计算科学与工程中的并行编程技术

Parallel Programming Technology in Computational Science and Engineering

都志辉

清华大学计算机系

Email：duzh@tsinghua.edu.cn

Phone: 62782530

http://hpclab.cs.tsinghua.edu.cn/~duzh

GPU并行程序设计

都志辉

CPU与GPU的简要对比

- CPU 少量计算核心， multicore （几个-几十个）
- GPU 大量计算核心，manycore （几百到几千个）

Intel CPU的tick-tock模式

- “Tick”
  - 集成度提高
- “Tock”
  - 体系结构的升级

CPU的指令集

- CISC
  - complex instruction set computer
- RISC
  - reduced instruction set computer

指令级并行方法

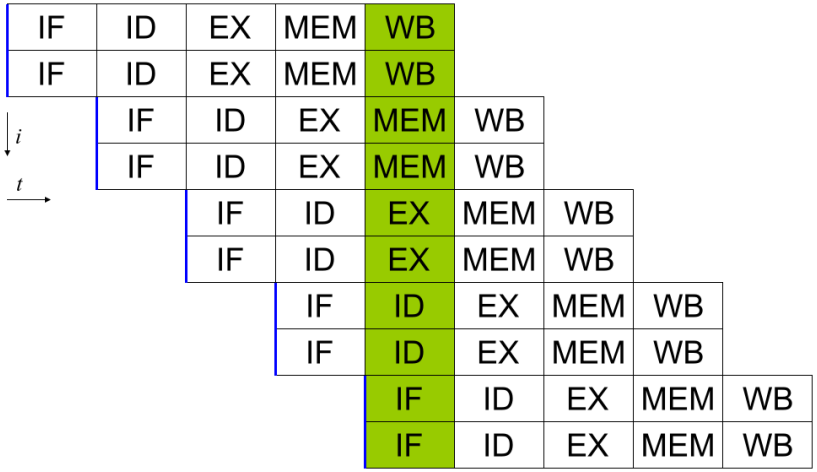
- ILP （Instruction Level Pipeline）
  - Instruction pipelining

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

(IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back

指令级并行方法

- multiple execution units are used to execute multiple instructions in parallel.
  - Superscalar



  - VLIW （Very Long Instruction Word）编译器协助

指令级并行方法

- Out-of-order execution （dynamic execution）
  - 那条指令准备好了就执行那条，避免等待
- Register renaming
  - 避免寄存器访问冲突

#	Instruction
1	R1 = M[1024]
2	R1 = R1 + 2
3	M[1032] = R1
4	R1 = M[2048]
5	R1 = R1 + 4
6	M[2056] = R1

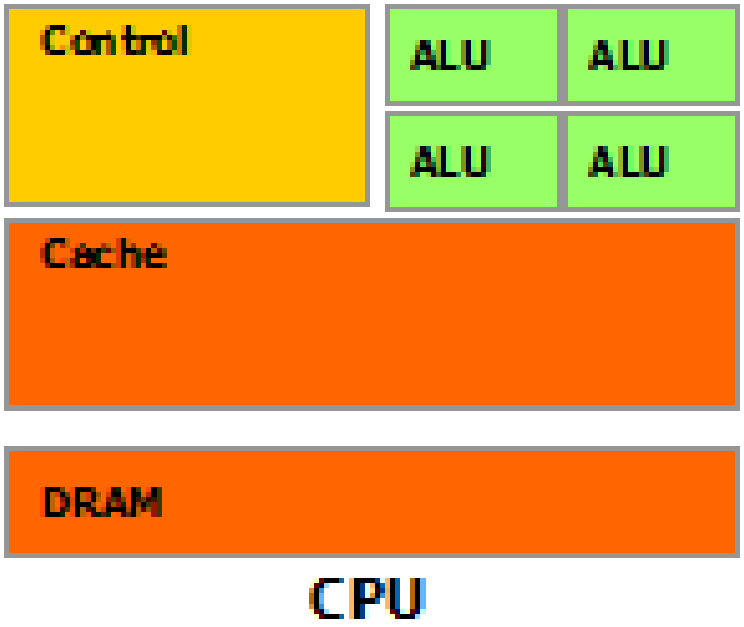
#	Instruction
1	R1 = M[1024]
2	R1 = R1 + 2
3	M[1032] = R1

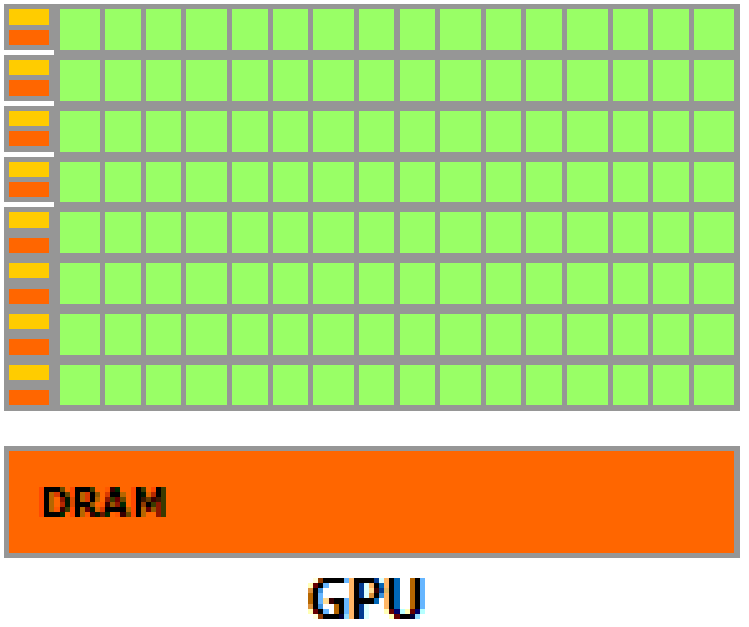
#	Instruction
4	R2 = M[2048]
5	R2 = R2 + 4
6	M[2056] = R2

指令级并行方法

- Speculative execution + Branch prediction
  - 避免等待

GPU与CPU利用硬件资源的不同

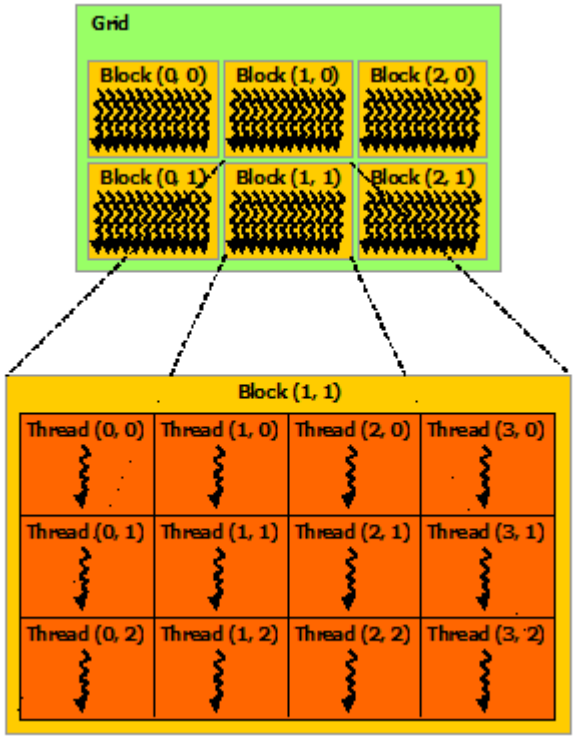


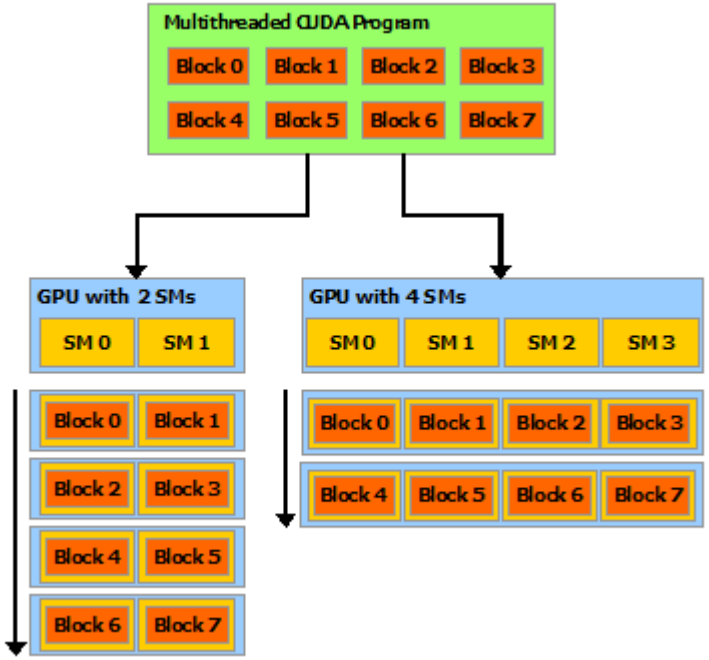


CUDA

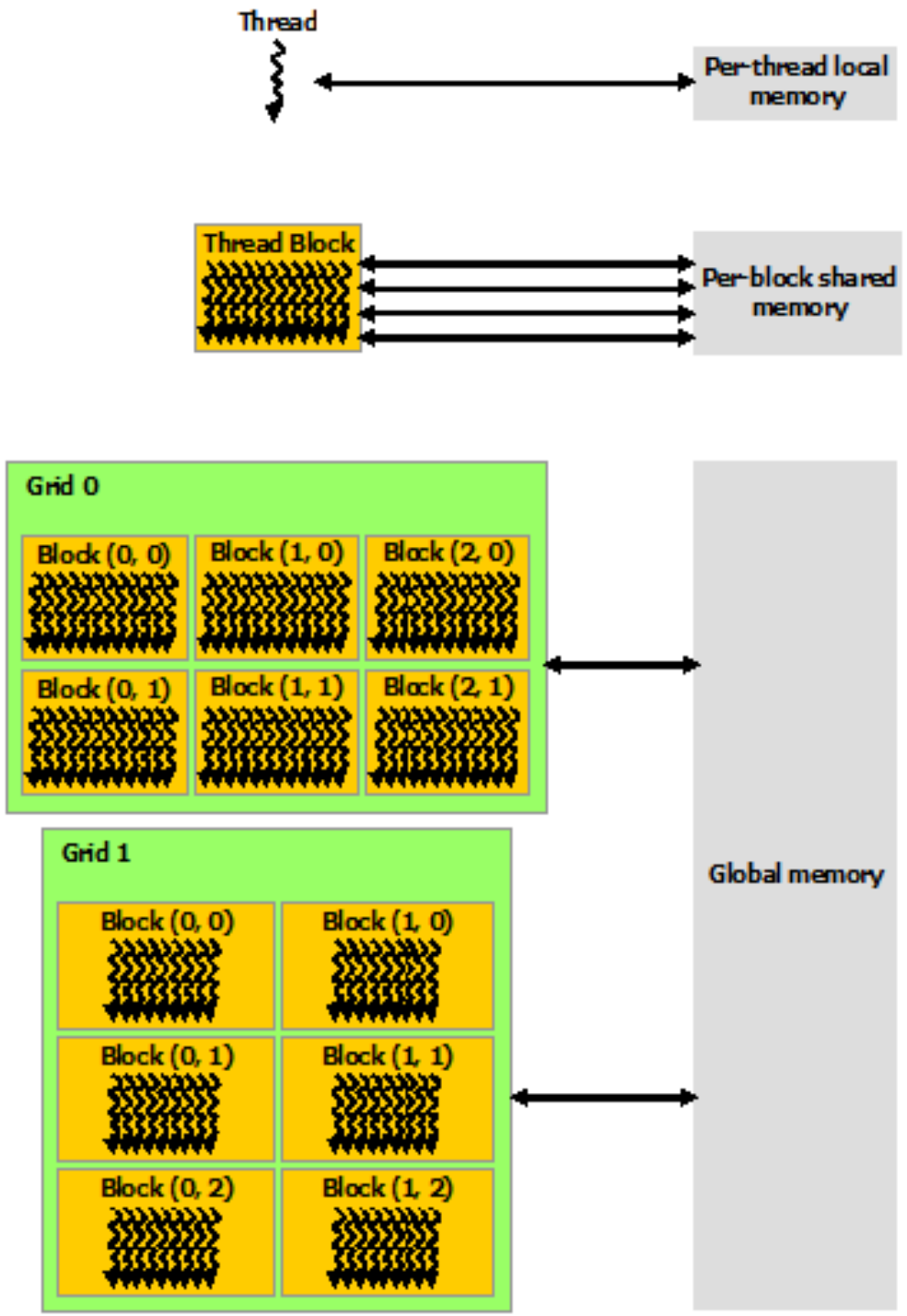
- NVIDIA GPU的并行编程环境与模型
  - Compute Unified Device Architecture
- OpenCL
  - an open standard maintained by the non-profit technology consortium Khronos Group

CUDA线程组织

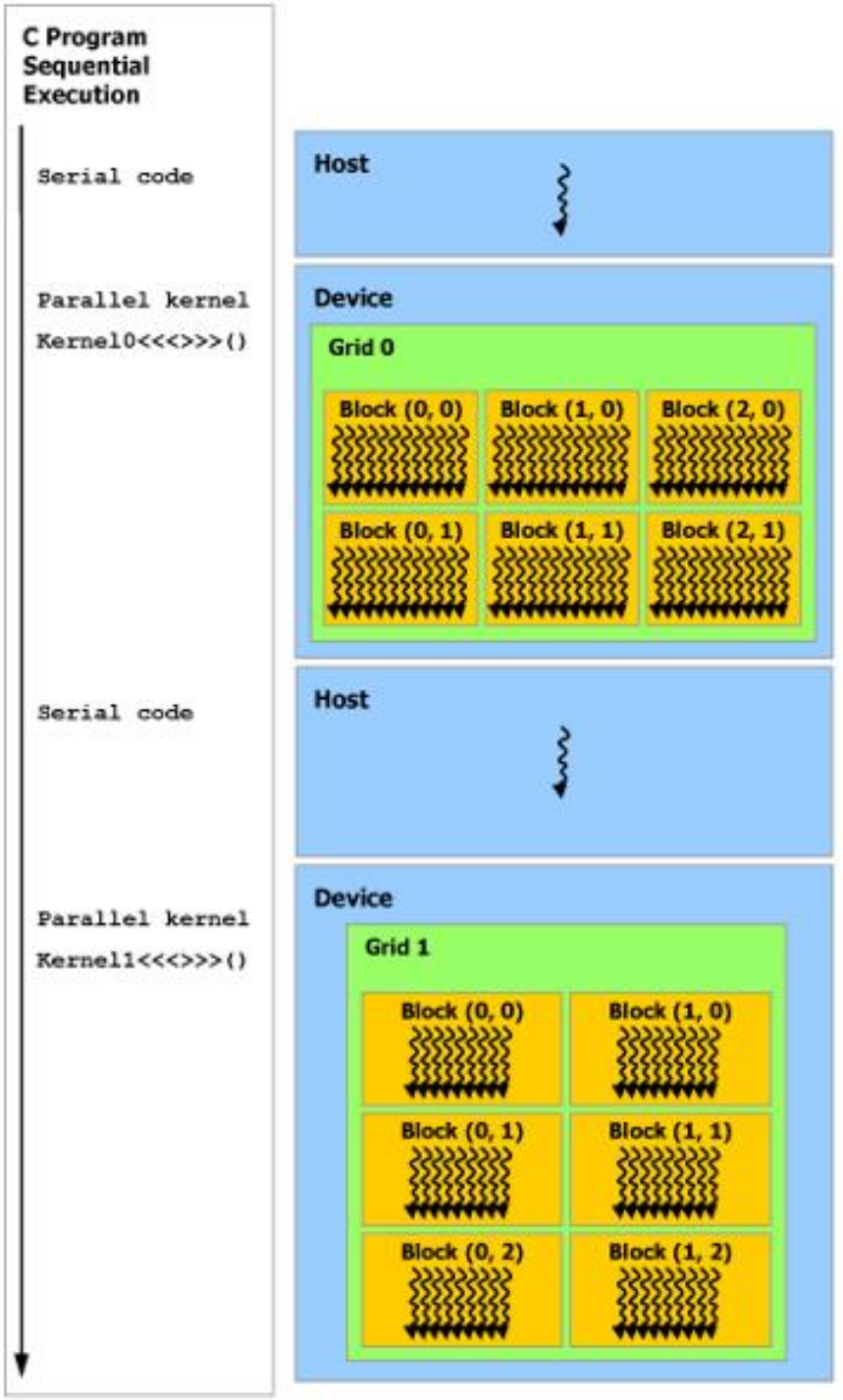

Grid
Block (0, 0)
Block (1, 0)
Block (2, 0)
Block (0, 1)
Block (1, 1)
Block (2, 1)
Block (1, 1)
Thread (0, 0)
Thread (1, 0)
Thread (2, 0)
Thread (3, 0)
Thread (0, 1)
Thread (1, 1)
Thread (2, 1)
Thread (3, 1)
Thread (0, 2)
Thread (1, 2)
Thread (2, 2)
Thread (3, 2)


Multithreaded CUDA Program
Block 0
Block 1
Block 2
Block 3
Block 4
Block 5
Block 6
Block 7
GPU with 2 SMs
SM 0
SM 1
GPU with 4 SMs
SM 0
SM 1
SM 2
SM 3

不同线程共享的存储结构



执行过程


C Program
Sequential
Execution
Serial code
Parallel kernel
Kernel0<<<>>>()
Serial code
Parallel kernel
Kernel1<<<>>>()
Host
Device
Grid 0
Block (0, 0)
Block (1, 0)
Block (2, 0)
Block (0, 1)
Block (1, 1)
Block (2, 1)
Host
Device
Grid 1
Block (0, 0)
Block (1, 0)
Block (0, 1)
Block (1, 1)
Block (0, 2)
Block (1, 2)

例子

```
// Kernel
definition __global__ void MatAdd(float A[N][N], float B[N][N], fl
oat C[N][N])
{ int i = threadIdx.x;
int j = threadIdx.y;
C[i][j] = A[i][j] + B[i][j];
}
int main() { ... // Kernel invocation with one block of N * N * 1
threads
int numBlocks = 1;
dim3 threadsPerBlock(N, N);
MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
...
}
```

多个BLOCK

```
// Kernel
definition __global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{ int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.y;
if (i < N && j < N)
C[i][j] = A[i][j] + B[i][j];
}
int main() { ... // Kernel invocation
dim3 threadsPerBlock(16, 16);
dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
...
}
```

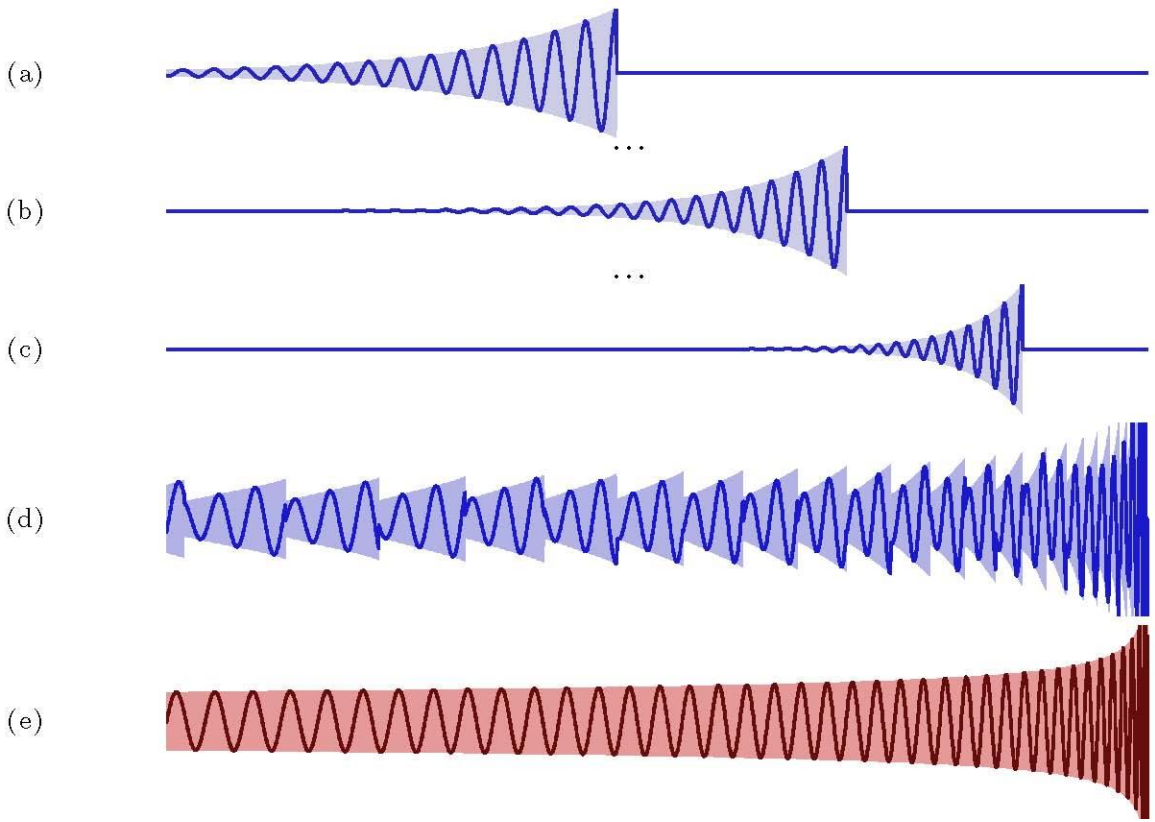
GPU应用

在引力波中的应用

Achievements

- GW data processing (120X speedup) on GPU
  - Compared with sequential CPU program on desktop computer
  - Real time data processing on a PC
- GW source modeling (5Xspeedup) on GPU cluster
  - Compared with the best CPU parallel MPI program

Fancy Idea: SPIIR(Summed Parallel Infinite Impulse Response) Filtering Method

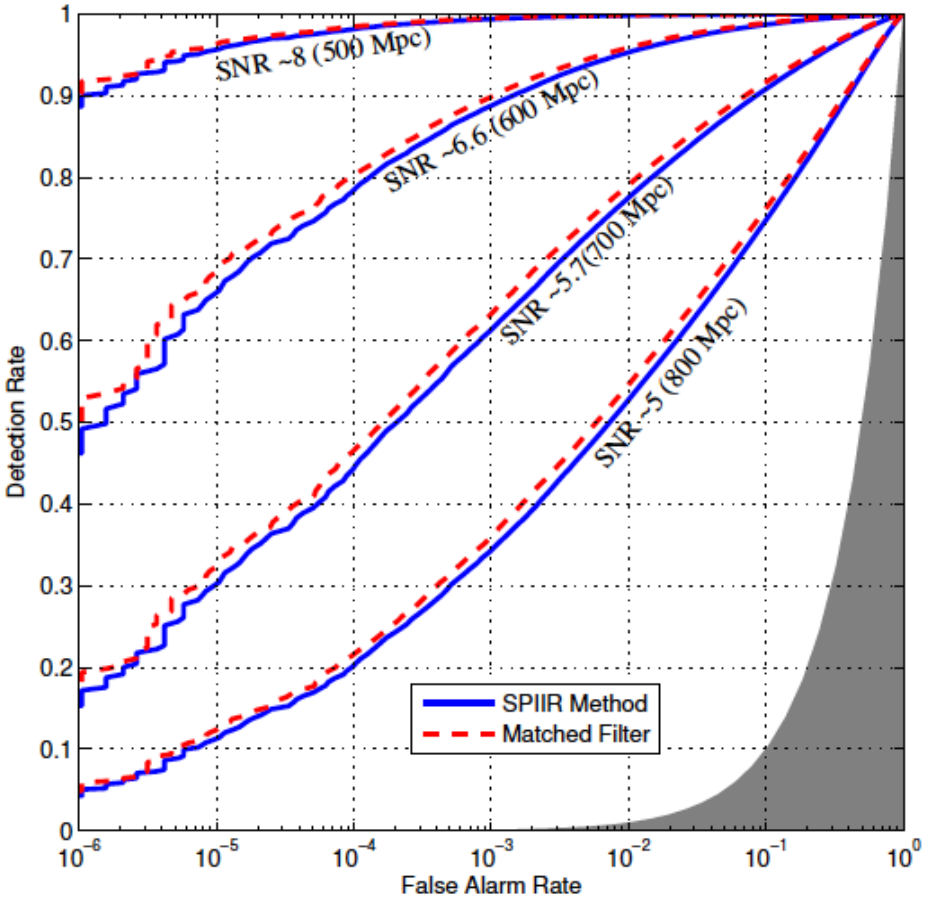


Approximate waveform

Predicted CBC waveform

Applying each filter to data is equivalent to correlating the data with a damped-sinusoid.

Detection Efficiency Comparable to Optimal Matched Filtering



SPIIR vs Optimal Matched Filtering for Gaussian Noise for 1.4+1.4 Msun

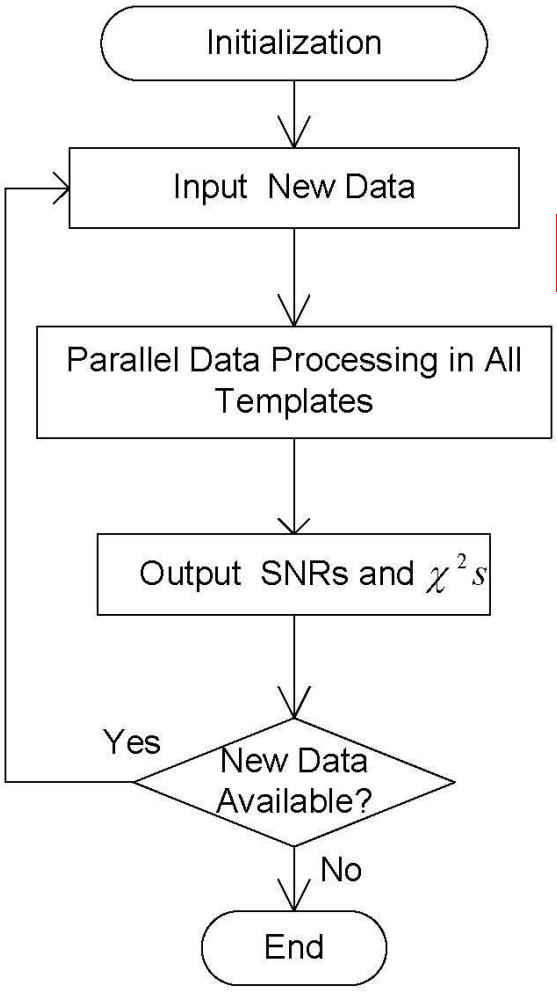
Hooper, S. 2014 PhD Thesis, Hooper S. 2012 PRD

Computing Hardware

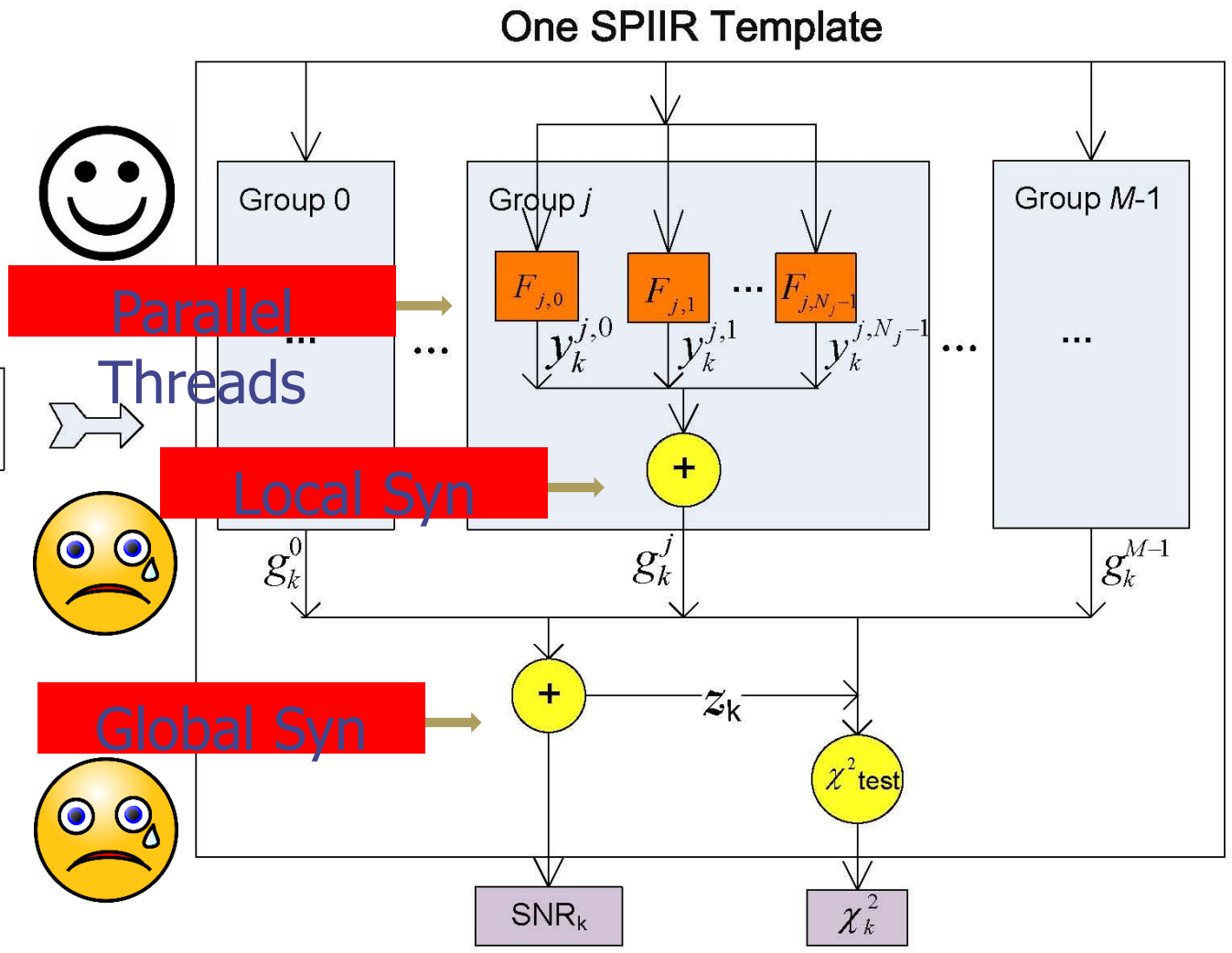
- 16 SMM (128 cores)
- 2048 cores
- 4612Gflops
- L2 cache 2M
- 4GB memory

SPIIR Method Processing Loop

(Summed Parallel Infinite Impulse Response)



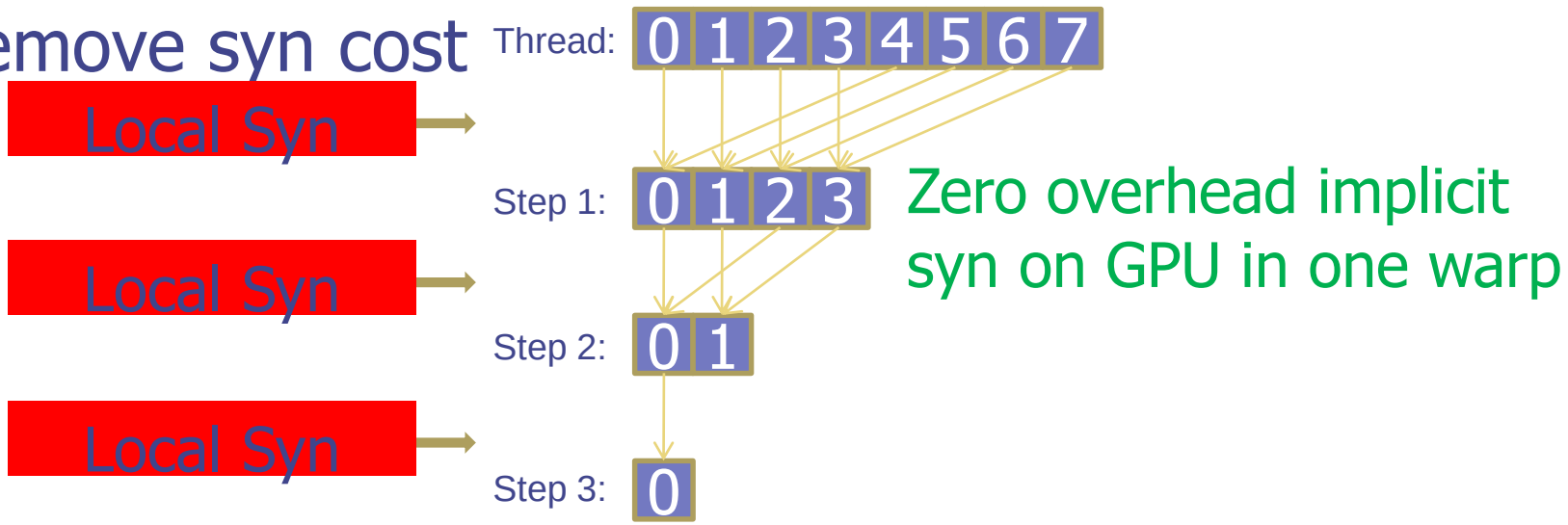
(a)



(b)

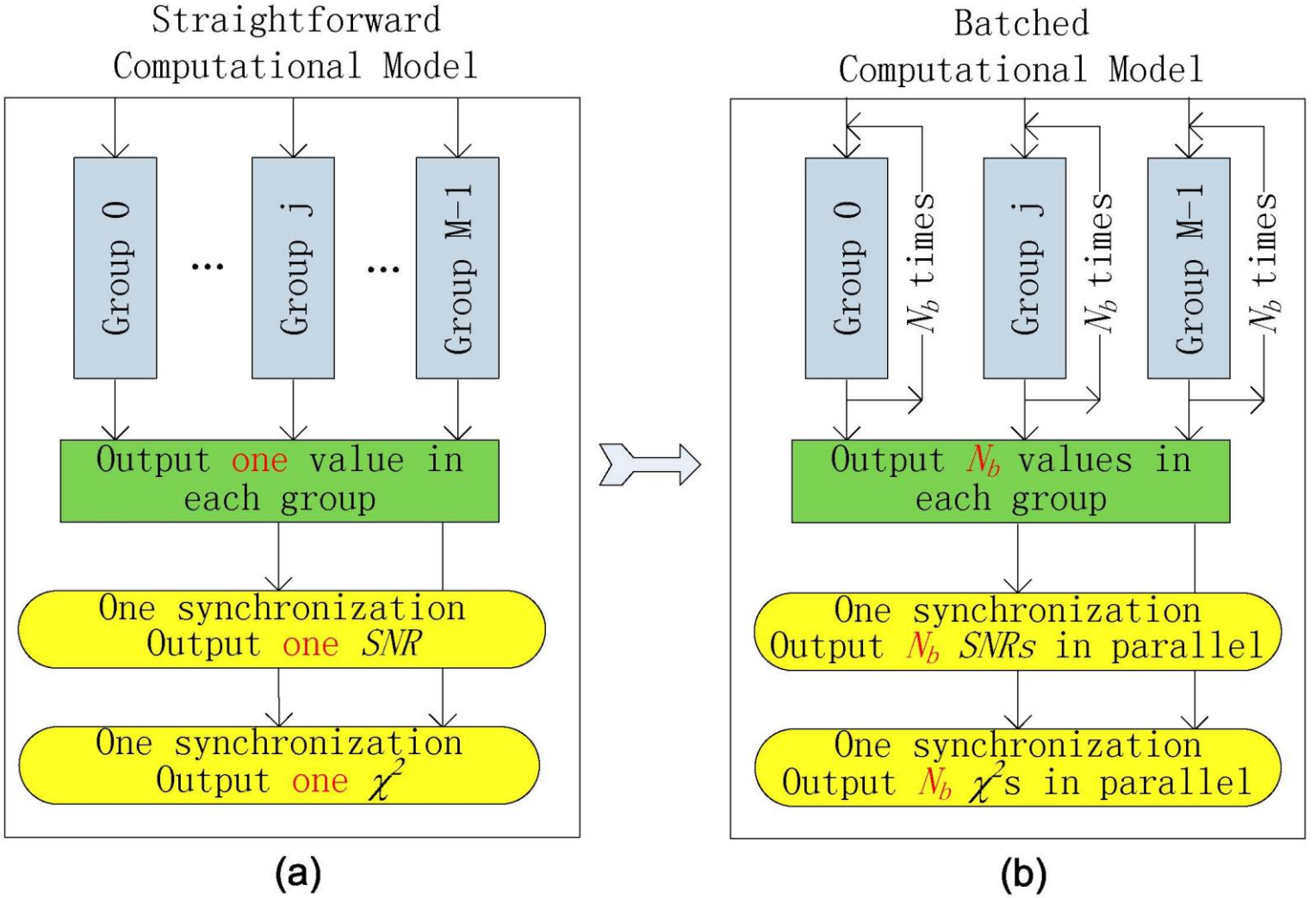
Attack local syn problem
Summation Methods

- Sequential Methods
  - N number, one thread, N-1 add steps
- Parallel sum reduction (improve parallelism,2x speedup)
  - Reduce add steps: N number, N threads, $\log_2 N$ add steps
  - Remove syn cost

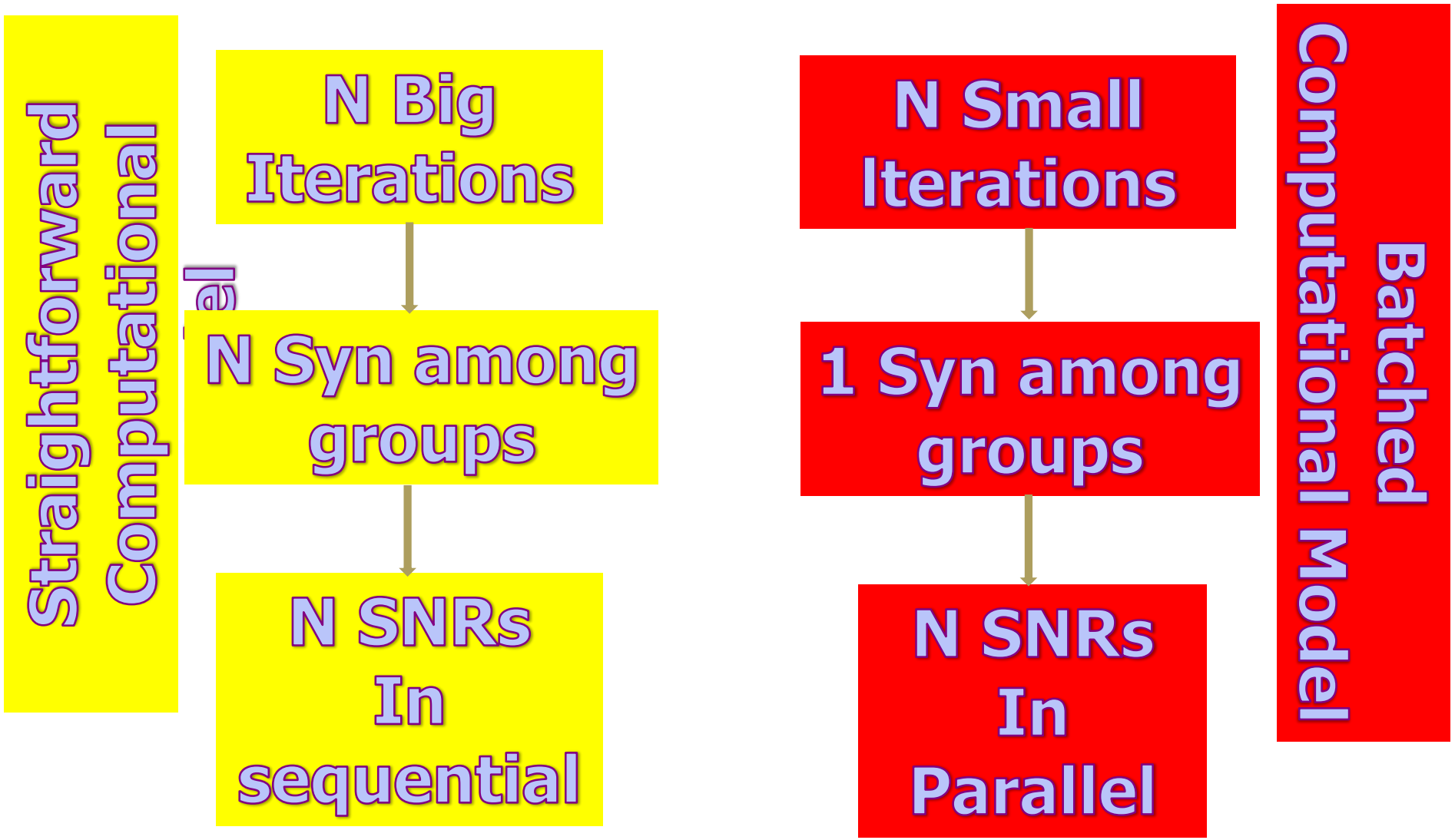


Attack global syn problem

Batched Computational Model to Reduce #(Global Syn)

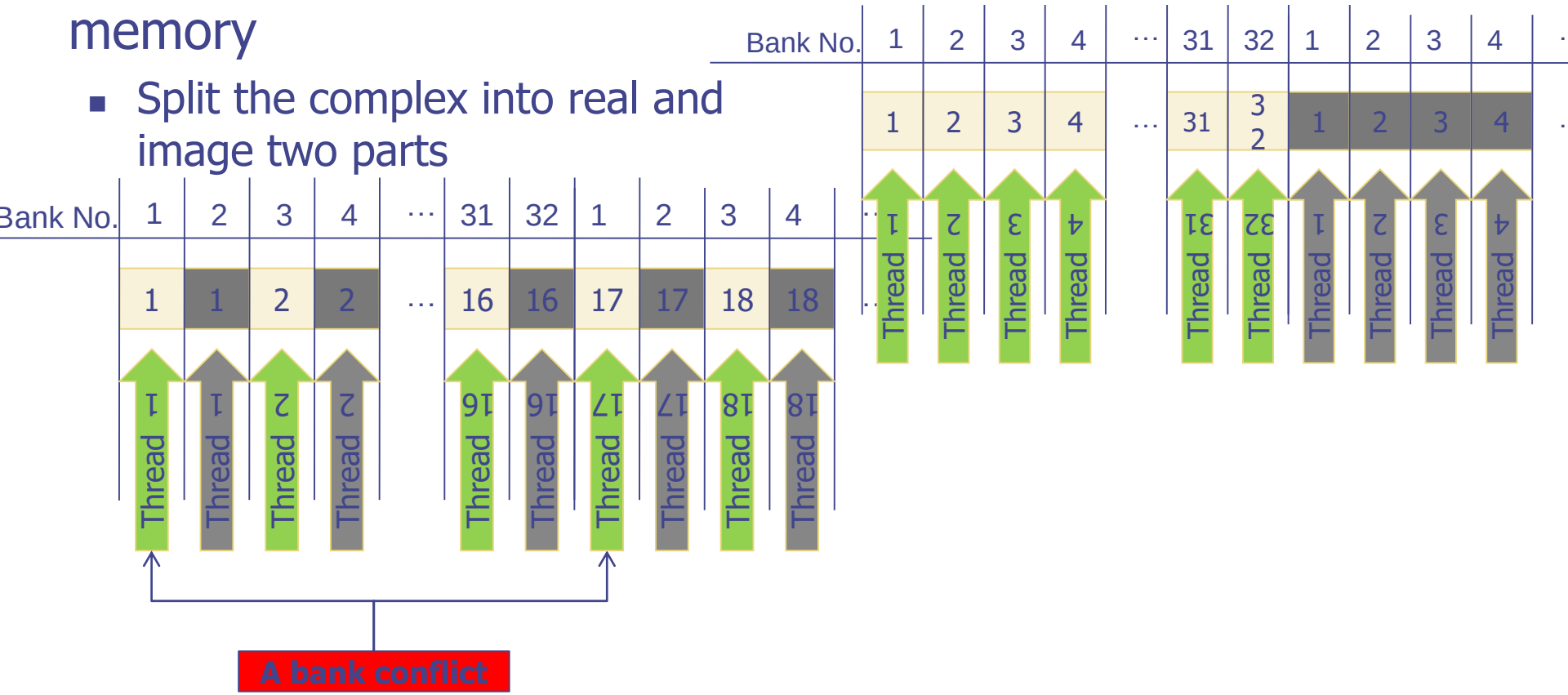


Comparison

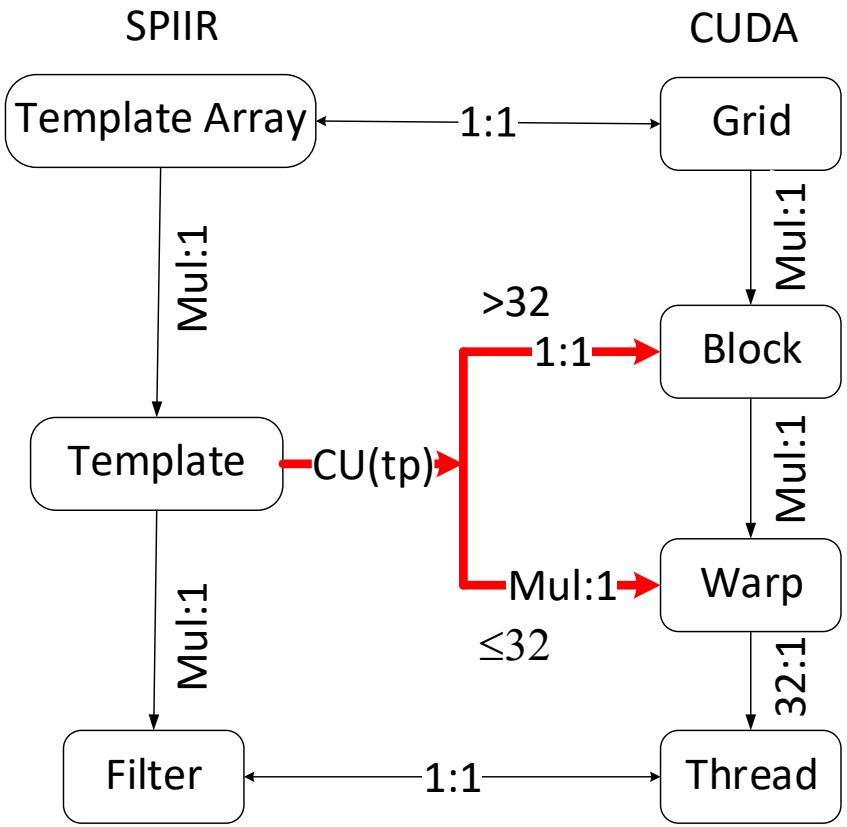

Straightforward Computational
N Big Iterations
N Syn among groups
N SNRs In sequential
N Small Iterations
1 Syn among groups
N SNRs In Parallel
Batched Computational Model

Memory Optimization

- Texture Cache
  - Take advantage of high access performance of texture cache and the locality of our application
  - 1.43x performance gain
- Rearrange the data structure to avoid bank conflicts in shared memory
  - Split the complex into real and image two parts



Adaptive template mapping to avoid idle threads



$$\mathrm{CU(tp)} = \begin{cases} 2^t, 2^{t-1} < size(tp) \leq 2^t \leq 32 \\ 32t, 32 < 32(t-1) \leq size(tp) \leq 32t \end{cases}$$

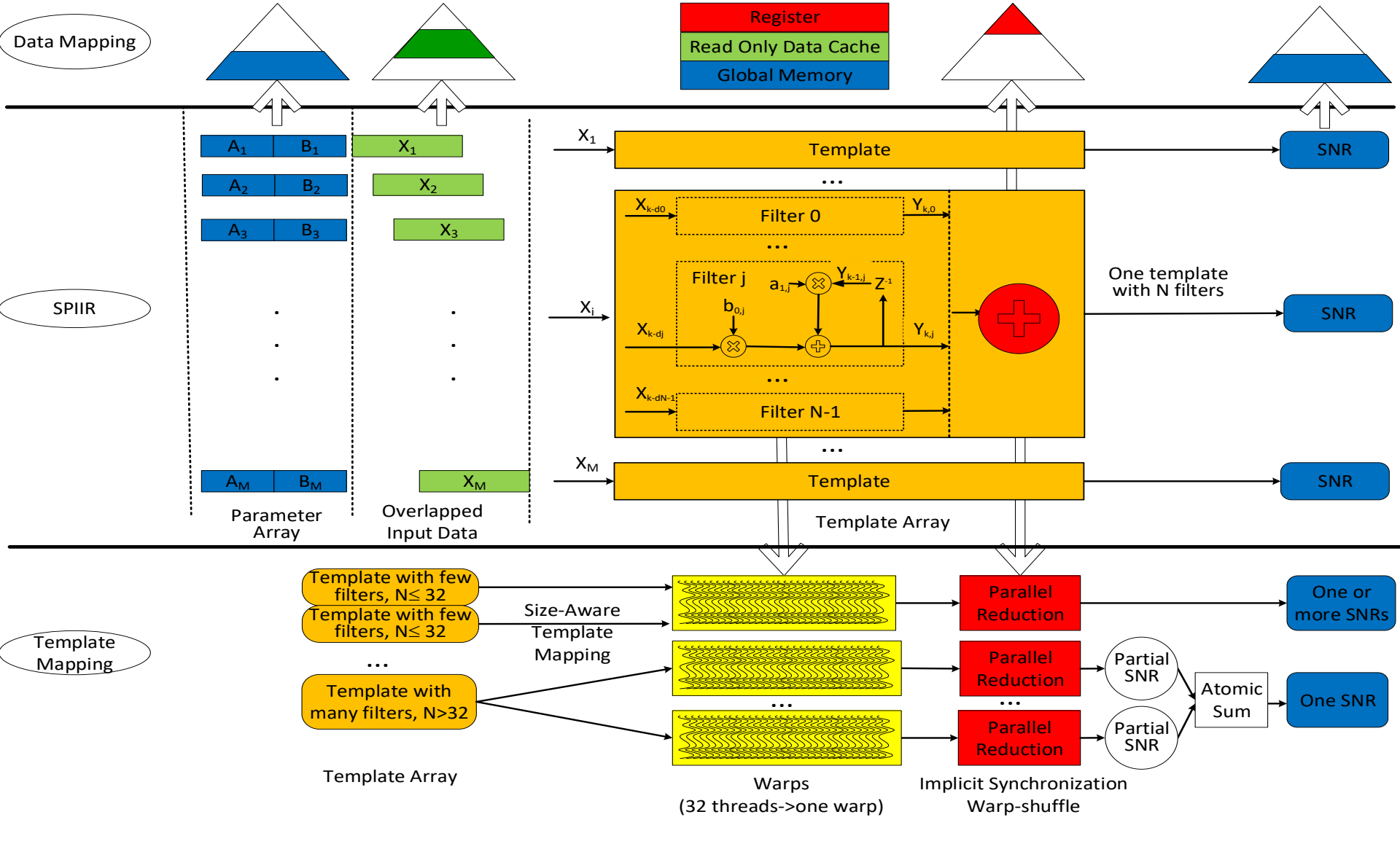
Improve the hardware utilization

- Adjust register and shared memory usage to improve occupancy (active threads)

New Hardware Features

- Read only data cache
  - Take advantage of data locality
- Atomic operation
  - Cost less
- Warp-shuffle
  - Share data among one warp

Optimized Implementation on Maxwell GPU



Experimental Results

Hardware	CPU	Intel *Core* i7-3770 3.40 GHz
	GPU	NVIDIA *GeForce* GTX 980
	Host Memory	8 GB DDR3
Software	Operating System	Fedora 20 64-bit
	CUDA Version	6.5
	Host Compiler	gcc 4.8.3

Method	Overall speedup
Straightforward	7.0
Batched+PSR-IS	25
+Texture memory	42
+Avoid bank conflicts	48
+Tuning resource usage	58

Application features based methods

General methods

Half of the speedup from the hardware based optimization, another half of the speedup from the application based optimization

Speedup Employ New Hardware Features

Table 2. speedup ratio of different GPU kernels compared to CPU counterpart

Template Size	4	8	16	32	64	128	256	512
GPUv2	63.10	70.62	61.22	55.98	109.76	124.41	125.58	123.97
GPU v1	20.97	19.59	29.41	42.51	54.58	64.42	72.74	78.31
GPU prev	6.13	10.79	19.61	37.50	42.64	48.72	39.51	29.78

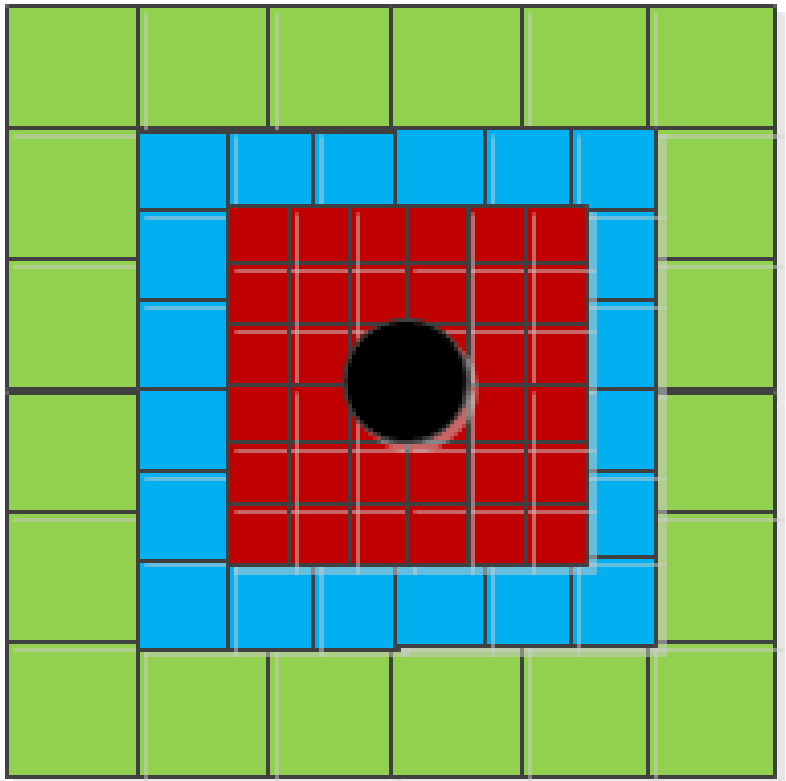
Numerical precision

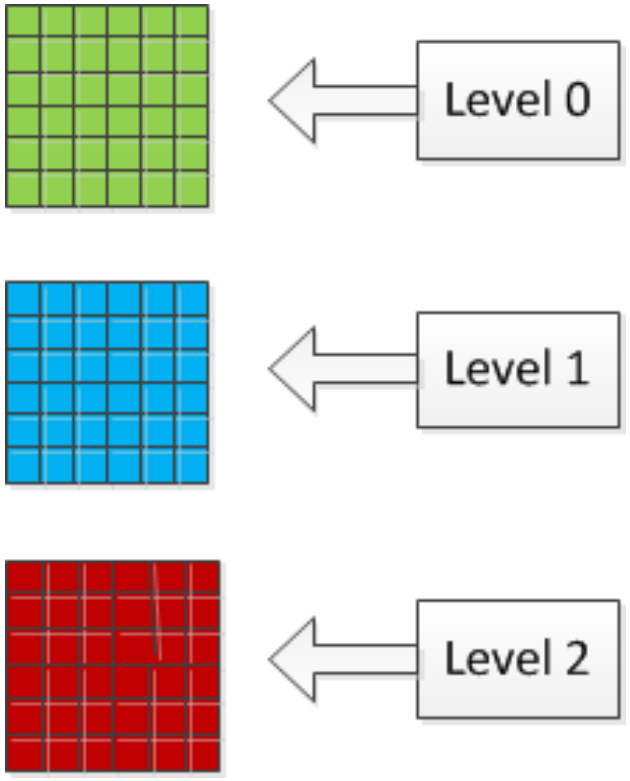
- GPU_FLOAT gave roughly 0.002% average fractional errors on the SNR values and 0.0002% on the χ^2 values
- 99% of the SNR and the χ^2 values had less than 0.007% and 0.001% errors respectively.

Accelerate AMSS-NCKU on GPU Cluster

AMR in BBH simulation

- AMR (Adaptive Mesh Refinement)
  - Cover interesting area with refined mesh
  - Number of grid points of each level is similar

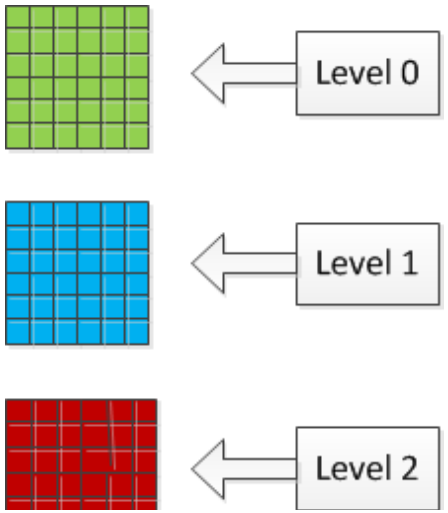


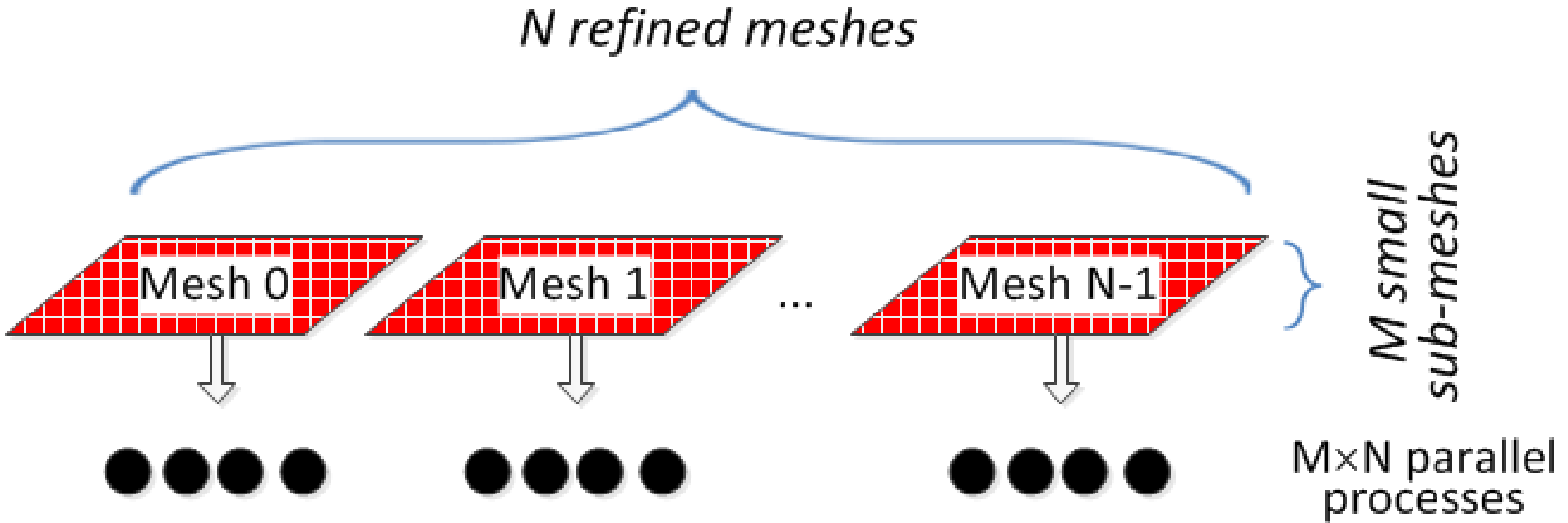


Fancy idea: MPM Approaches

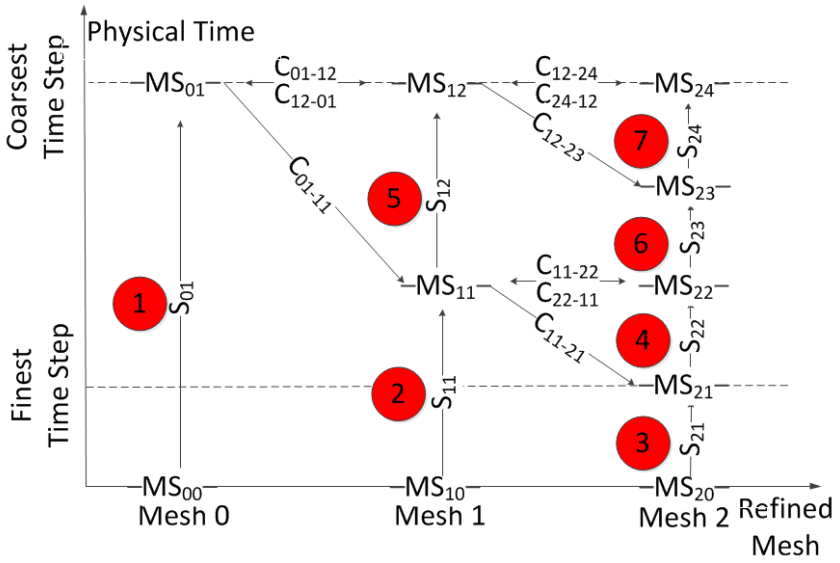
Mesh based Parallel AMR Method

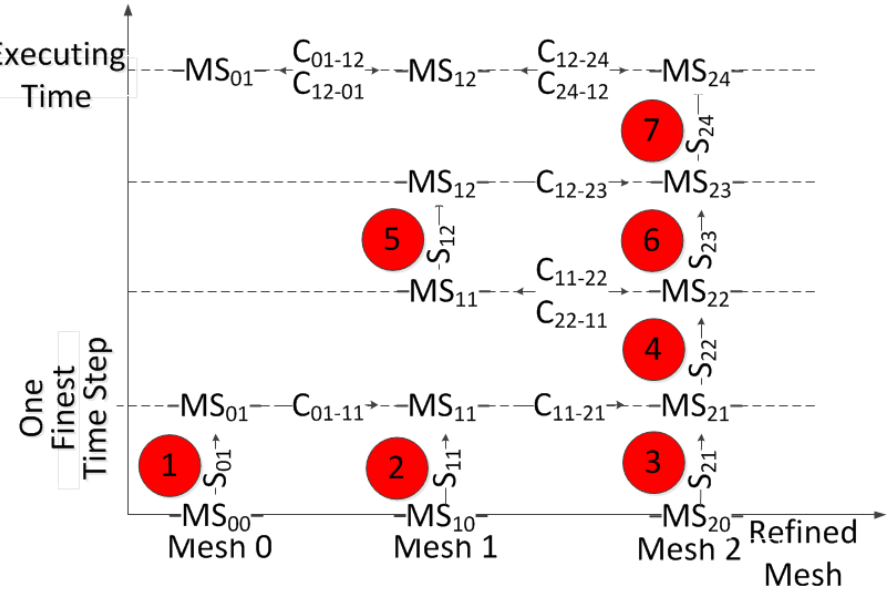
- Exploring two-level parallelism
  - Mesh level (new)
  - Sub-mesh level (existing)





Meshes can be parallelized!





Simulation step 1 {1} Simulation step 5{5}
Simulation step 2 {2} Simulation step 6{6}
Simulation step 3 {3} Simulation step 7{7}
Simulation step 4 {4}

Simulation step 1 {1,2,3}
Simulation step 2 {4}
Simulation step 3 {5,6}
Simulation step 4 {7}

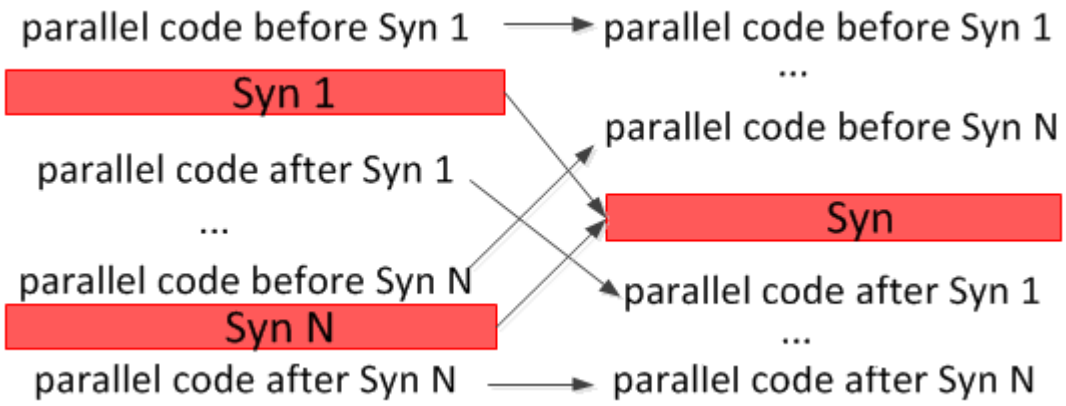
Hardware

Implementation
Two-level Parallel Program

MPI (among nodes)+CUDA (in each node)

GPU Implementation
RHS calculation

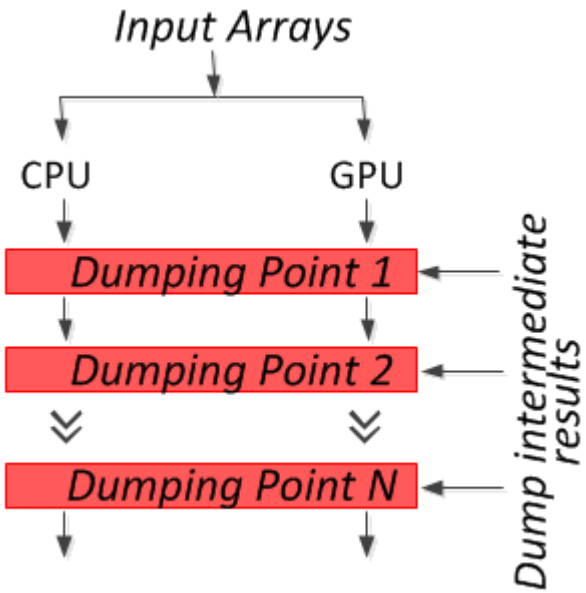
- Reducing the number of data copy
  - Reuse the array on GPU between different kernels
  - Only copy necessary data
- Reducing the number of synchronization

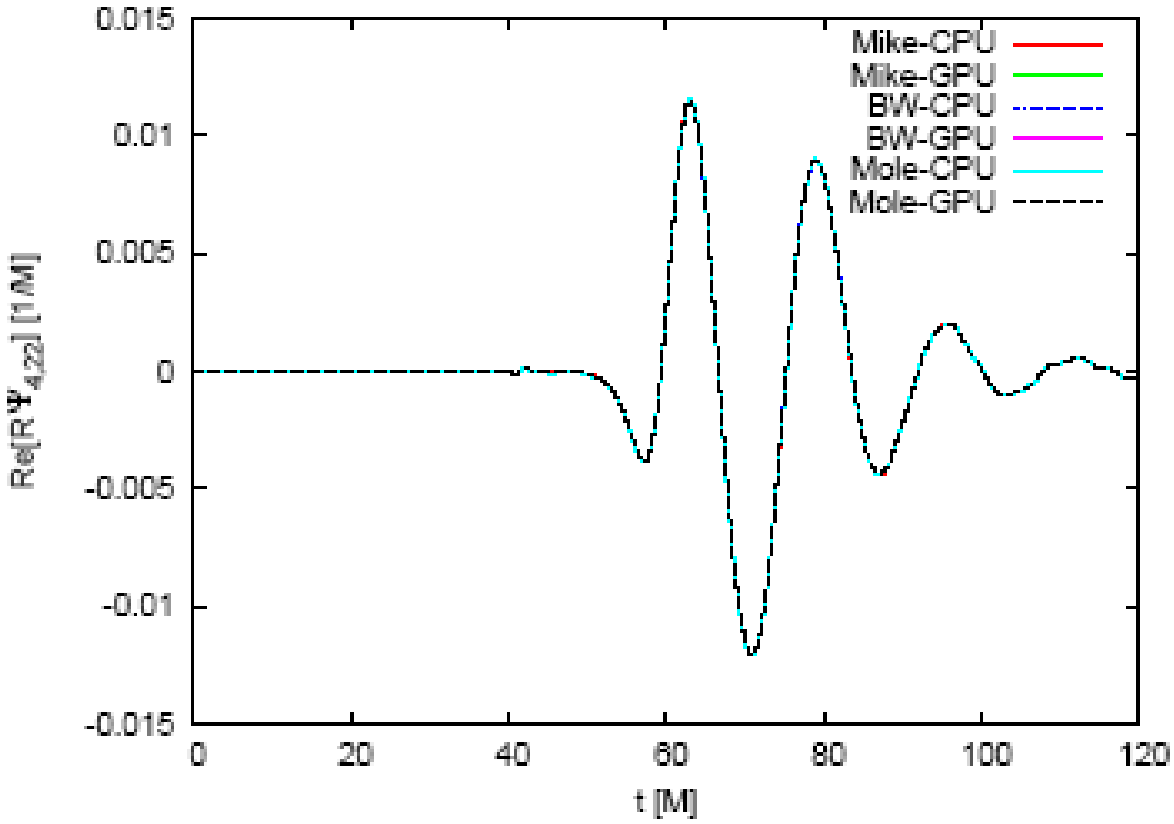


- Coalesced memory access and shared memory

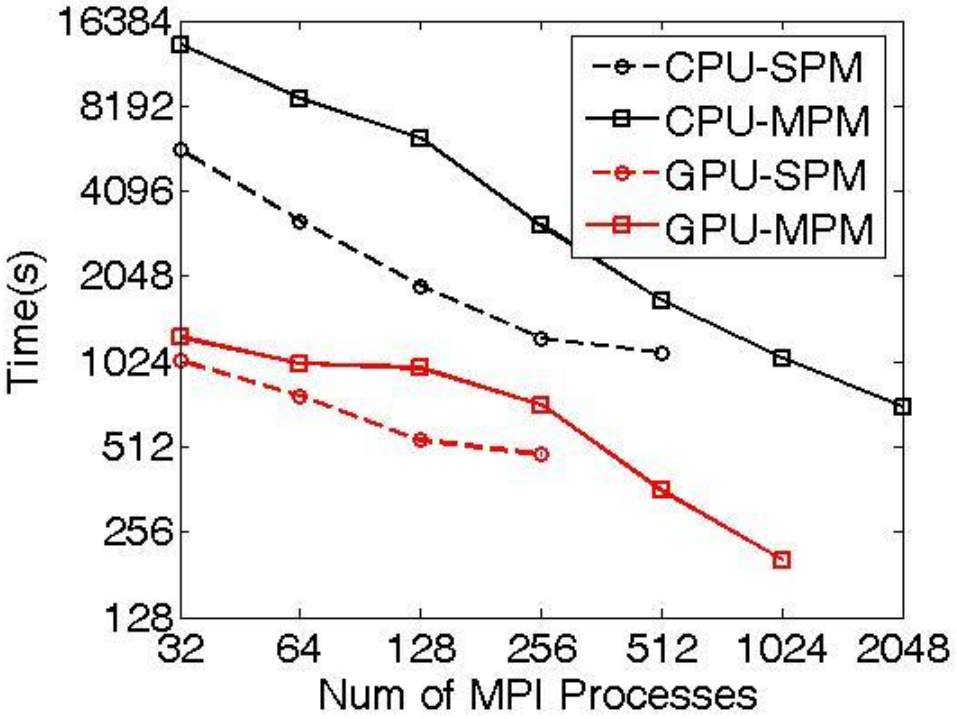
Correctness evaluation

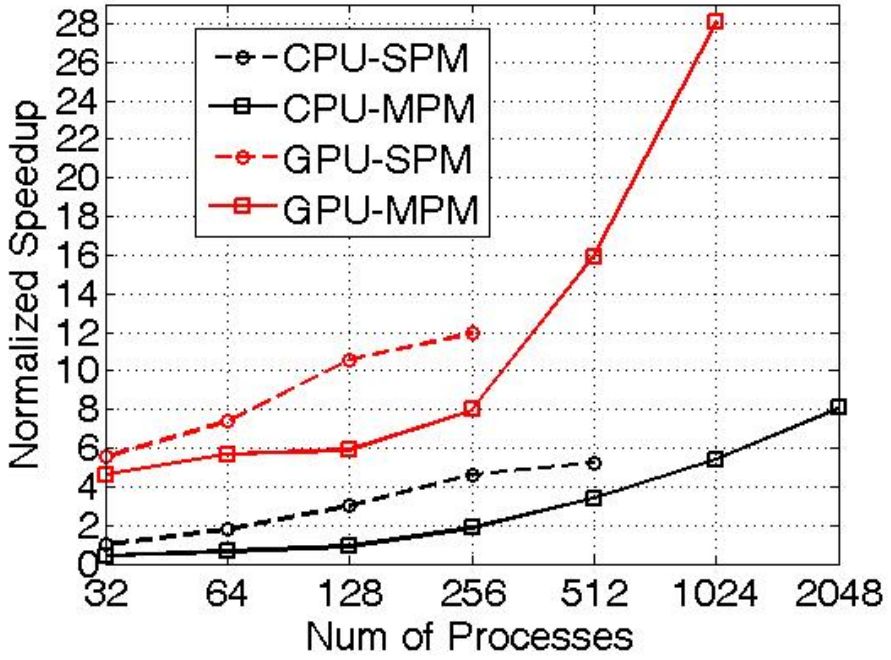
- Synthetic input (less than 10^{-14})
- Typical Input





Strong Scaling on BW

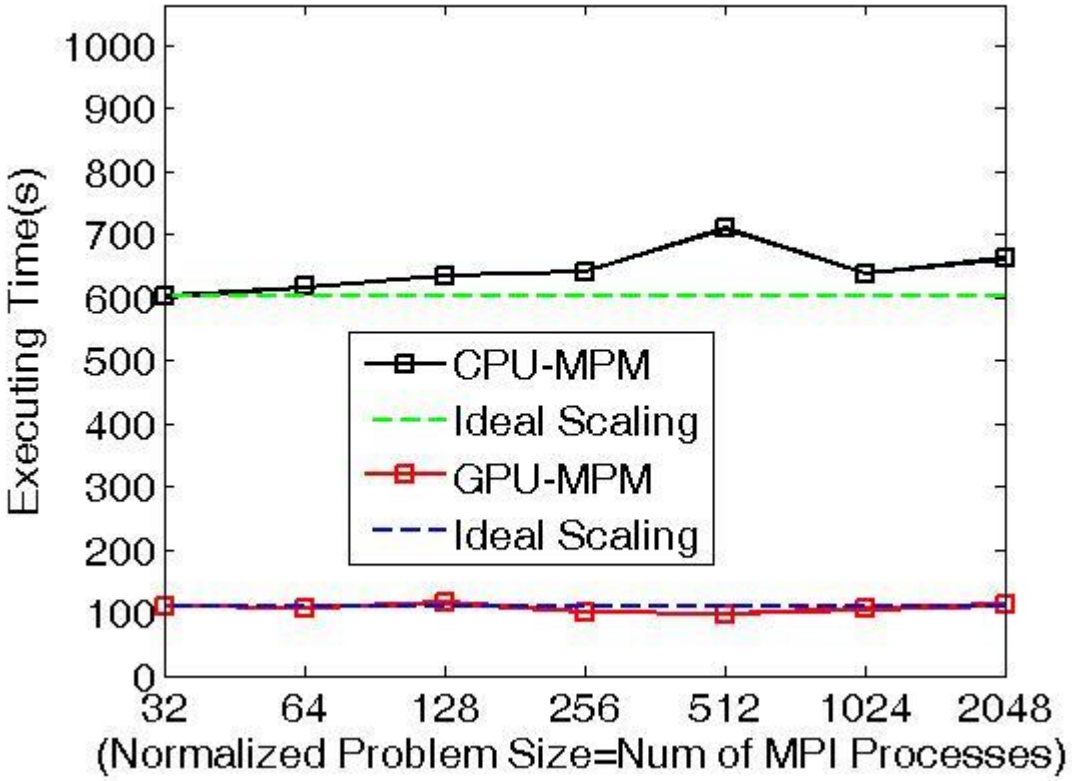




128*128*64, 8 refined mesh levels, 5 physical steps

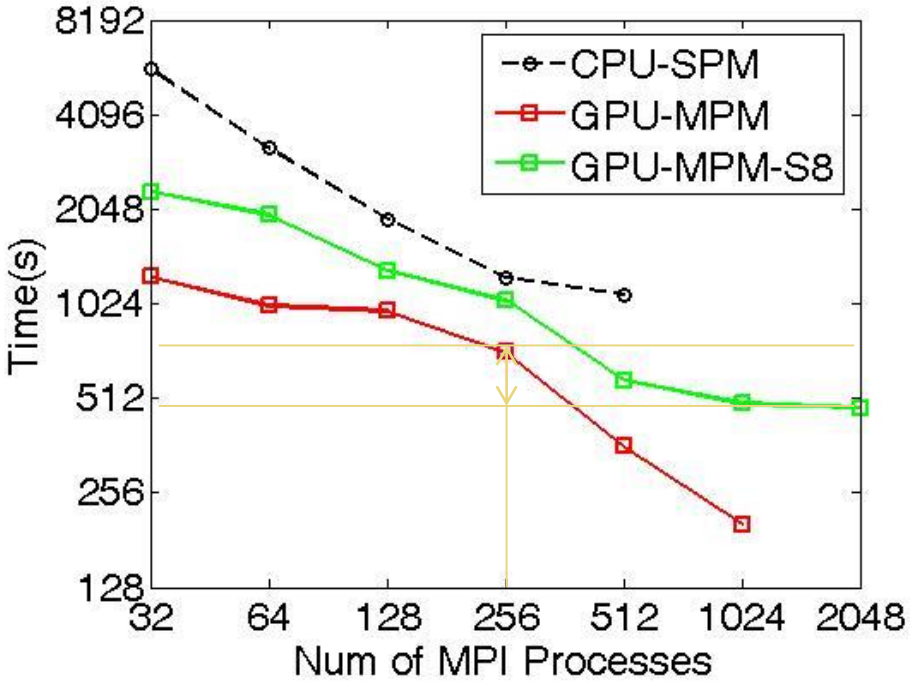
28X speedup, half from GPU half from the new algorithm

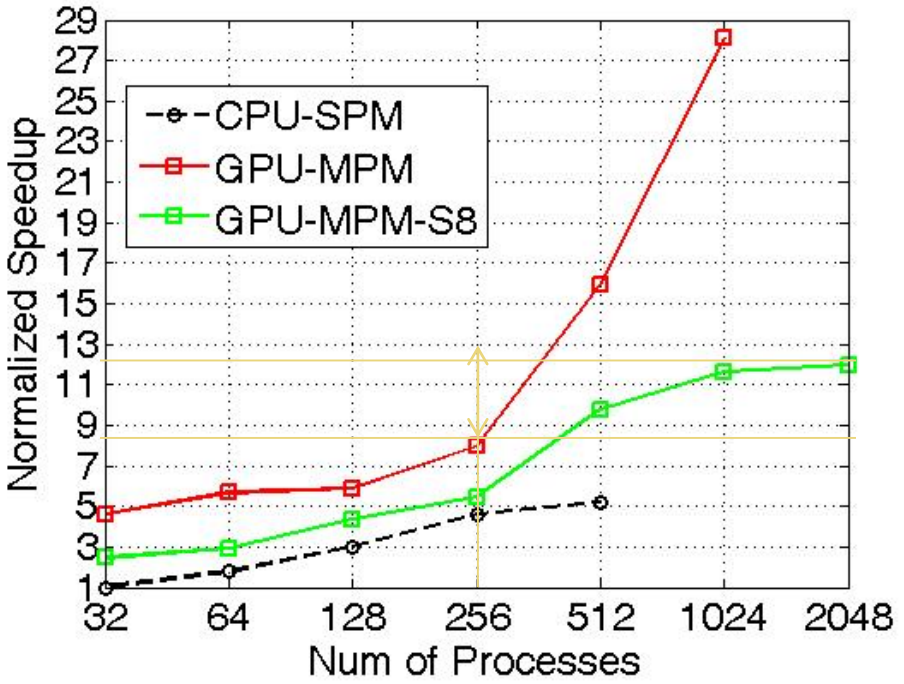
Weak Scaling on BW



320*320*160, 2 refined mesh levels, 5 physical time steps

Sharing GPU to improve performance





2048 MPI processes share GPU can achieve 12X speedup, but 256 MPI processes use 256 GPU can only achieve 8Xspeedup

Another very critical problem

Mesh partition algorithm can significantly affect the performance

Mesh Partition Algorithm

as even/cubic as possible

input : $Mesh, Proc, Min$
output: $SubMesh$

begin
  $tmp[0] = tmp[1] = tmp[2] = 1;$
  while $(tmp[0] \times tmp[1] \times tmp[2] < Proc)$ **do**
    select the dimension d which can get maximum value of $\frac{Mesh[d]}{tmp[d]}$;
    if $(\frac{Mesh[d]}{(tmp[d]+1)} < Min)$ break;
    $tmp[d]++;$
  end
  for $i \leftarrow 0$ **to** 2 **do**
    $SubMesh[i] = ceiling(\frac{Mesh[i]}{tmp[i]})$
  end
end

Mesh: 3D array to be partitioned.
Proc: # processors
Min: minimal sub-mesh size
SubMesh: partition result

HTK包训练过程的GPU加速

Outline

- Background
- Our Optimization Methods on GPU
- Experimental Results
- Conclusion

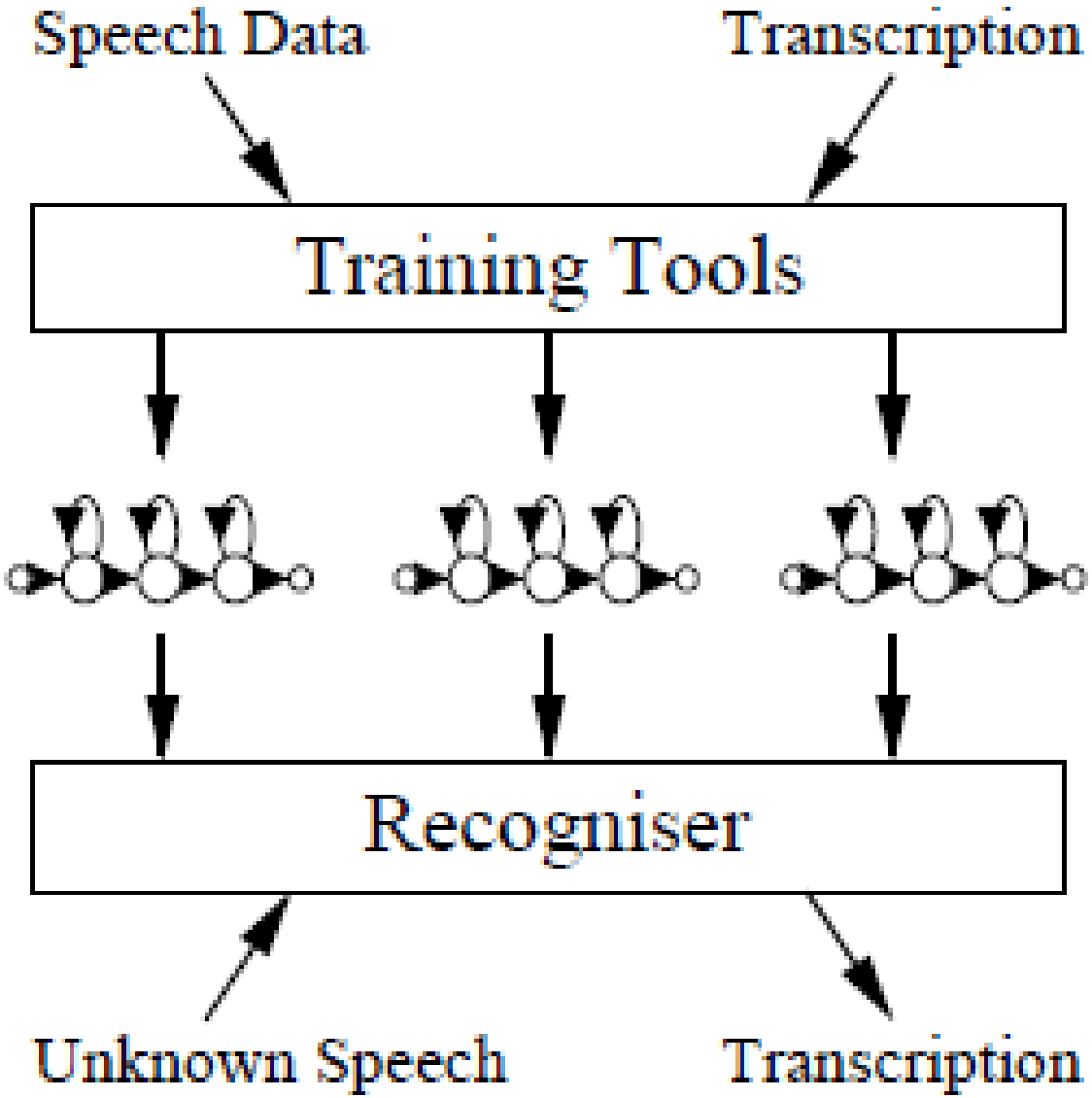
Motivation and Challenge

- Motivation
  - Can some speech recognition tools be provided with high-performance and low cost so it will be as popular as PC
- Challenge
  - high quality/high recognition ratio
  - time consuming (computational intensive)

OUR WORK

Improve the performance of an open source speech recognition software package HTK

Brief introduction of HTK



Analysis of HTK Hidden Markov Model Toolkit

- Training part takes much more time
- The most time consuming section of training can be calculated independently (parallel)
  - 80-20 rule.
  - We focus on improving the performance of the most time consuming section.

Our methods

- Identify the most time consuming codes
- Port and optimize those codes on GPU

The most time consuming codes-

regular and simple calculation

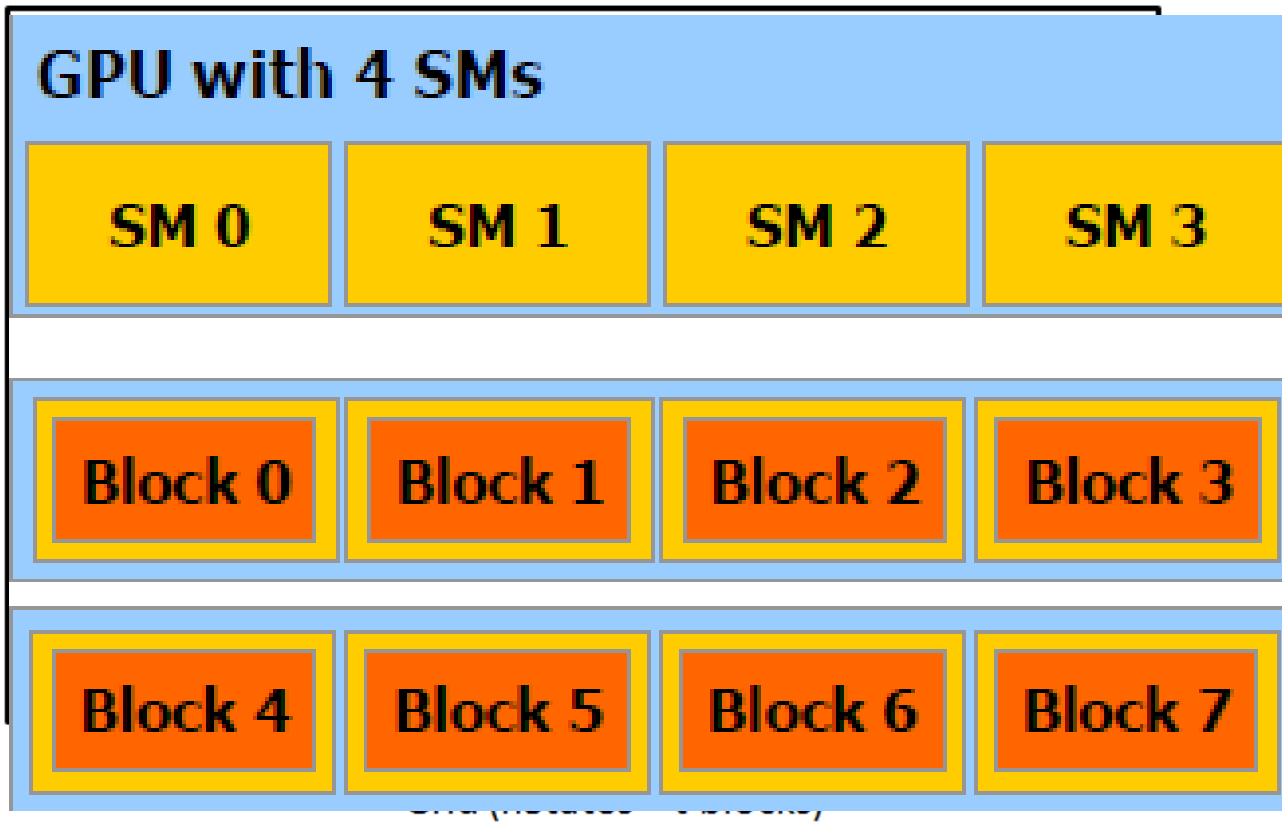
- Frequently calculate the output distribution

```
for i = 0 to t-1 {://observation loop 1
    obs =O_i   ;
    for j = 0 to nStates-1{://state loop 1
        x = 0;
        for m = 0 to M-1 { //loop 3
            weight =C_jm  ;
            for k = 0 to dim-1 {
                xmm = obs[k] – mean[k];
                mixp += (xmm * xmm * var[k]);
            }
            mixp *= 0.5;
            x = logadd(x,weight + mixp);
            ppScores[i][j] = x;
        }
    }
}
```

How to build independent blocks

- Unrolling loop 2&1

```
for i = 0 to t-1:
    obs = O_i ;
    for j = 0 to nStates-1:
        x = computed;
        ppScores[i][j] = x;
```



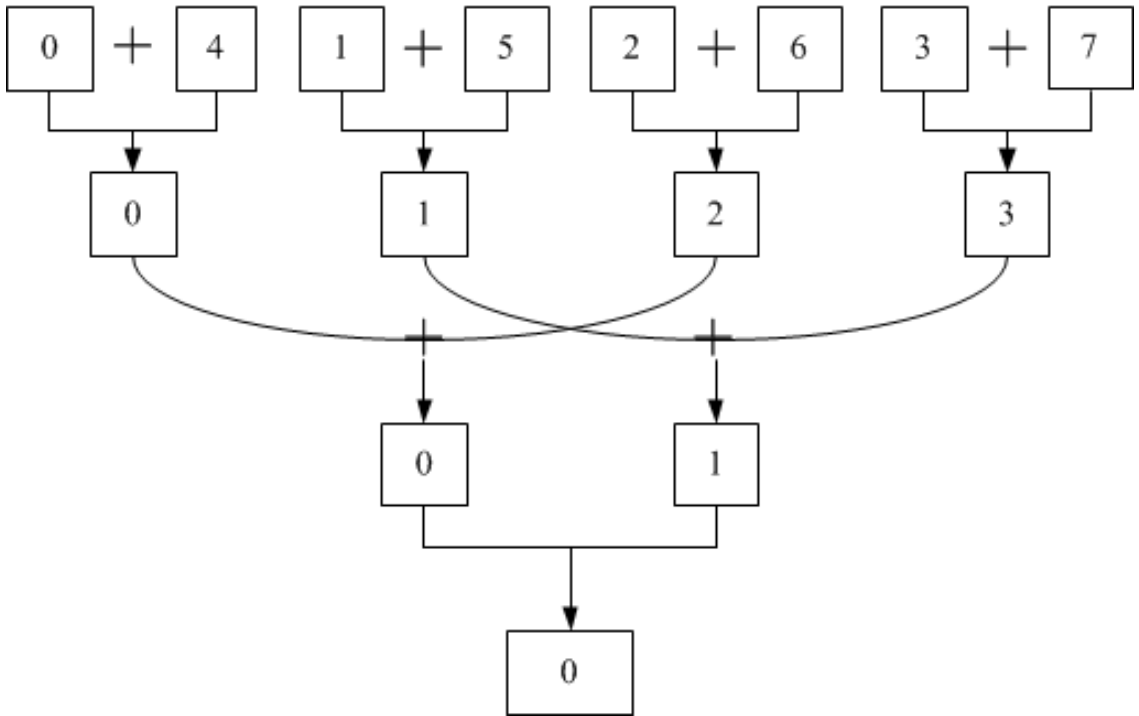
Sum Reduction:

improve parallel degree

calculation reduced from N to logN

```
for k = 0 to dim-1
xmm = obs[k] – mean[k];
mixp += (xmm * xmm * var[k])
```

```
increment = N/2;
while(increment > 0)
   for i = 0 to increment - 1
      xmm[i] += xmm[i+increment
];
   increment /= 2;
```



Memory optimization

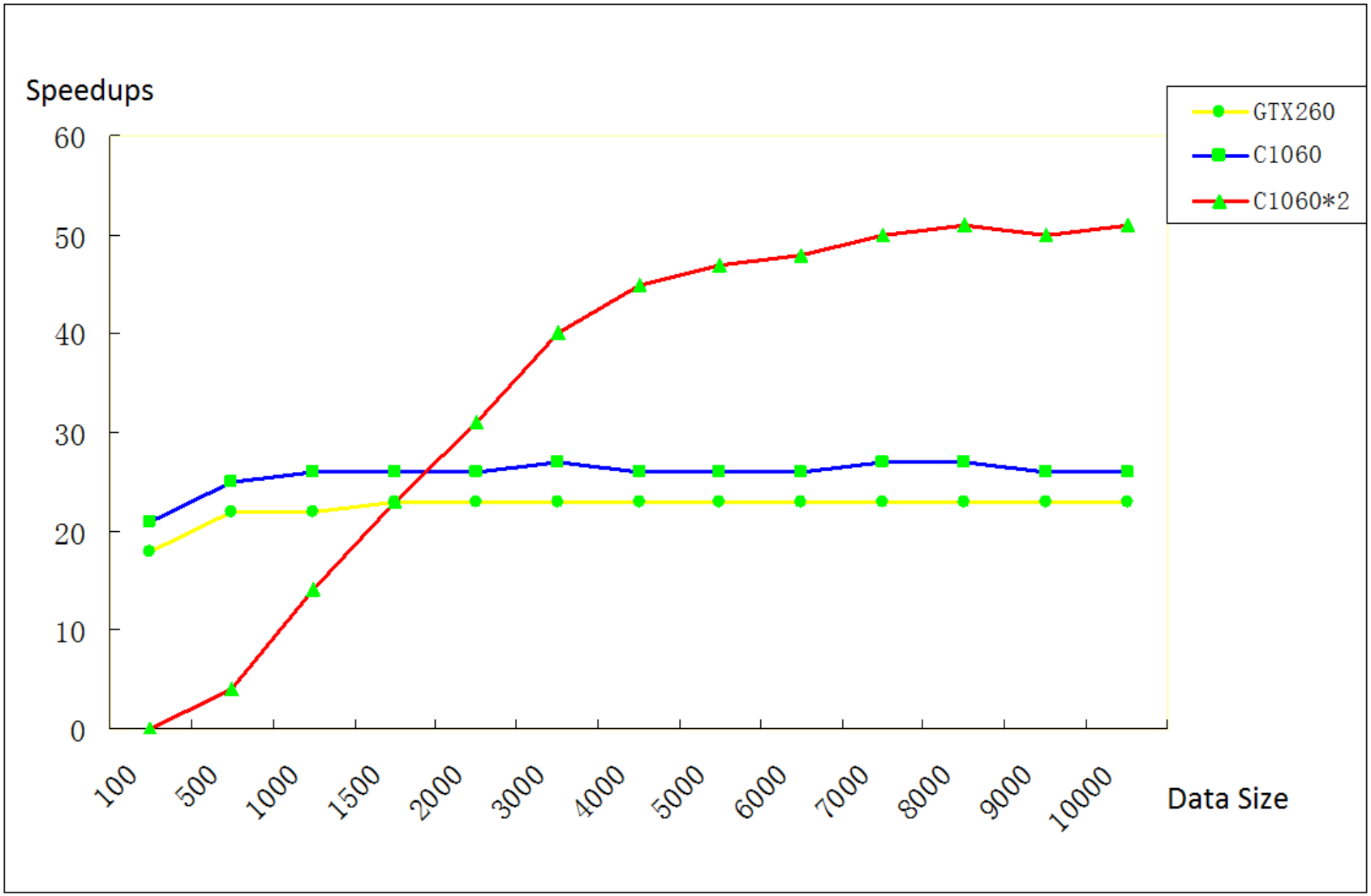
- Organize the data *32 * sizeof(float)* as basic group unit for each warp (32 threads)
  - Enable coalesced global memory access
  - *avoid bank conflicts in shared memory*
- Shared memory
  - Multiple accessed arrays will be put into shared memory
  - Significant results for GPU before Fermi architecture

Experimental Results

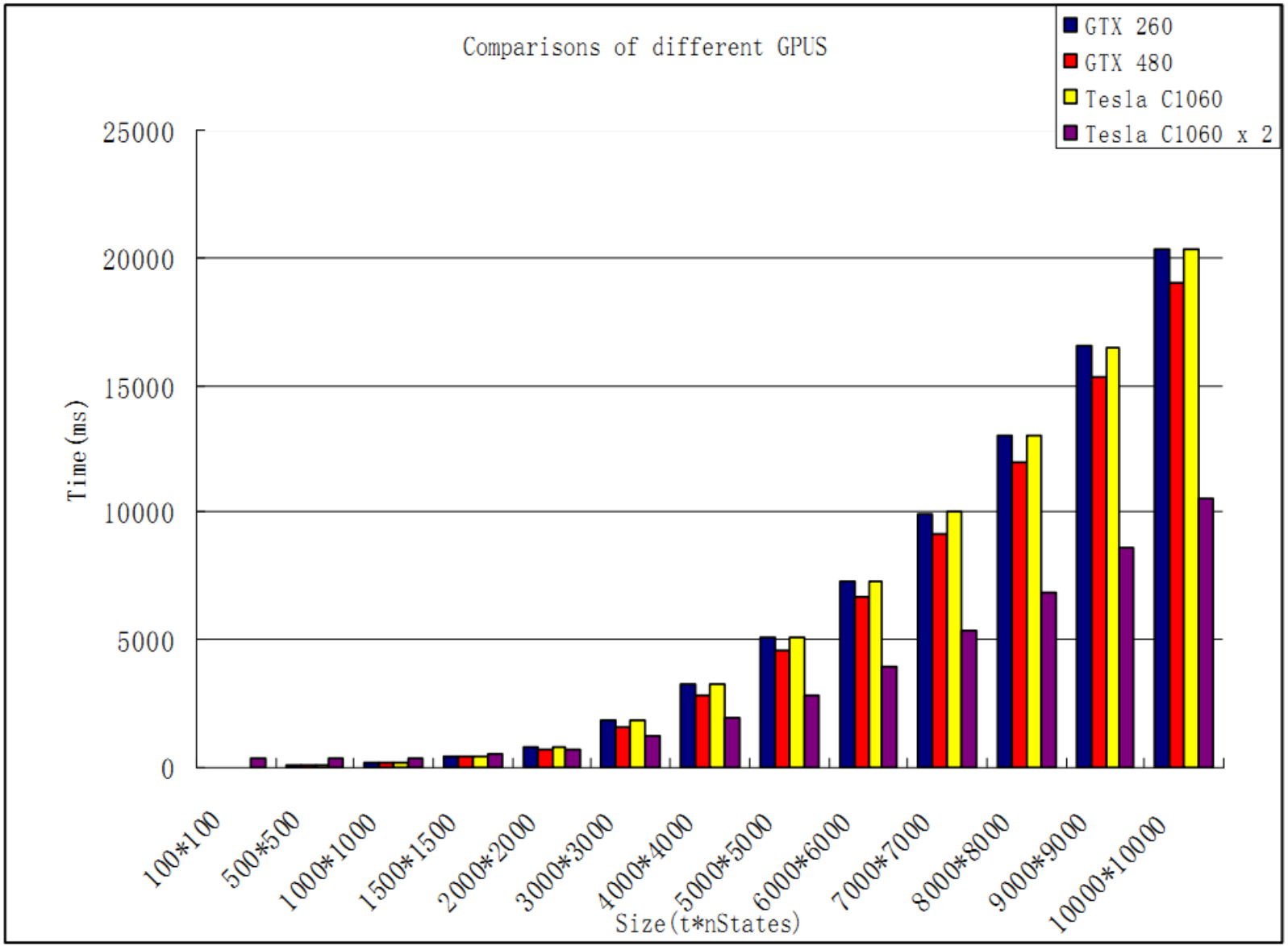
Experimental Setup

- Three different hardware platforms
  - 2.67GHz Intel i5 with NVIDIA GeForce GTX 260
  - 2.93GHz Intel X5670 Xeon with NVIDIA GTX 480
  - 2.27GHz E5520 Intel Xeon with two NVIDIA Tesla C1060
- Different size
  - Number of states*number of observations
  - 100*100->10000*10000

Speedups

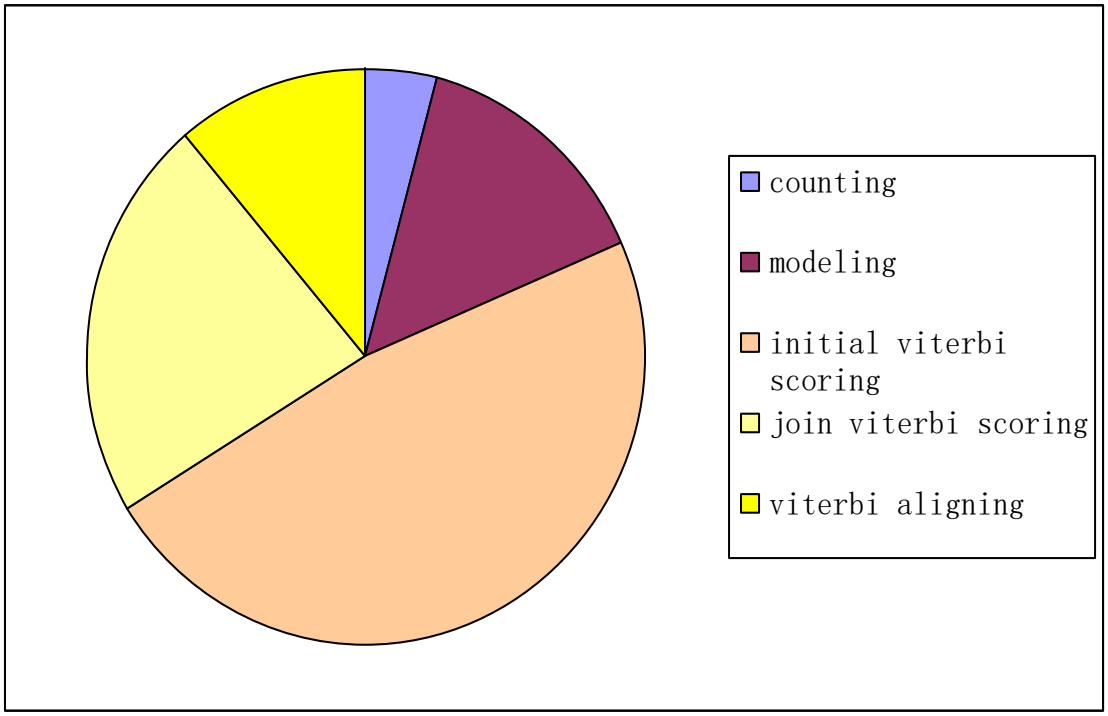

Speedups
60
50
40
30
20
10
0
100
500
1000
1500
2000
3000
4000
5000
6000
7000
8000
9000
10000
Data Size
GTX260
C1060
C1060*2

Execution time

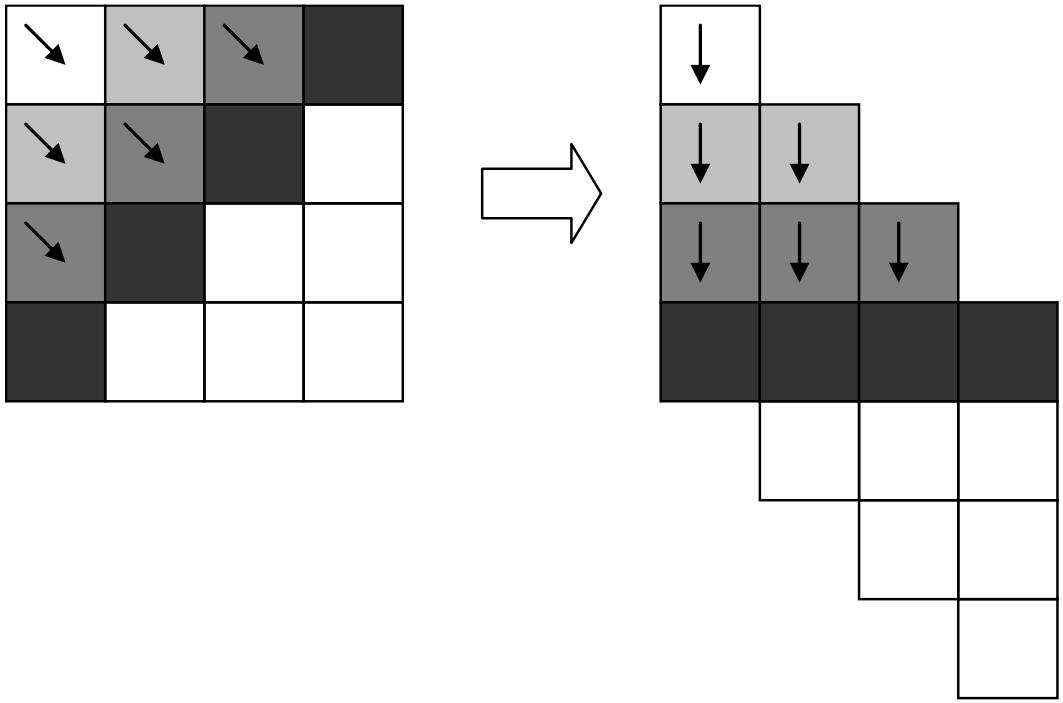


GPU加速序列比对Viterbi算法

The Importance of Accelerating Viterbi Algorithm



Wave-front Parallel Strategy

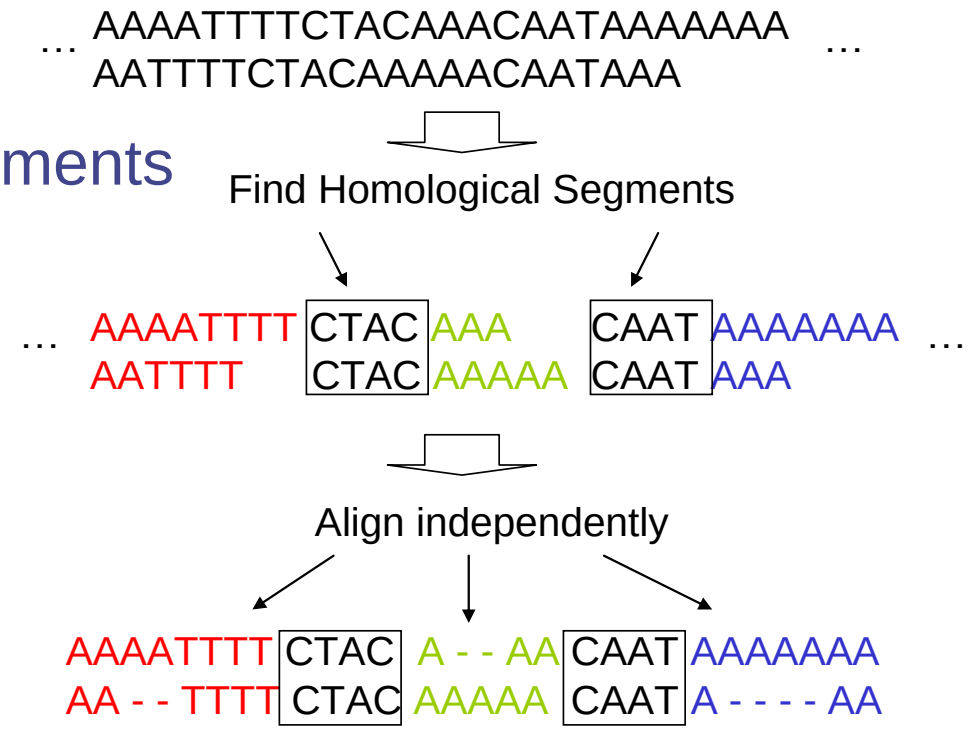


Problem of existing wave-front method

- Can not process long sequence
- Can not process between different wave in parallel

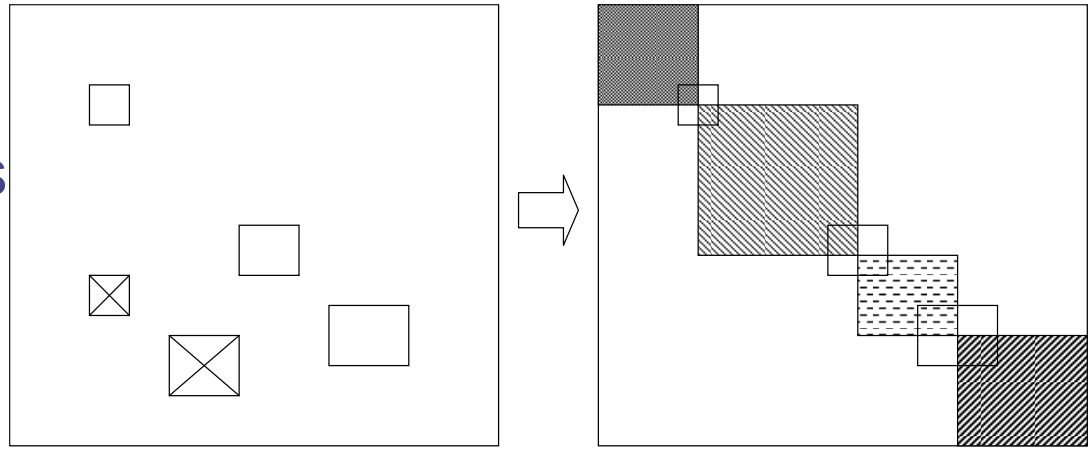
The Tile Based Algorithm

Step1:
Utilizing Homological Segments to divide long sequence



The Tile Based Algorithm

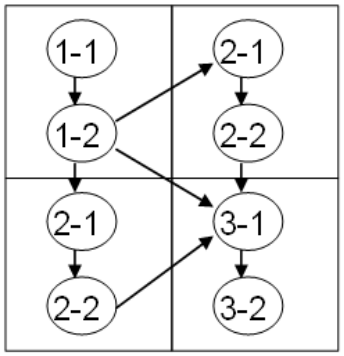
Step2:
Align sub-matrices

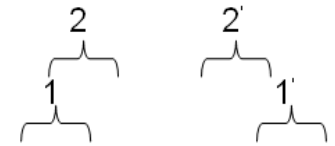


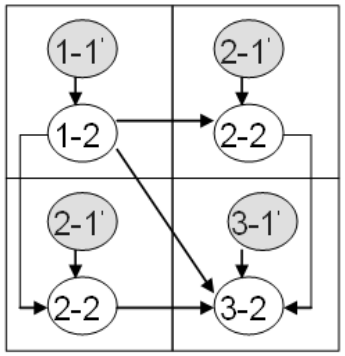
Find homological segment pairs

Divide sequence(shaded area)

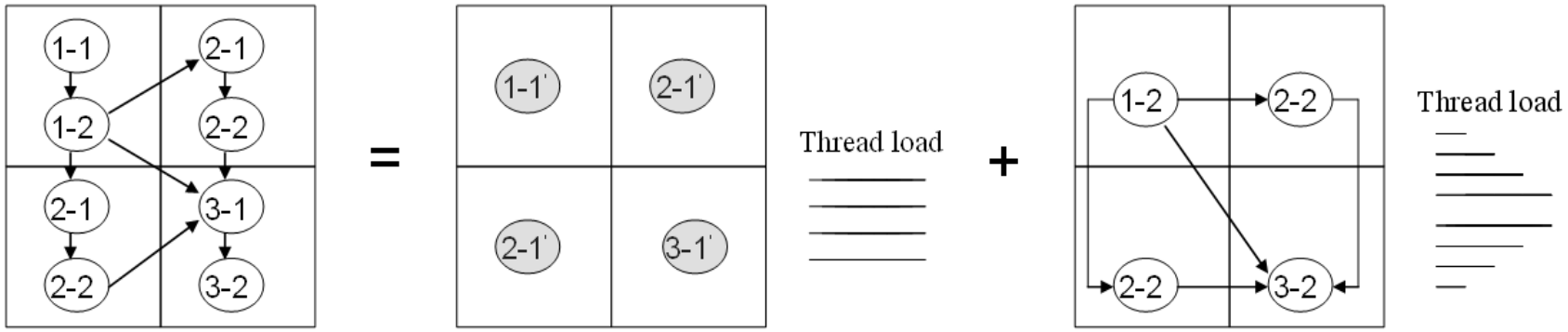
Partition of Different Kind of Computation







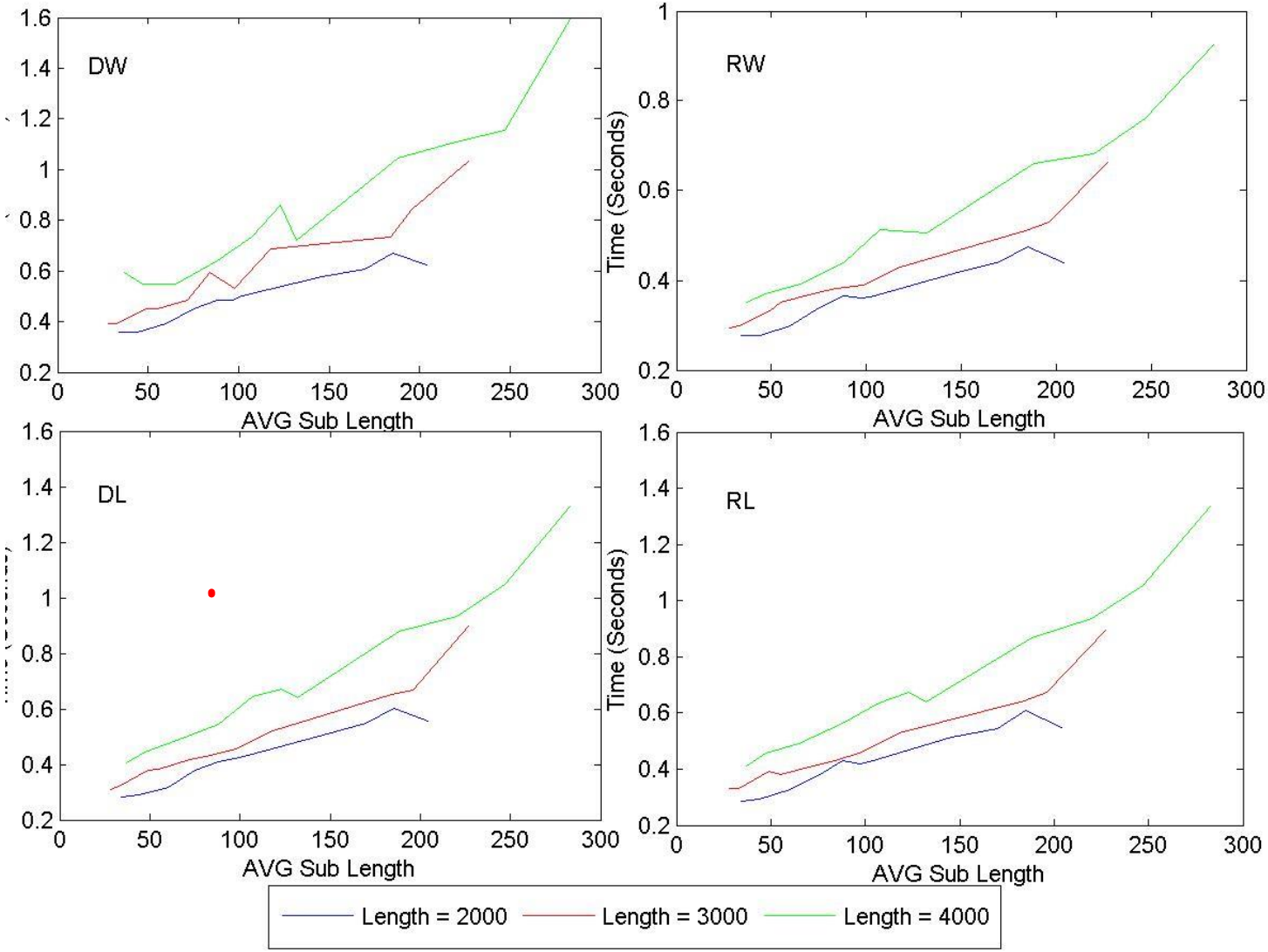
Partition of dependent and independent computations



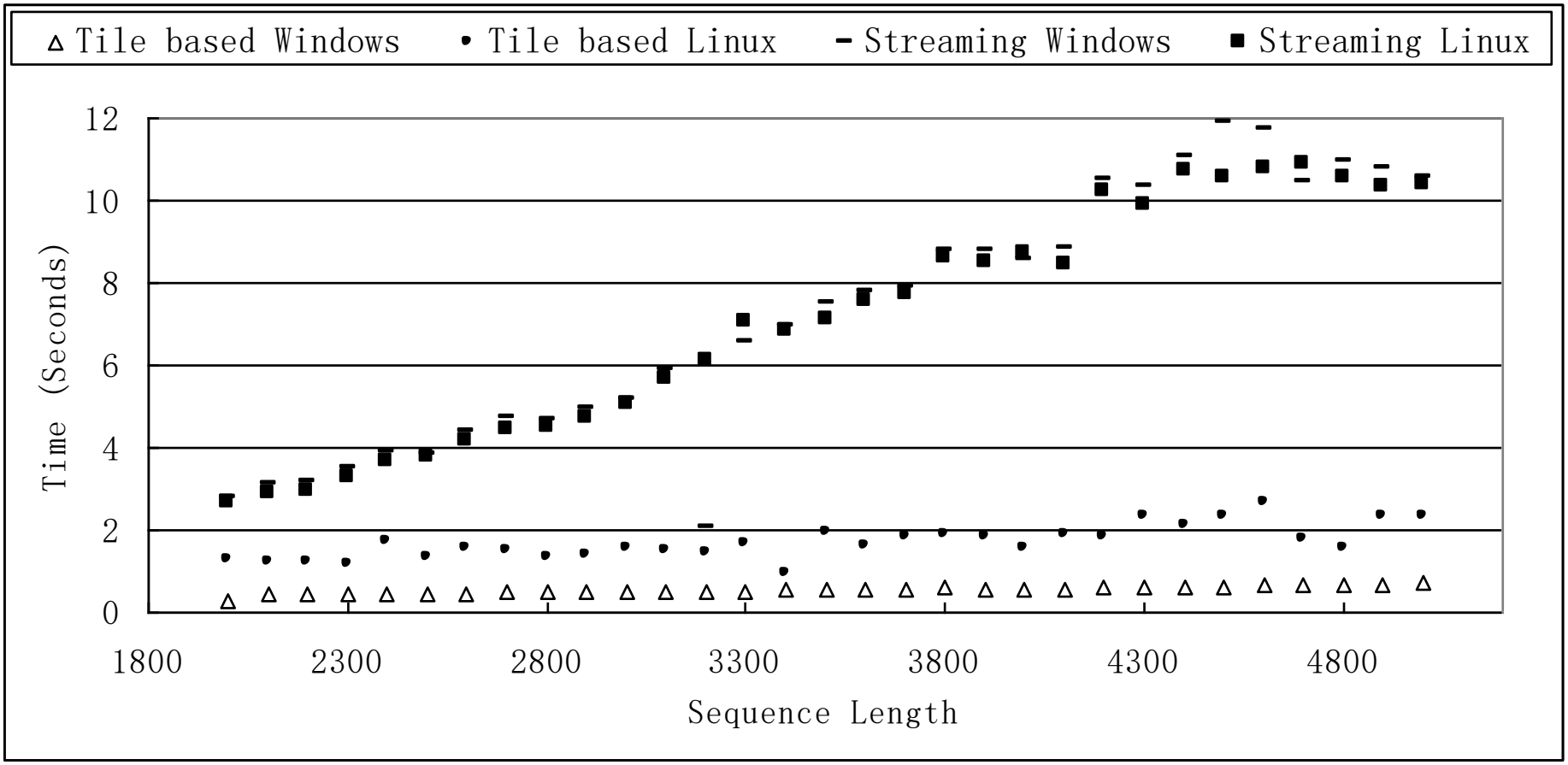
Results for short sequences

Seq-Length		Execution Time (Second)/Speedup						
		serial	*Simple Wave-front*		*Streaming*		*Tile-based*	
100	DW	0.73	0.37	1.97	0.38	1.92	0.28	2.61
	RW	0.017	0.007	2.42	0.02	0.85	0.006	2.83
	DL	0.063	0.008	7.87	0.023	2.74	0.007	9
	RL	0.027	0.007	3.86	0.023	1.17	0.007	3.86
200	DW	2.34	0.39	6	0.44	5.32	0.39	6
	RW	0.05	0.03	1.67	0.061	0.82	0.028	1.79
	DL	0.324	0.035	9.26	0.065	4.98	0.029	11.17
	RL	0.142	0.035	4.06	0.065	2.18	0.029	4.9
300	DW	5.89	0.42	14.02	0.46	12.8	0.43	13.7
	RW	0.12	0.068	1.76	0.1	1.2	0.055	2.18
	DL	0.647	0.07	9.26	0.112	5.78	0.054	11.98
	RL	0.283	0.068	4.16	0.116	2.44	0.054	5.24
400	DW	9.93	0.50	19.86	0.52	19.1	0.45	22.07
	RW	0.21	0.13	1.61	0.159	1.32	0.098	2.14
	DL	1.112	0.12	9.27	0.2	5.56	0.099	11.23
	RL	0.485	0.122	3.98	0.174	2.79	0.097	5
500	DW	15.9	0.54	29.44	0.54	29.44	0.52	30.58
	RW	0.34	0.19	1.78	0.239	1.42	0.174	1.95
	DL	1.783	0.198	9	0.262	6.8	0.155	11.5
	RL	0.783	0.191	4.10	0.251	3.12	0.153	5.12
1000	DW	62.1	0.99	62.73	1.10	56.45	0.86	72.21
	RW	1.34	0.64	2.09	0.686	1.95	0.554	2.42
	DL	6.98	0.64	10.91	0.725	9.63	0.53	13.17
	RL	3.07	0.635	4.83	0.62	4.952	0.512	6.0

Test of Tile based Algorithm



Test of Long Sequences



练习

请比较CPU与GPU提高性能手段的不同并指出GPU性能优化的手段。