# グリッドコンピューティング

2012/1/16

福田圭祐（11M37264 松岡研究室）

# 紹介論文

## Efficient Support for Matrix Computations on Heterogeneous Multi-core and Multi-GPU Architectures

Fengguang Song†, Stanimire Tomov† and Jack Dongarra†‡

† University of Tennessee

‡Oak Ridge National Laboratory, University of Manchester

# Abstract

- Exploiting heterogeneous systems is a challenging task
- Designed a heterogeneous algorithm for linear algebra on a Multi-core and Multi-GPU system
- Developed a runtime system to support the algorithm
- Applied it to QR and Cholesky factorization and achieved good scalability and load-balancing

# Background :LA for CPU and GPU

- Linear algebra (LAPACK and BLAS) on CPU/GPU :
  - PLASMA [1] for multicore CPUs
  - MAGMA [2] for a GPU

(Both are developed in University of Tennessee)

# Background : Tile algorithms

- Tile algorithms
  - Divides a matrix A ($n \times n$) into a number of $b \times b$ submatrices

$$\begin{pmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,n_b} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,n_b} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n_b,1} & A_{n_b,2} & \cdots & A_{n_b,n_b} \end{pmatrix}$$

  - Great amount of parallelism

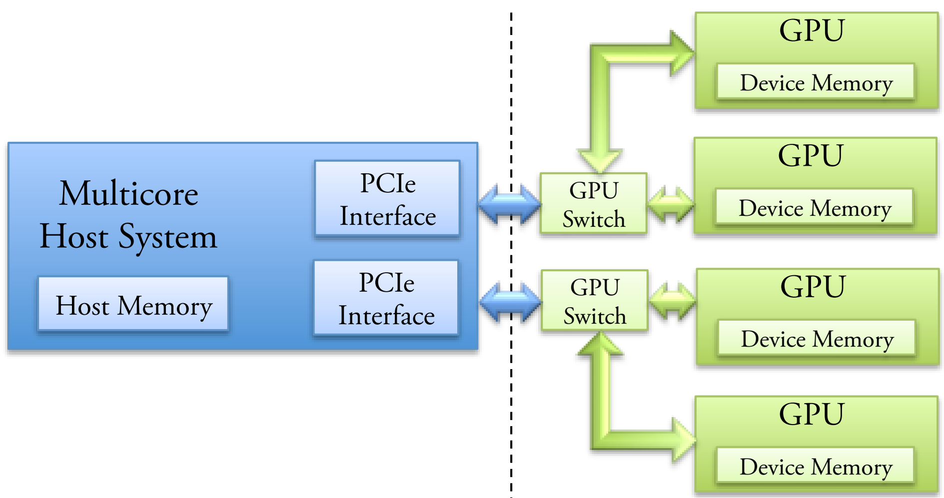
# Background : Tile algorithms

- Ex. QR factorization
  1. Compute a QR factorization of $A_{1,1}$
  2. Output of 1. is used to update $A_{1,2}$, … $A_{1,nb}$ in embarrassingly parallel way
     (nb = n/b)
  3. After updating all $A_{i,2} \sim A_{i,nb}$, start again from $A_{2,1}$ …

$$\begin{pmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,n_b} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,n_b} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n_b,1} & A_{n_b,2} & \cdots & A_{n_b,n_b} \end{pmatrix} \begin{pmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,n_b} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,n_b} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n_b,1} & A_{n_b,2} & \cdots & A_{n_b,n_b} \end{pmatrix} \begin{pmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,n_b} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,n_b} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n_b,1} & A_{n_b,2} & \cdots & A_{n_b,n_b} \end{pmatrix} \begin{pmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,n_b} \\ A_{2,1} & A_{2,2} & \cdots & A_{2,n_b} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n_b,1} & A_{n_b,2} & \cdots & A_{n_b,n_b} \end{pmatrix}$$

# Algorithm : Challenges for Heterogeneous systems

- Utilizing both of CPUs and GPUs is challenging:
  1. GPUs have different memory spaces
  2. Each GPU has  to be controlled by a host thread
  3. Processor heterogeneity



# Algorithm : Challenges for Heterogeneous systems

- Utilizing both of CPUs and GPUs is challenging:
  4. GPUs are optimized for throughput and expect large data size, while CPUs are optimized for latency
  5. Performance gap between computation and communication

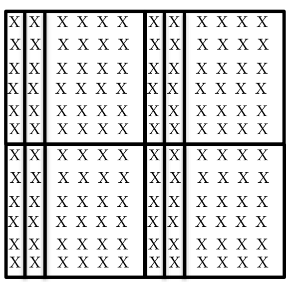
# Algorithm : New algorithm

- New algorithm:

  "Heterogeneous rectangular tile algorithm"
  - Based on tile algorithm
  - With hybrid tile sizes
  - Great emphasis on minimal communication
  - 1-D block cyclic data distribution
  - Auto-tuning to determine optimal tile size

# Algorithm : Tiling

- Hybrid size rectangular tiles



(a)

$$\left(\begin{array}{c|c|c} \overbrace{\begin{matrix} a_{11} & a_{12} \ldots & A_{1s} \\ a_{21} & a_{22} \ldots & A_{2s} \\ & \vdots & \\ a_{p1} & a_{p2} \ldots & A_{ps} \end{matrix}}^{B} & \overbrace{\begin{matrix} a_{1(s+1)} & a_{1(s+2)} \ldots & A_{1(2s)} \\ a_{2(s+1)} & a_{2(s+2)} \ldots & A_{2(2s)} \\ & \vdots & \\ a_{p(s+1)} & a_{p(s+2)} \ldots & A_{p(2s)} \end{matrix}}^{B} & \begin{matrix} \ldots \\ \ldots \\ \ddots \\ \ldots \end{matrix} \end{array}\right)$$

$a_{i,j} : B \times b$

$A_{i,j} : B \times (B - b(s-1))$

  - At the top level, a matrix is divided into rectangular tiles of size *B*×*B*
  - Each tile is subdivided into a number of small rectangular size of *B*×*b* and a remaining tile.
  - Assuming $n = pB$ and $B > b$

# Algorithm : Cholesky factorization

---

**Algorithm 1** Rectangular Tile Cholesky Factorization

```
for t ← 1 to p do
   for d ← 1 to s do
      k ← (t - 1) * s + d /* the k-th tile column */
      Δ ← (d - 1) * b /* local offset within a tile */
      POTF2'(A_tk[Δ,0], L_tk[Δ,0])                                  ←(b)
      for j ← k + 1 to t * s /* along the t-th tile row */ do
         GSMM(L_tk[Δ+b,0], L_tk[Δ+(j-k)*b,0], A_tj[Δ+b,0])          ←(c)
      end for
      for i ← t + 1 to p /* along the k-th tile column */ do
         TRSM(L_tk[Δ,0], A_ik, L_ik)                                ←(d)
      end for
      /* trailing submatrix update */
      for i ← t + 1 to p do
         for j ← k + 1 to i * s do
            j' = ⌈j/s⌉
            if (j' = t) GSMM(L_ik, L_tk[Δ+(j-k)%s*b,0], A_ij)       ←(e)
            else GSMM(L_ik, L_j'k[(j-1)%s*b,0], A_ij)
         end for
      end for
   end for
end for
```

$A_{ij}$[i,j] : $A_{ij}$'s submatrix that starts from its local x-th row and y-th column to its original bottom right corner

# Algorithm : Cholesky factorization

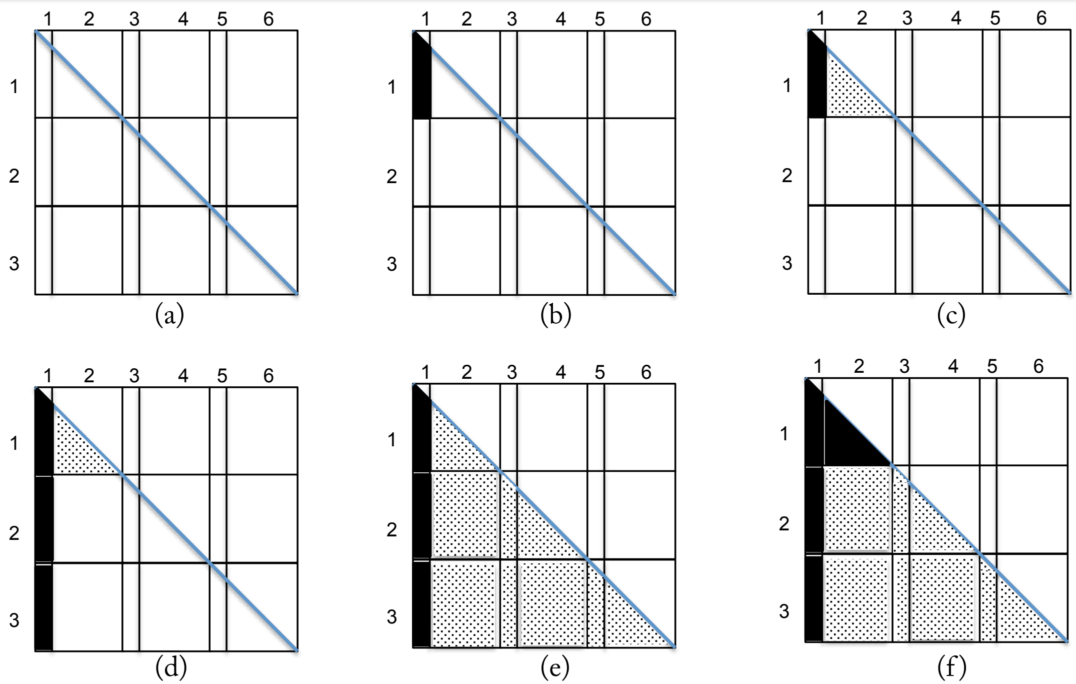
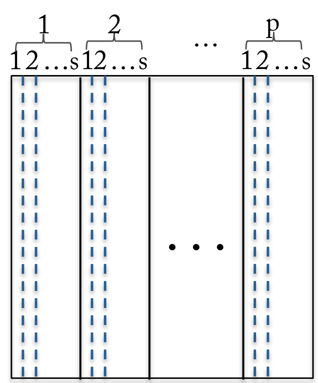



**Figure 4: The operations of heterogeneous rectangular tile Cholesky factorization. (a)** The symmetric positive definite matrix A. **(b)** Compute POTF2' to solve $L_{11}$. **(c)** Apply $L_{11}$ to update its right $A_{12}$ by matrix multiplication. **(d)** Compute TRSMs for all tiles below $L_{11}$ to solve $L_{21}$ and $L_{31}$. **(e)** Apply GSMMs to update all tiles on the right of TRSMs. **(f)** At the 2nd iteration, we repeat performing (b), (c), (d), (e) on the trailing submatrix that starts from the 2nd tile column.

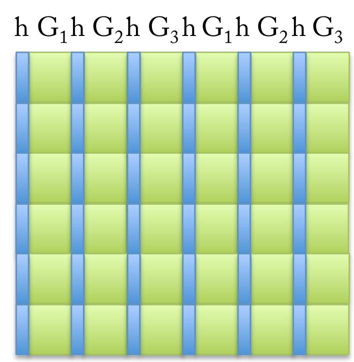
# Algorithm : Heterogeneous block cyclic distribution

- A matrix is divided $p \times (s \bullet p)$ tiles
- Distribute the columns to the host and P GPUs.
  - $P_0$ : host, $P_{x\geq 1}$ : x-th GPUs
  - 1-D heterogeneous cyclic distribution

$$x = \begin{cases} ((\frac{j}{s} - 1) \mod P) + 1 & : \quad j \mod s = 0 \\ 0 & : \quad j \mod s \neq 0 \end{cases}$$




(a)




(b)

# Algorithm : Communication cost

The number of words communicated by
at least one of the process:

$$\text{Word} \quad = \quad \sum_{k=0}^{p-1}(n-kB)B\log(P+1) \simeq \frac{n^2}{2}\log(P)$$

We can attain ScaLAPACK's $O(\frac{n^2}{\sqrt{P}}\log P)$ by using 2-D
Cyclic distribution algorithm, but 1-D is practically faster.

The number of messages communicated by
At least one of the process:

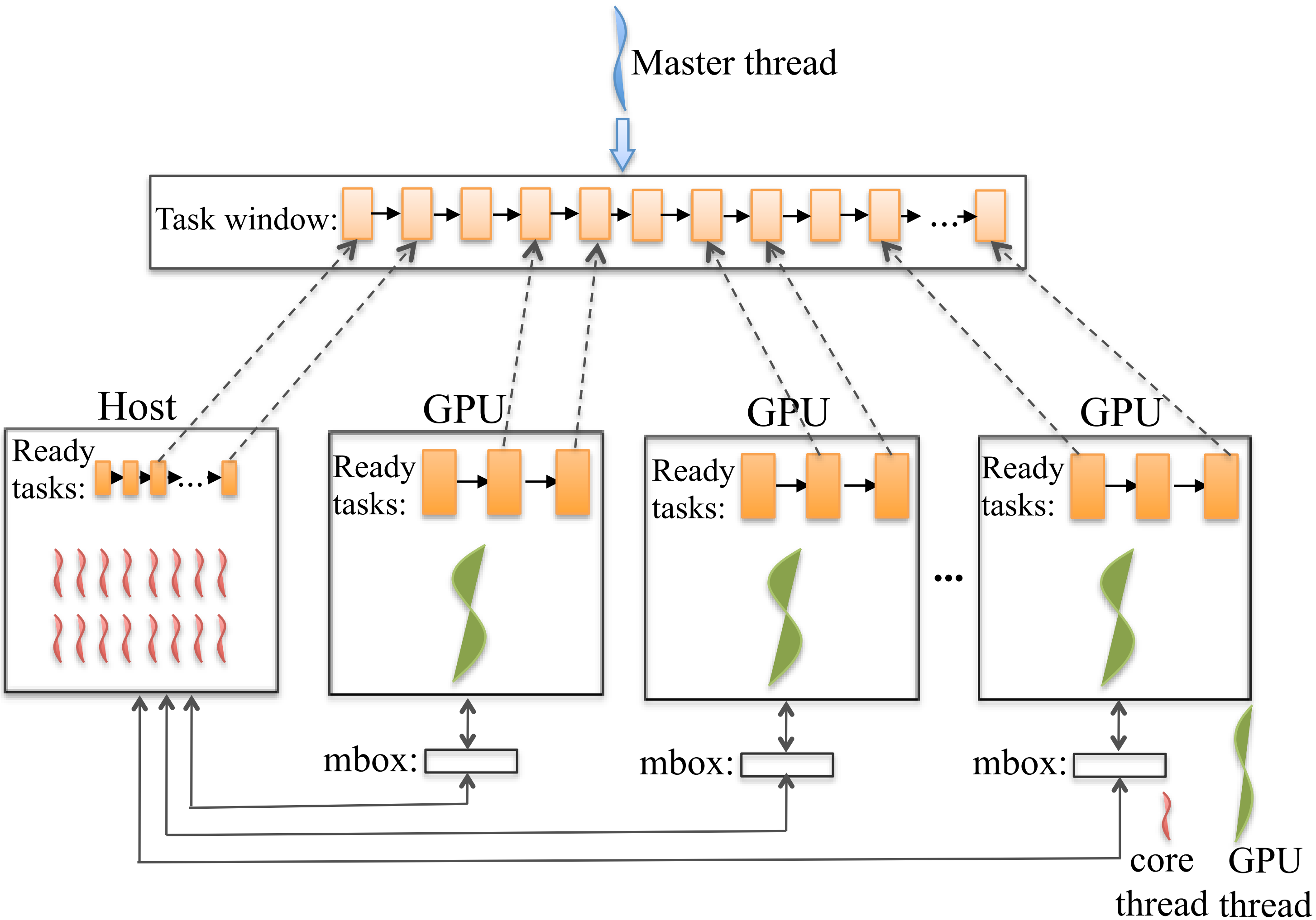
$$\text{Message} \quad = \quad \sum_{k=0}^{p-1}(p-k)s\log(P+1) \simeq \frac{p^2 s}{2}\log(P)$$

It is larger than ScaLAPACK's O(P), but each message is smaller
And we can get higher degree of parallelism

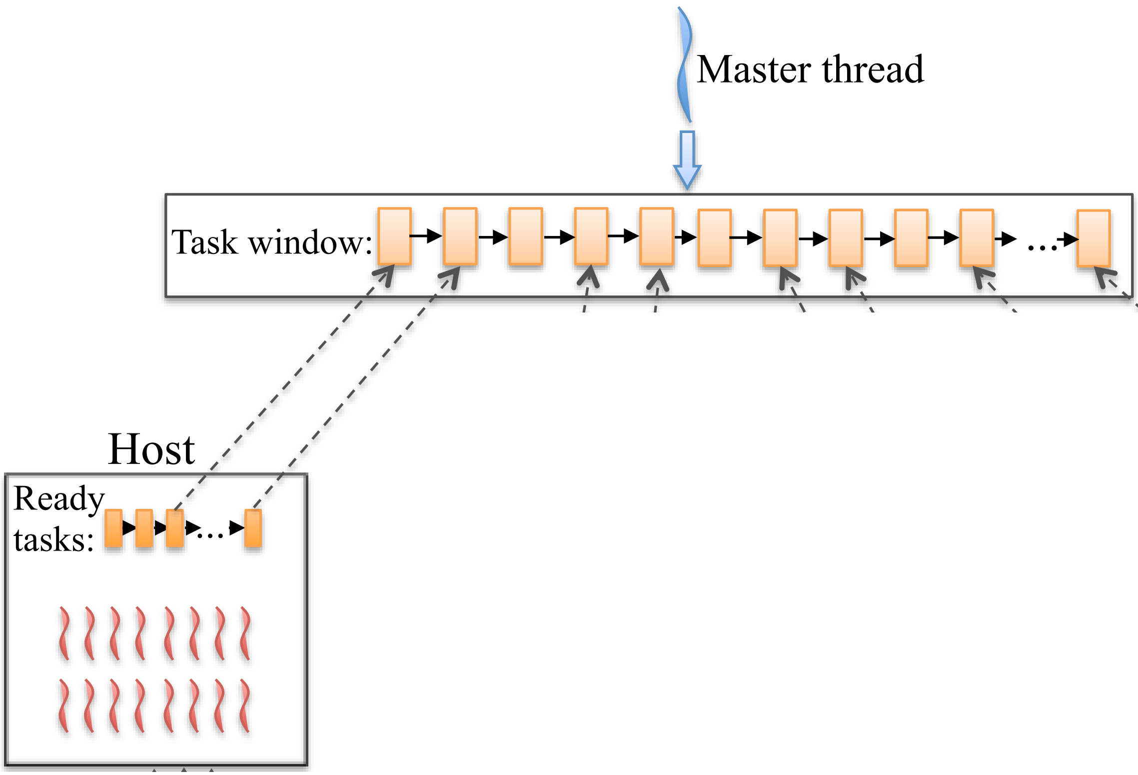
# Implementation

- The runtime system is based on the author's centrarized-version runtime system[3]
- 4 components:
  - Master thread
  - Task window
  - Ready task queue
  - Computational thread

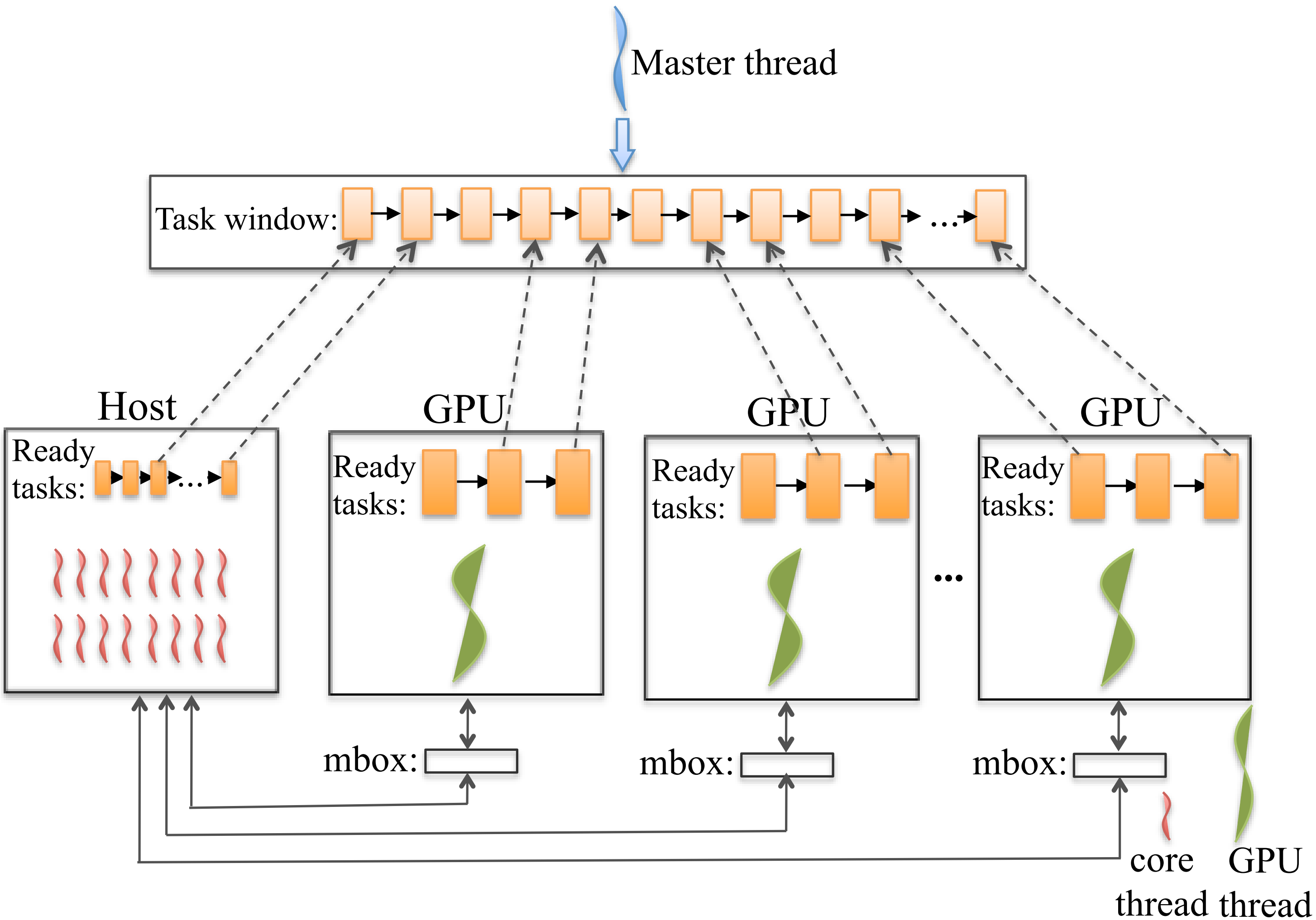
# Implementation : Centralized–version runtime system



# Implementation : Centralized–version runtime system



# Implementation : Centralized–version runtime system

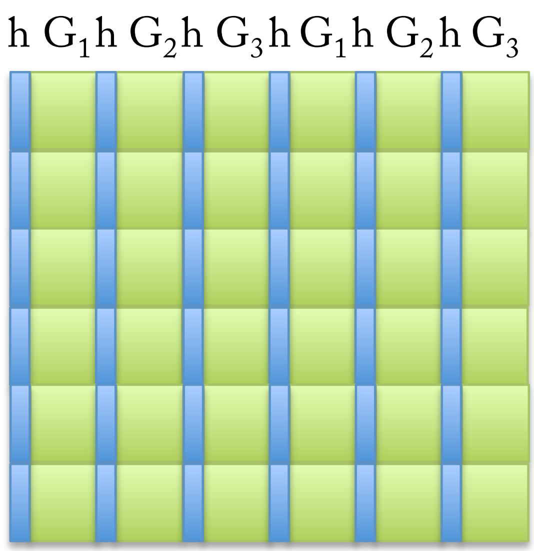


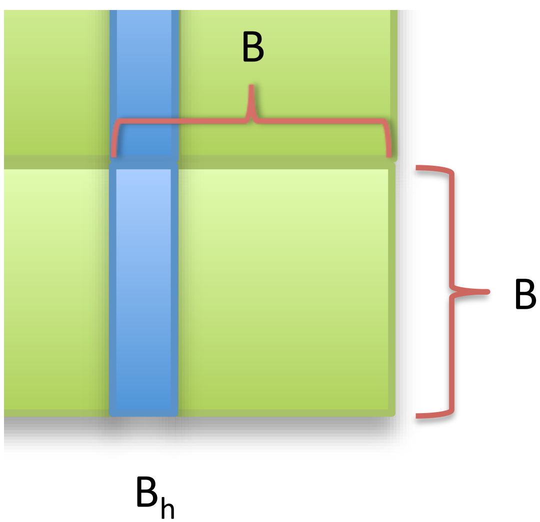
# Implementation : Centralized–version runtime system

- Each GPU belongs to 1 host thread and the thread manages the GPU including all communications (GPU-host, host-GPU).
- If the host has n cores, (n-P) cores are used to compute

# Implementation : Auto-tuning of tile size

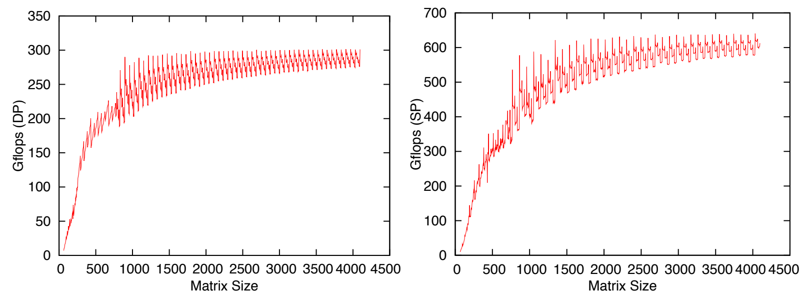
- Two steps of auto-tuning
  1. Determine B
  2. Determine $B_h$ : to be cut off from the top level BxB tiles



# Implementation : Auto-tuning of tile size

- To find best B:
  - Search for minimal matrix size that provides the maximum performance for the dominant GPU kernel (i.e. GEMM for Cholesky and SSFRB for QR)
  - Search range is 128 to 2048, (ex. 980 for Fermi GPUs)




(a) Double precision (b) Single precision

**Figure 3: Matrix multiplication with CUBLAS 3.2 on an Nvidia Fermi M2070 GPU. (a)** The maximum

# Implementation : Auto-tuning of tile size

- To find best $B_h$:
  1. Estimate $B_h$

$$B_h = \frac{\mathrm{Perf}_{core} \cdot \#Cores}{\mathrm{Perf}_{core} \cdot \#Cores + \mathrm{Perf}_{gpu} \cdot \#GPUs} \cdot B$$

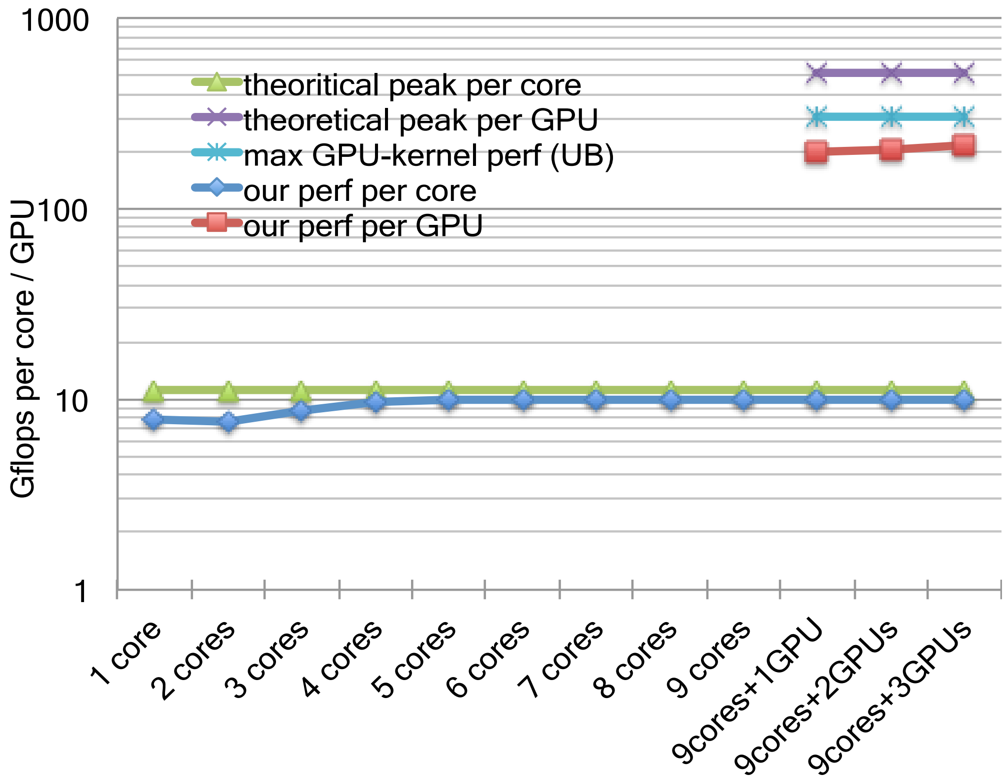
  2. Search for an optimal $B_h^*$ near $B_h$:
     - Run a script that execute QR and Cholesky factorization with a random matrix of size $N = c_0 \cdot B \cdot \#GPU$
     - It searches for a minimal differenct of CPU and GPU computation time.

# Performance Evaluation: settings and upper bound

**Table 1: Experiment Environment**

| | Host | Attached GPUs |
|---|---|---|
| Processor type | Intel Xeon X5660 | Nvidia Fermi M2070 |
| Clock rate | 2.8 GHz | 1.15 GHz |
| Processors per node | 2 | 3 |
| Cores per processor | 6 | 14 SMs |
| Memory | 24 GB | 6 GB per GPU |
| Theo. peak (double) | 11.2 Gflops/core | 515 Gflops/GPU |
| Theo. peak (single) | 22.4 Gflops/core | 1.03 Tflops/GPU |
| Max gemm (double) | 10.7 Gflops/core | 302 Gflops/GPU |
| Max gemm (single) | 21.4 Gflops/core | 635 Gflops/GPU |
| Max ssrfb (double) | 10.0 Gflops/core | 223 Gflops/GPU |
| Max ssrfb (single) | 19.8 Gflops/cores | 466 Gflops/GPU |
| BLAS/LAPACK lib | Intel MKL 10.3 | CUBLAS 3.2, MAGMA |
| Compilers | Intel compilers 11.1 | CUDA toolkit 3.2 |
| OS | CentOS 5.5 | Kernel module 260.19.14 |
| System interface | – | PCIe x 16 Gen2 |

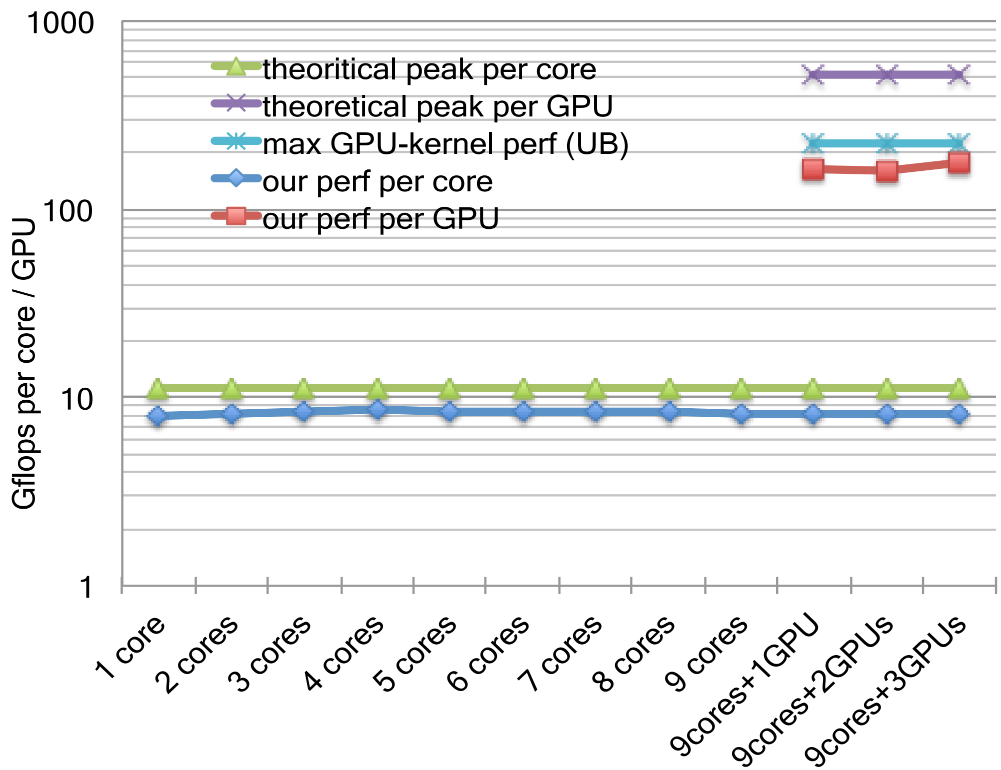
# Performance Evaluation : weak scaling




(a) Cholesky in double precision

9 cores and 3 GPUs : 742 Gflops ( 74% upper bound & 45% of the theoretical peak)

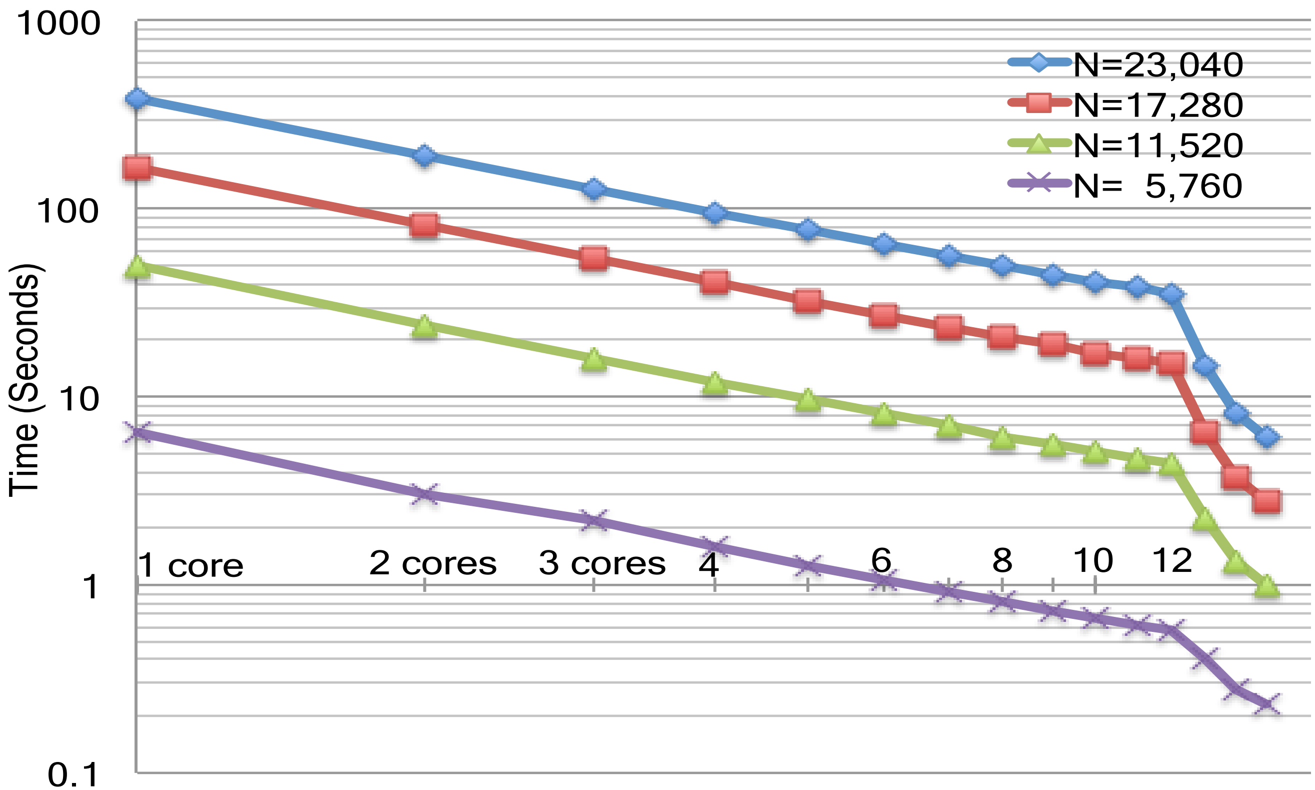
# Performance Evaluation : weak scaling




(b) QR in double precision

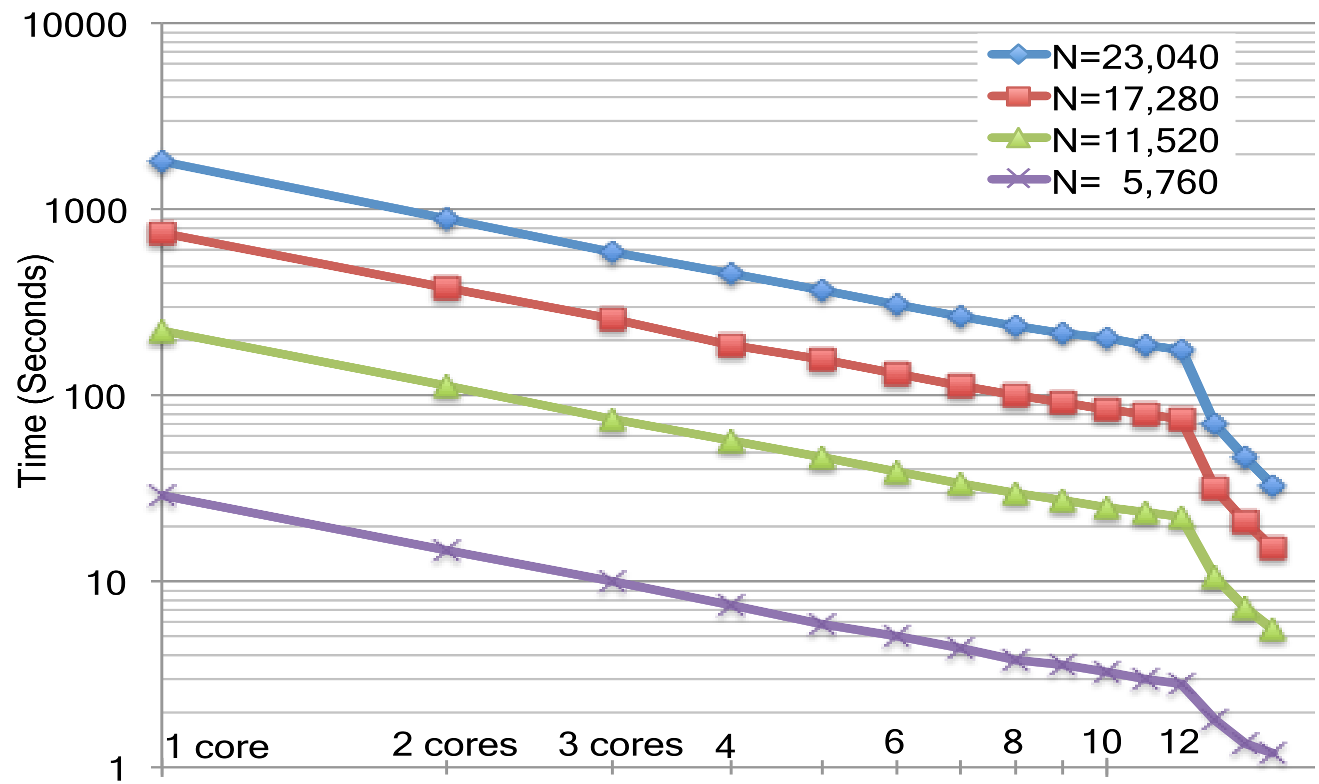
9 cores and 3 GPUs : 79% upper bound

# Performance Evaluation : strong scaling




(a) Cholesky in double precision

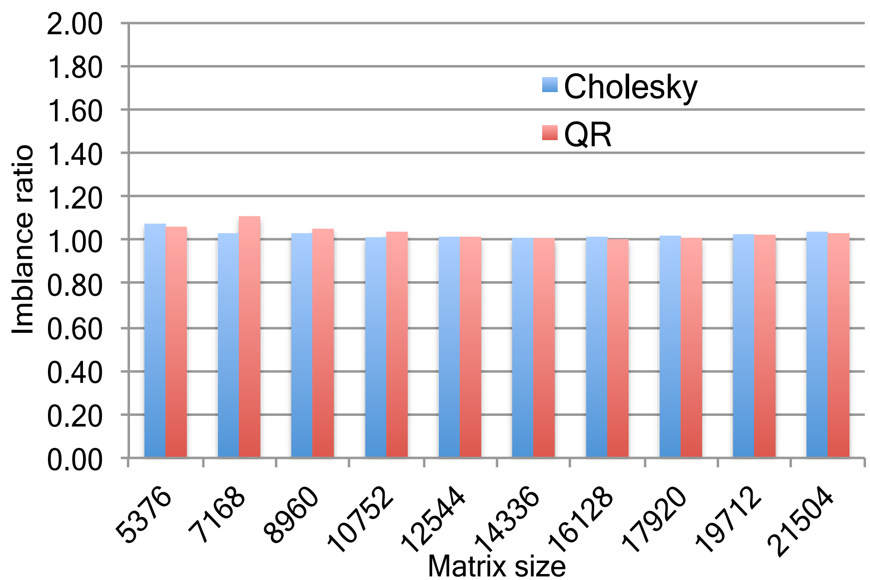
# Performance Evaluation : strong scaling




(b) QR in double precision

# Performance evaluation: Load balancing

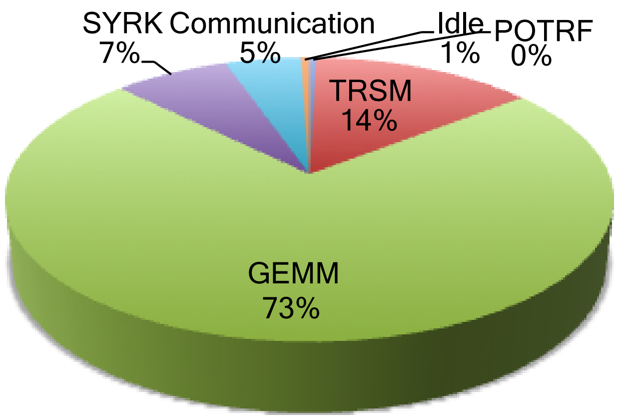
- Imbalance ratio = $\dfrac{MaxLoad}{AvgLoad}$ (load : time[s])



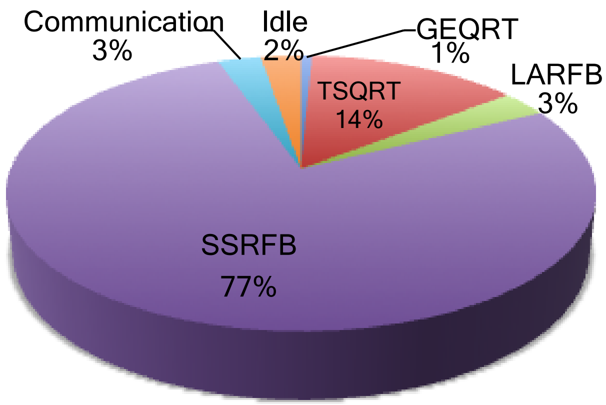

(a) 4Cores+1GPU (double)

- Most of the imbalance ratio is less than 5%
- A few is up to 17% (because of too few toplevel tiles)

# Performance evaluation: Runtime System Efficiency




(a) Cholesky factorization




(b) QR factorization

**Figure 10: Execution time break down on a GPU for double precision Cholesky and QR factorizations.**

Cholesky : Kernels (94%), communication(5%), idle(1%)
QR : Kernels (95%), communication(3%), idle(2%)

# Conclution

- Exploiting heterogeneous systems is challenging because of processor heterogeneity, separated memory space and communication.
- They presents a hybrid rectangular tile algorithm for linear algebra and an efficient runtime system to support it.
- They applies the system to QR and Cholesky factorization and achieves high perforamce and load balancing.

# Future work

- The largest matrix size is constrained by the memory capacity of each GPU (because static cyclic distribution is used).
  - Need to adopt a different algorithm (such as left-looking algorithm)
- Applying the algorithm to heterogeneous cluster systems by distribution the top level tiles into nodes in a 2-D cyclic distribution.

# Discussion

- Strong scaling with GPUs is not linear ?
  - CPU core are used to control GPUs?
  - Communication overhead ?
  - Why up to 12 CPU cores are used in strong scaling ?

- Comparison between dynamic scheduling?

# References

[1] E. Agullo, J. Dongarra, B. Hadri, J. Kurzak,
J. Langou, J. Langou, H. Ltaief, P. Luszczek, and A. YarKhan. PLASMA Users' Guide. Technical report, ICL, UTK, 2010.

[2] S. Tomov, R. Nath, P. Du, and J. Dongarra. MAGMA Users' Guide. Technical report, ICL, UTK, 2010.

[3] F. Song, A. YarKhan, and J. Dongarra. Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In SC'09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, pages 1–11, New York, NY, USA, 2009. ACM.