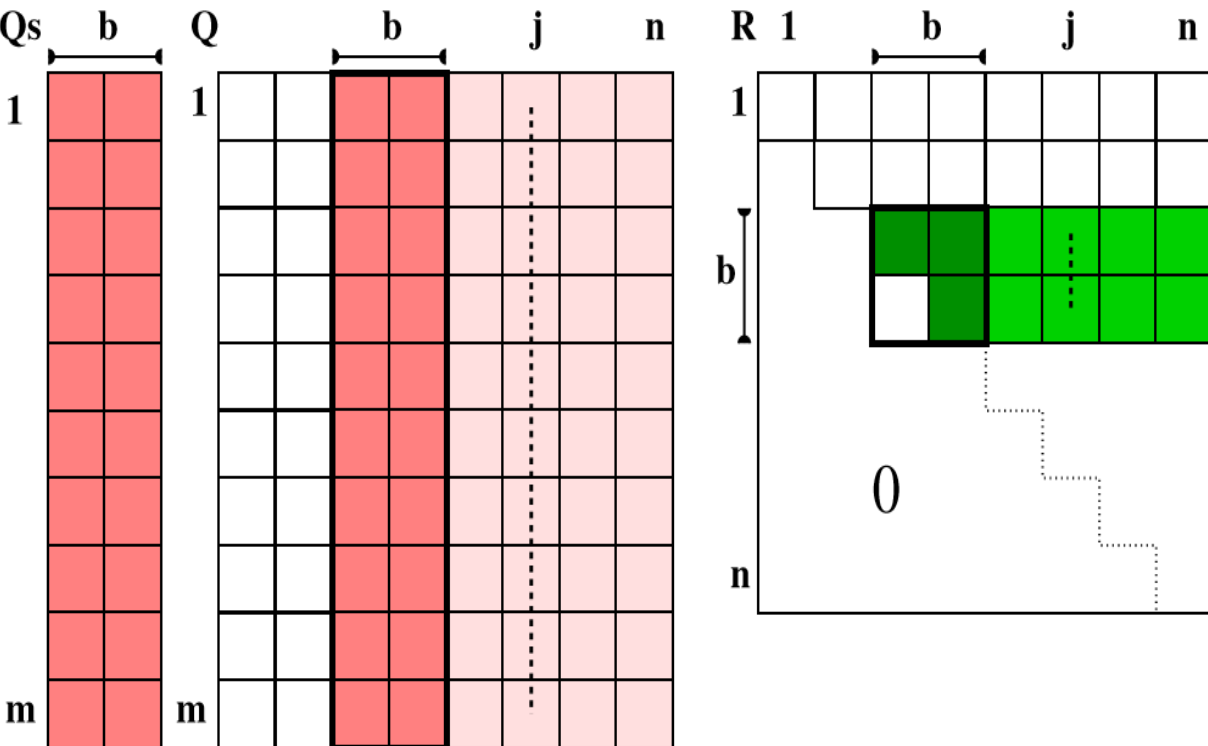
The outermost loop

```
R = 0
for  $i = 1, \dots, n$ 
  for  $j = i, \dots, n$ 
    for  $k = 1, \dots, m$ 
       $r_{ij} = r_{ij} + q_{ki}q_{kj}$ 
    end
  end
   $r_{ii} = \sqrt{r_{ii}}$ 
  for  $k = 1, \dots, m$ 
     $q_{ki} = q_{ki}/r_{ii}$ 
  end
  for  $j = i + 1, \dots, n$ 
     $r_{ij} = r_{ij}/r_{ii}$ 
  end
  for  $j = i + 1, \dots, n$ 
    for  $k = 1, \dots, m$ 
       $q_{kj} = q_{kj} - q_{ki}r_{ij}$ 
    end
  end
end
```



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:



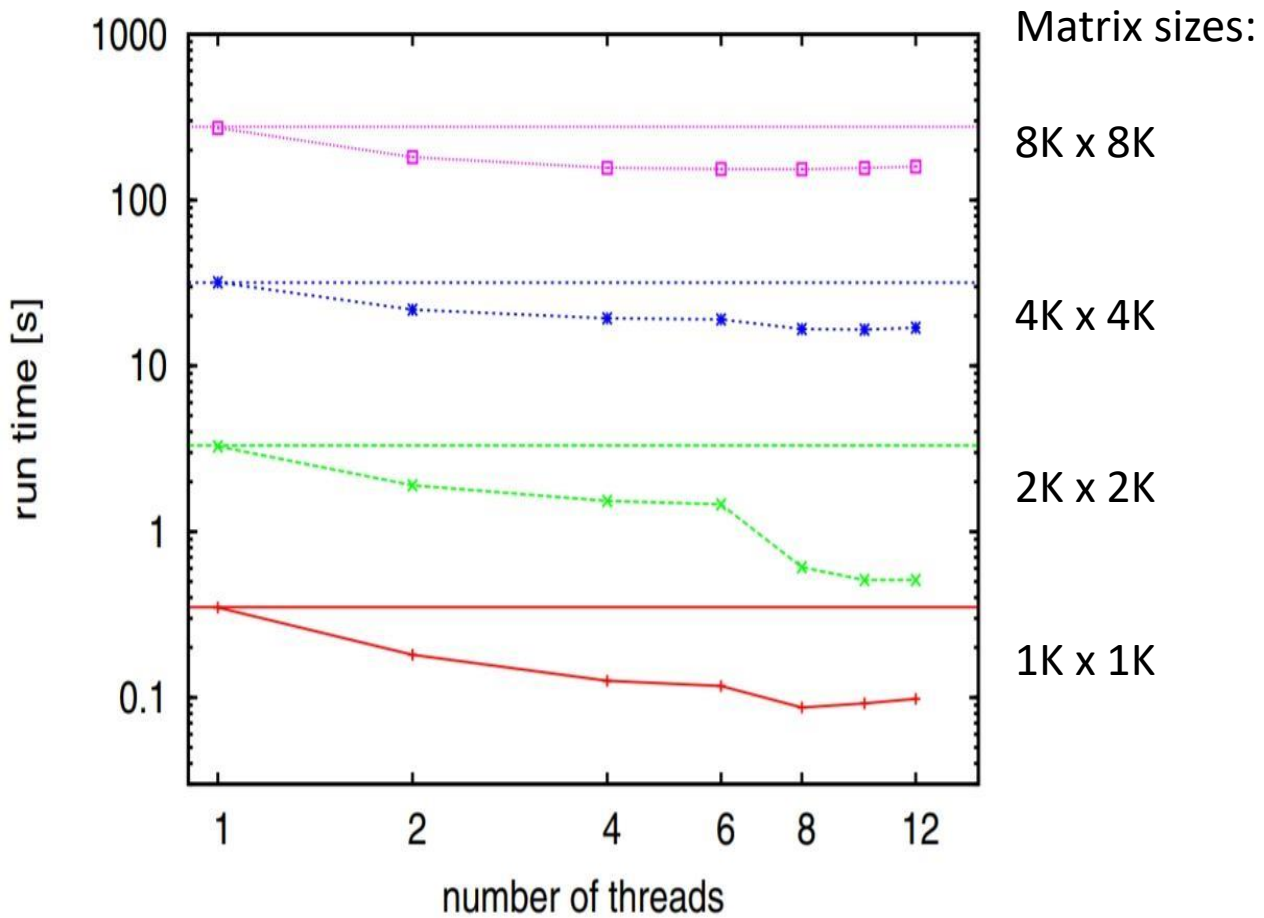
But How much should the size of  $b$ ??

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

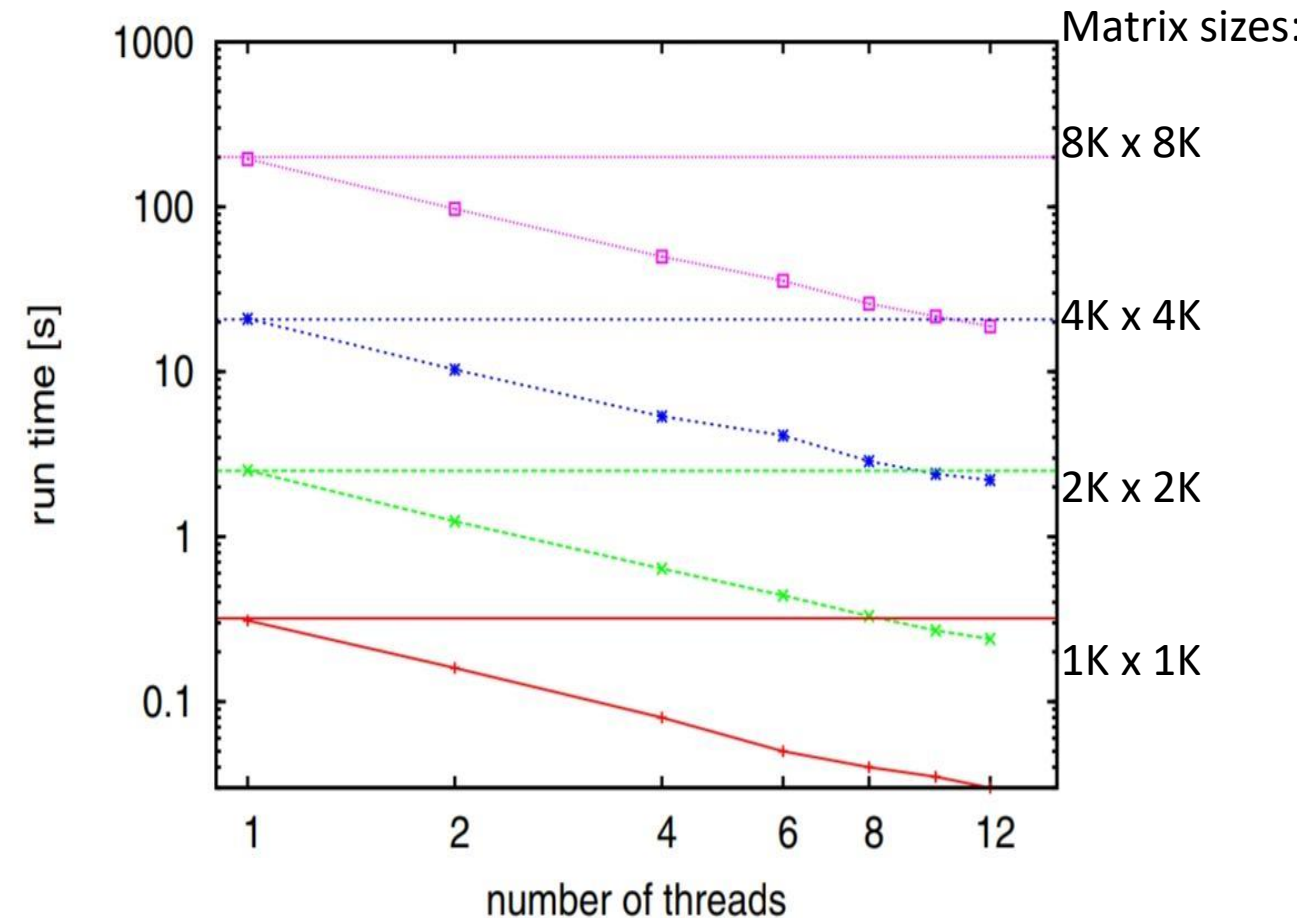
```

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

```

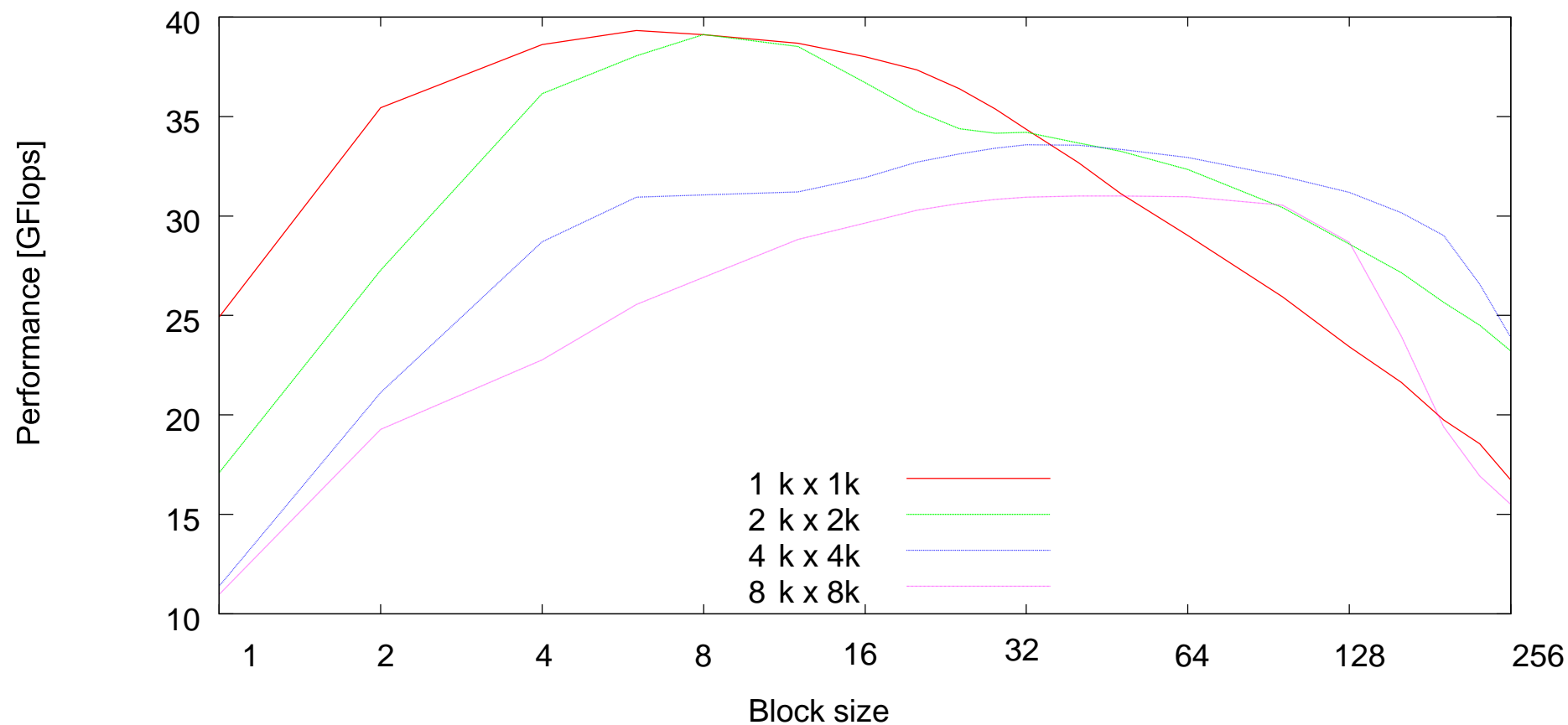


Without blocking.



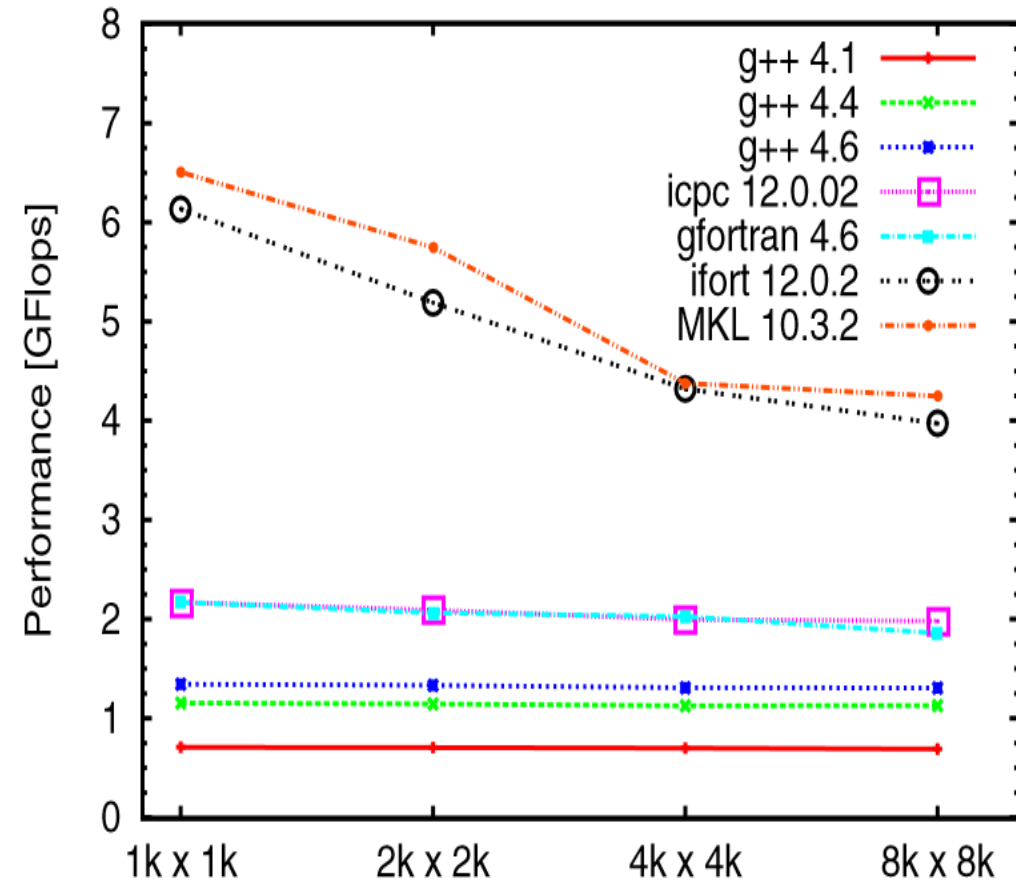
With blocking : this graph shows a nearly ideal scaling with the number of threads.

# The performance respecting to the size of b:





# Language and Compiler dependence:



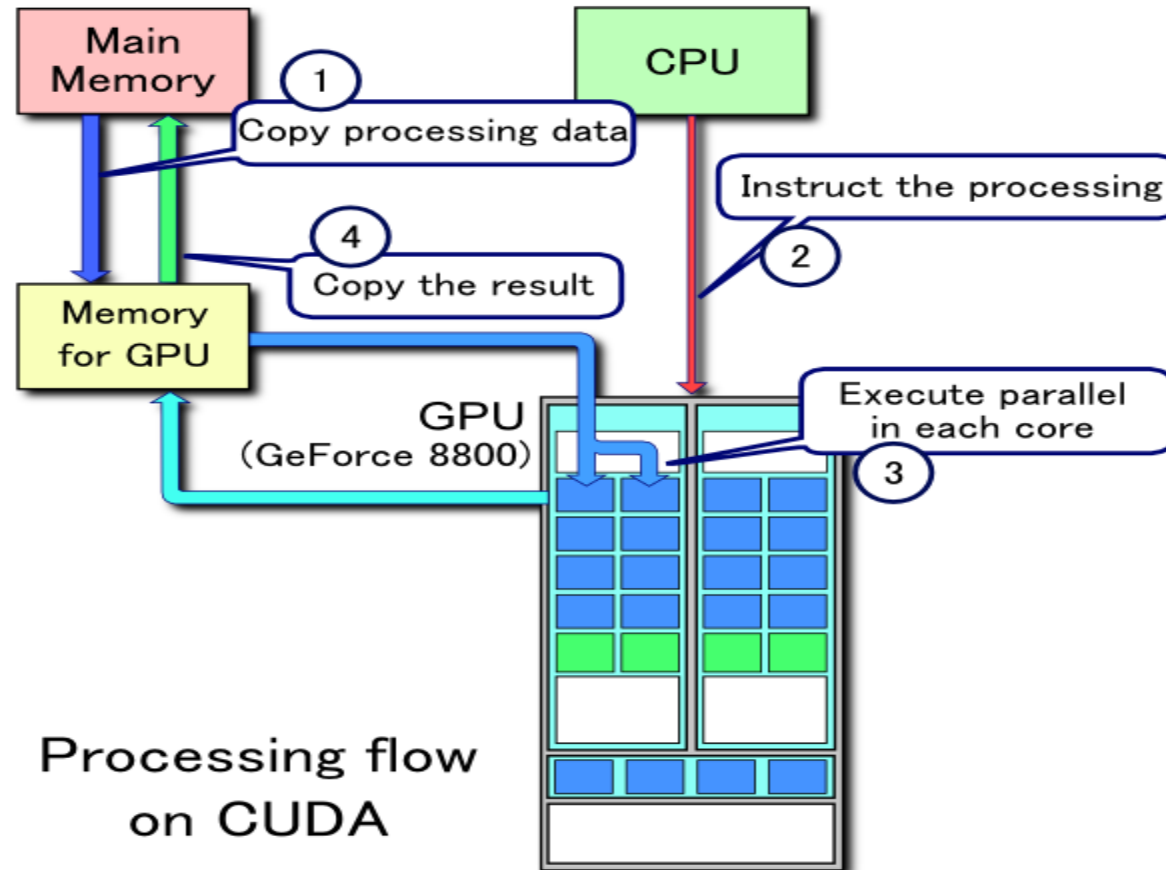
Performance in Gigaflops of a serial CPU version compiled using various compilers on a Intel Xeon X5650 @ 2.67 GHz CPU, depending on the matrix size.

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 x 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 = threadIdx.x;
    ty = threadIdx.y;
    j  = blockIdx * NY + ty + i + 1;

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

         $q_{kj} = q_{kj} - r_{ij}q_{ki};$ 

    }
}

```

Using global memory

```

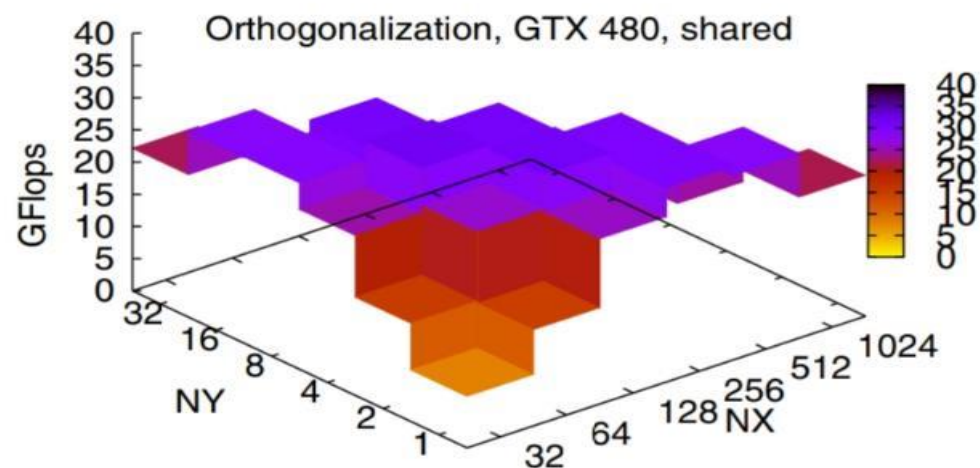
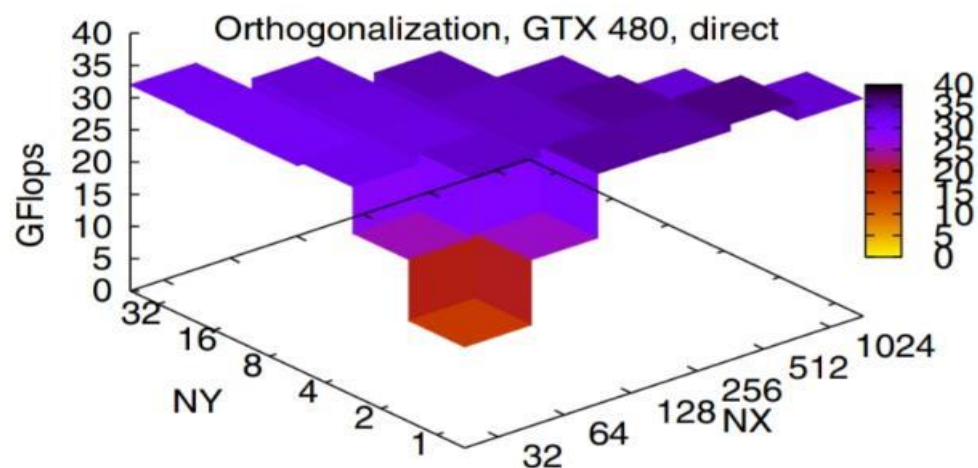
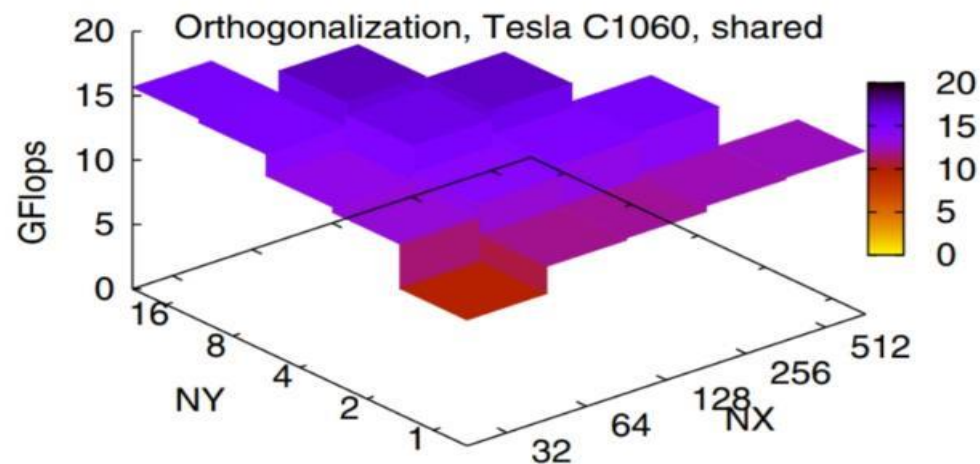
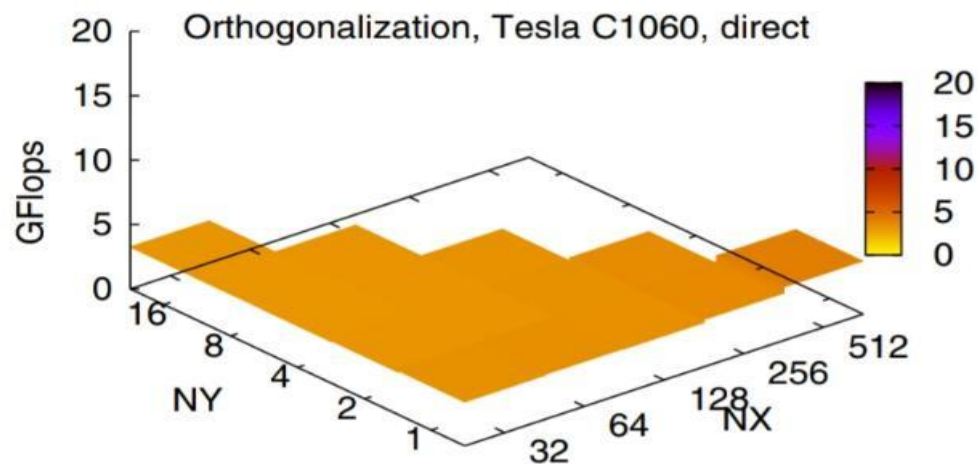
...
...
{
    __shared__ ri[NY], qi[NX];

    ...
    ...
    ...
    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];

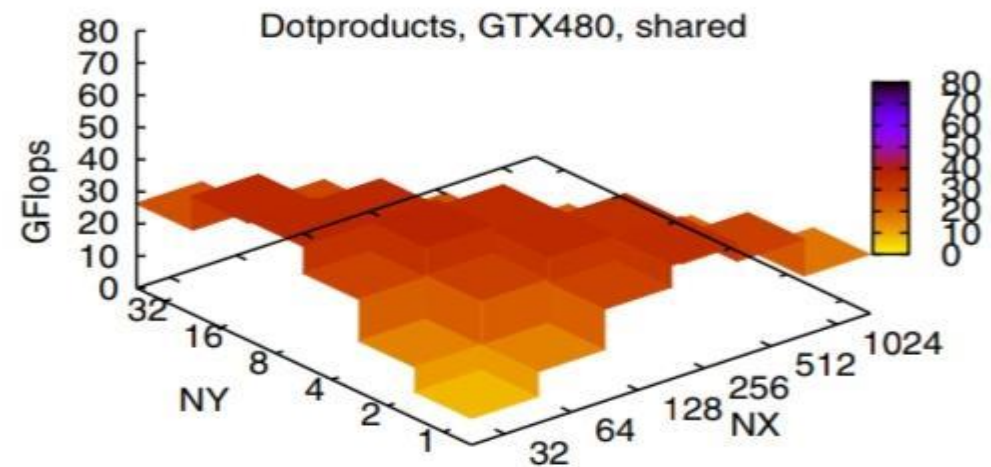
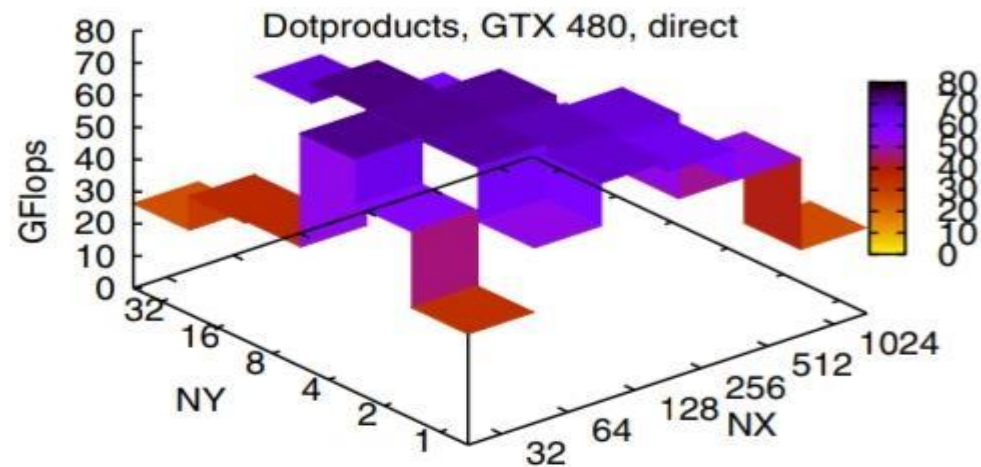
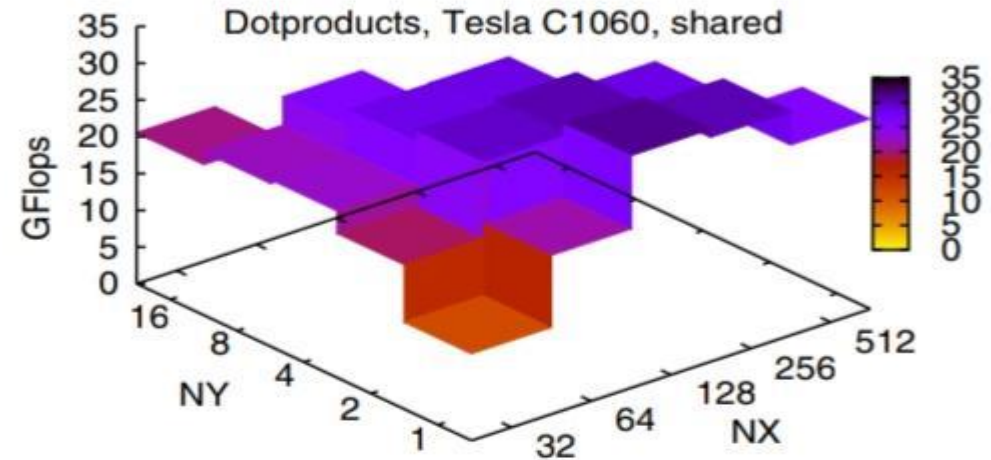
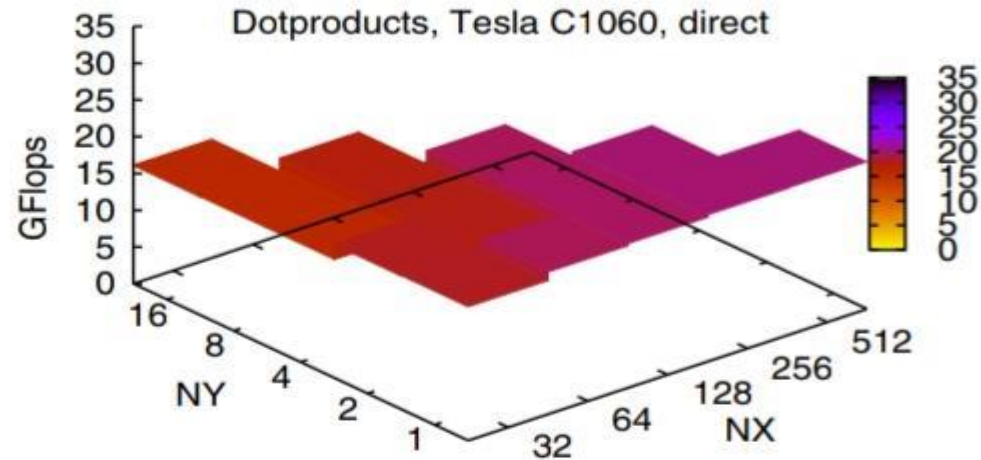
    tx = threadIdx.x;
    ty = threadIdx.x;
    j = blockIdx * NY + ty + i;
    sum = 0;
    for (k=tx+1; k<=m; k+=NX) {

        sum +=  $q_{ki}q_{kj}$ ;
    }
    // reduction:  $r_{ij} += \text{sum}$ ;
    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)  $r_{i,j} = \text{RS}[ty][0]$ ;
}
```

Using shared memory

```
...
...
{
    ...
    __shared__ qi[NX];
    ...
    ...
    ...
    for (...) {
        if (ty==0)  $qi[tx] = q_{ki}$ ;
        __syncthreads();
        sum +=  $qi[tx] * q_{kj}$ 
    }
    ...
    ...
    ...
    ...
    ...
    ...
    ...
}
```

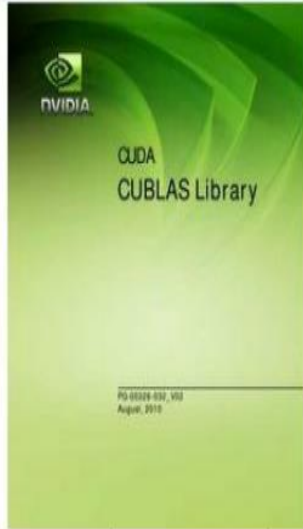
# Comparison of the performance between Tesla C1060 and GTX 480 using direct and shared memory:





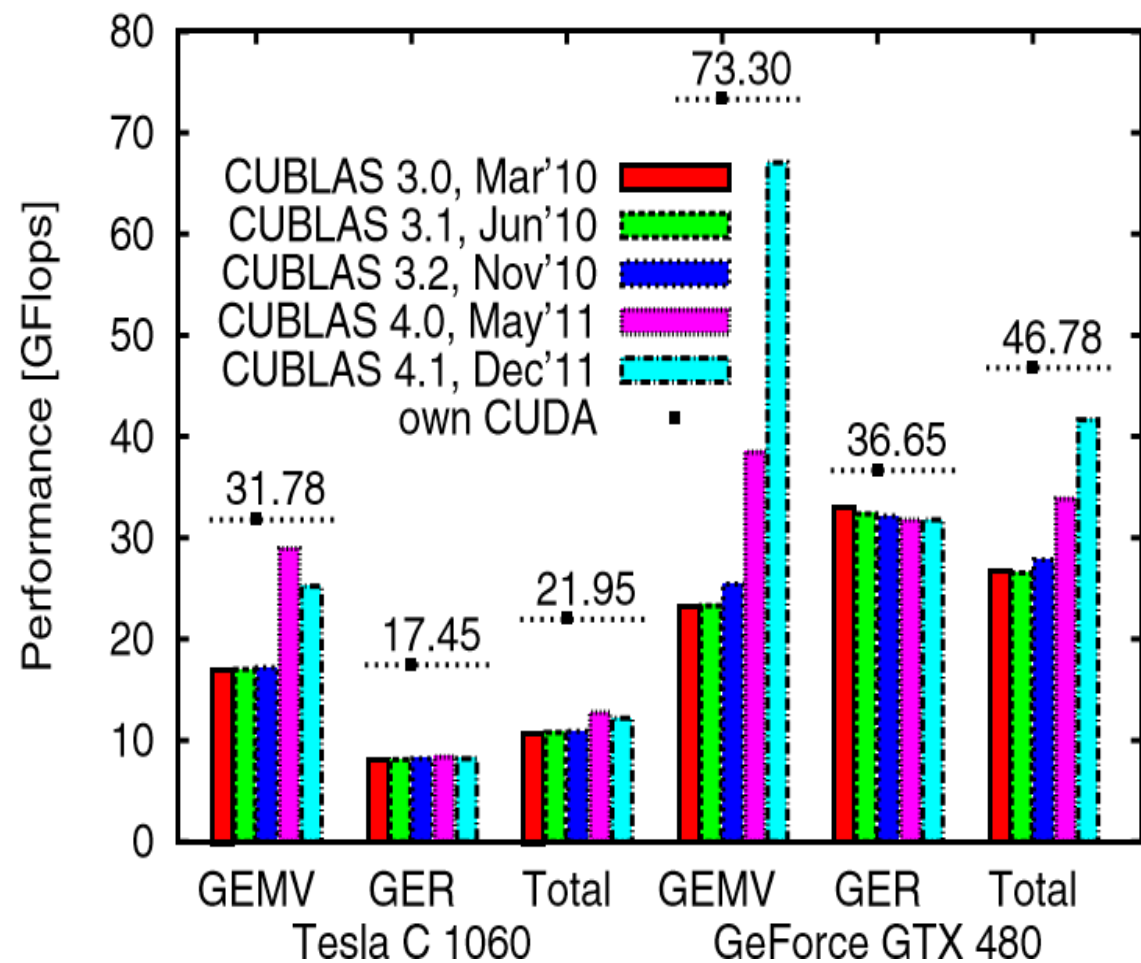
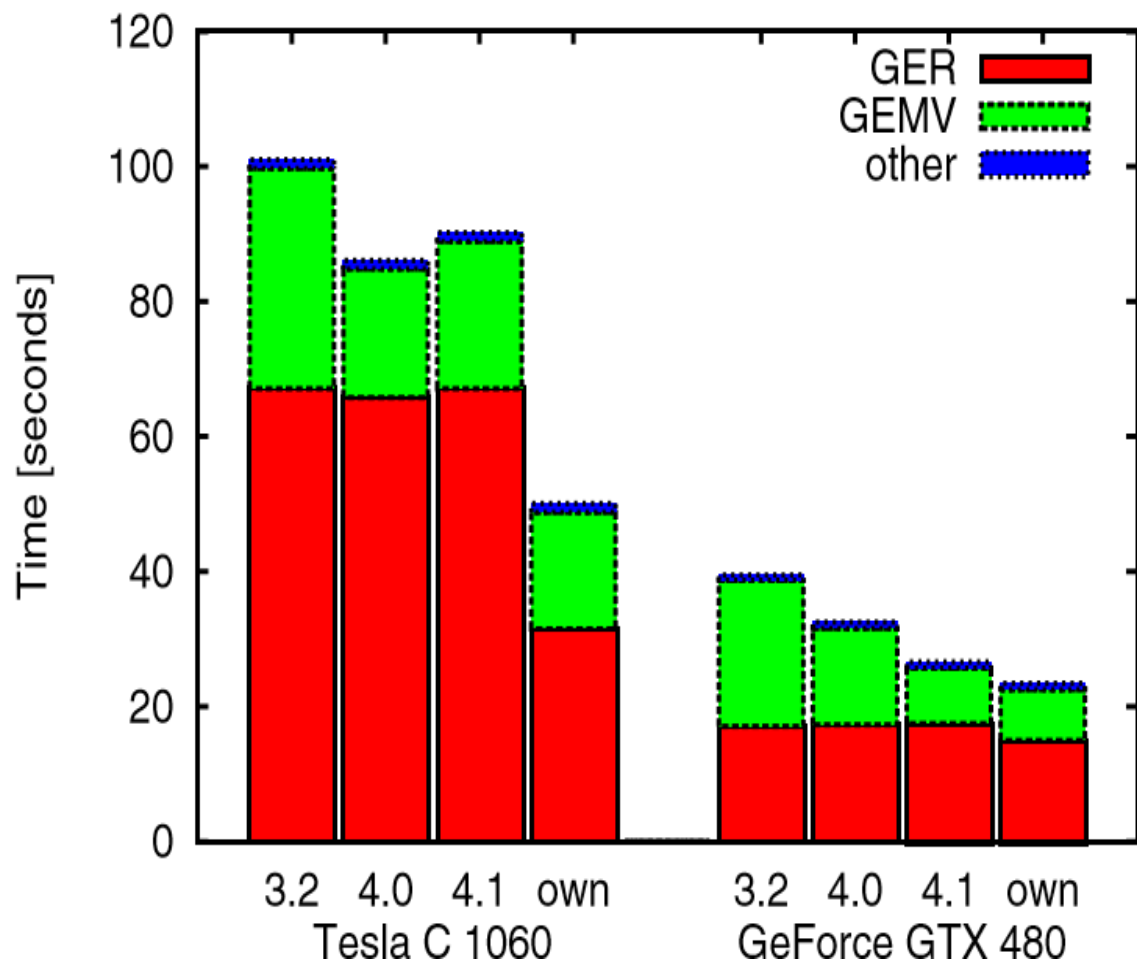
# What is CUBLAS Library?

- 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 streams
  - 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.

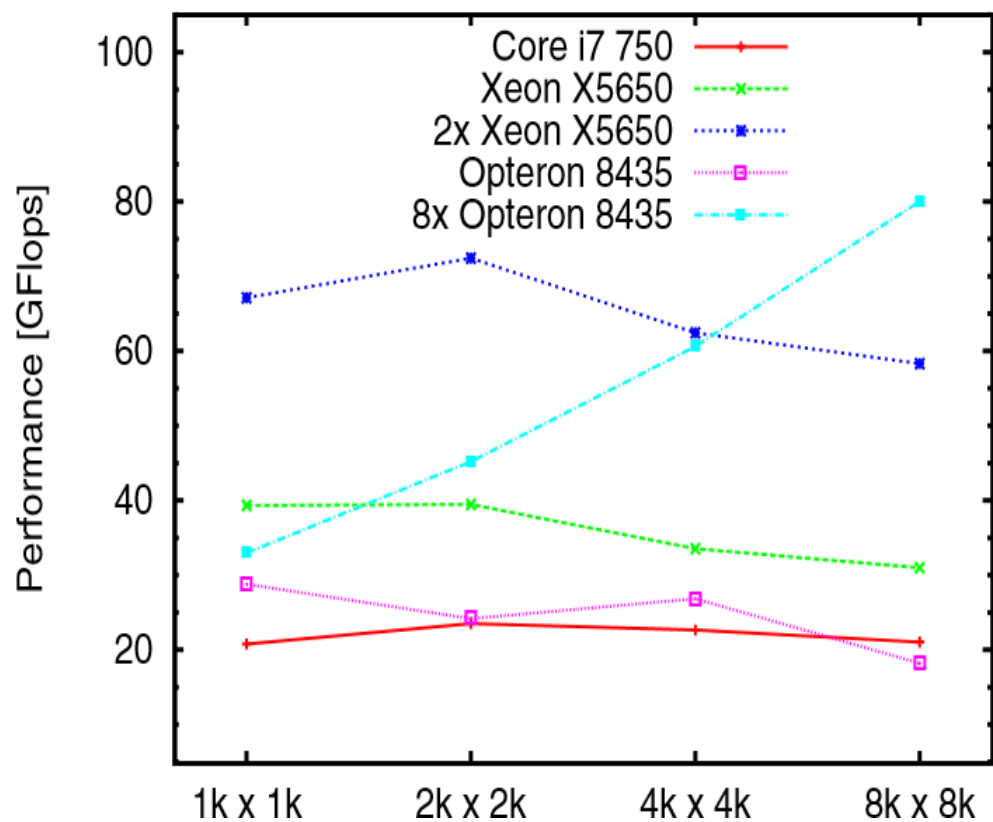
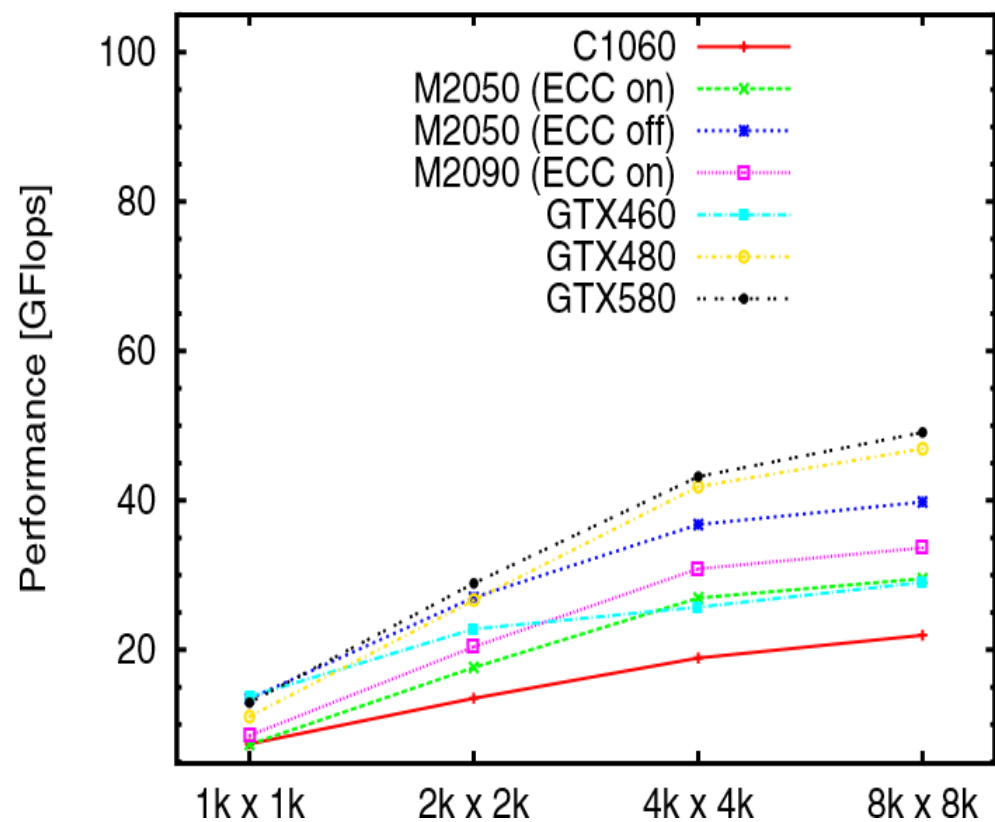
Comparison of  
CPU and GPU :

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)