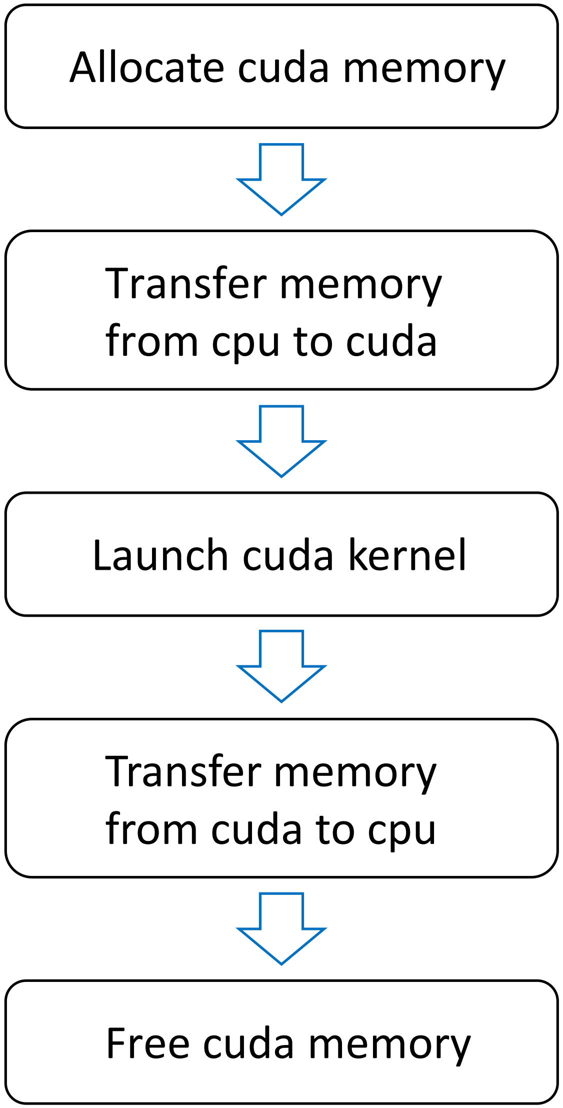
CUDA acceleration for ACTS seed finding

Beomki yeo (KAIST)

General flow of CUDA computing



Allocate cuda memory

cudaMalloc(void** devPtr, int size)

Transfer memory from cpu to cuda

cudaMemcpy(devPtr, hostPtr, int size, cudaMemcpyHostToDevice)

Launch cuda kernel

doSomething<<< grid size, block size >>>(devPtr1, devPtr2)

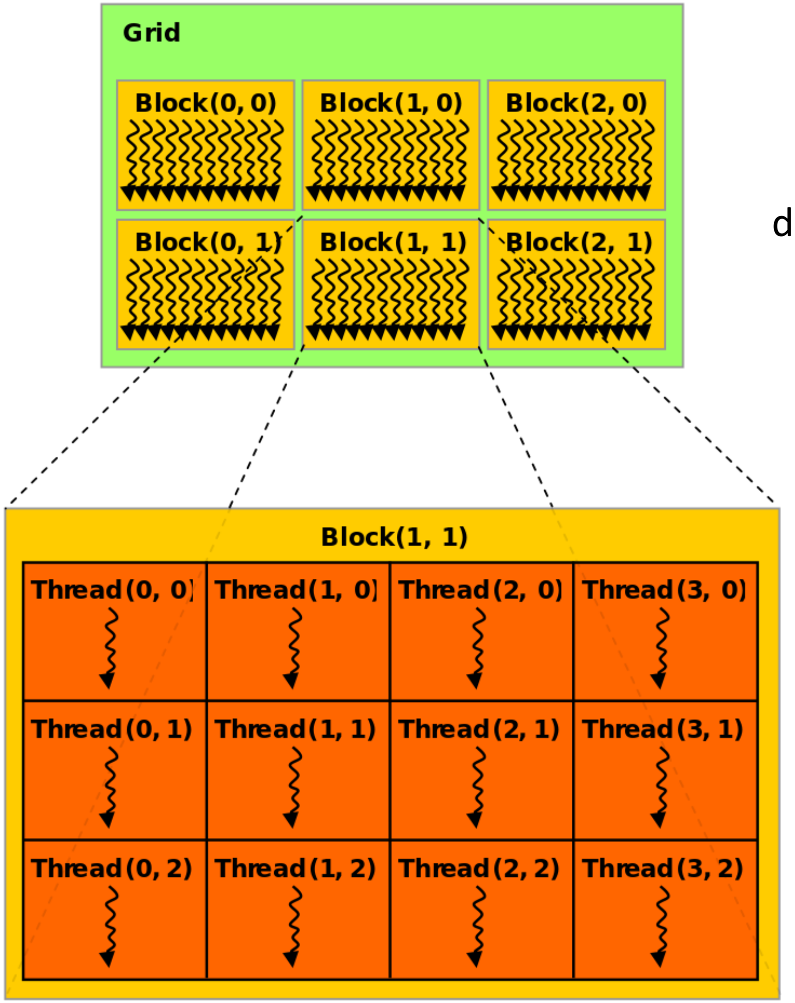
Transfer memory from cuda to cpu

cudaMemcpy(hostPtr, devPtr, int size, cudaMemcpyDeviceToHost)

Free cuda memory

cudaFree(devPtr)

CUDA kernel



- A kernel is launched with given size of grid and block

doSomething<<< grid size, block size >>>(devPtr1, devPtr2)

- threads in different blocks can not communicate each other, in principle.
→ Independent tasks are split into each block

- User can refer the thread ID and block ID inside kernel

Data structure for fast memory access

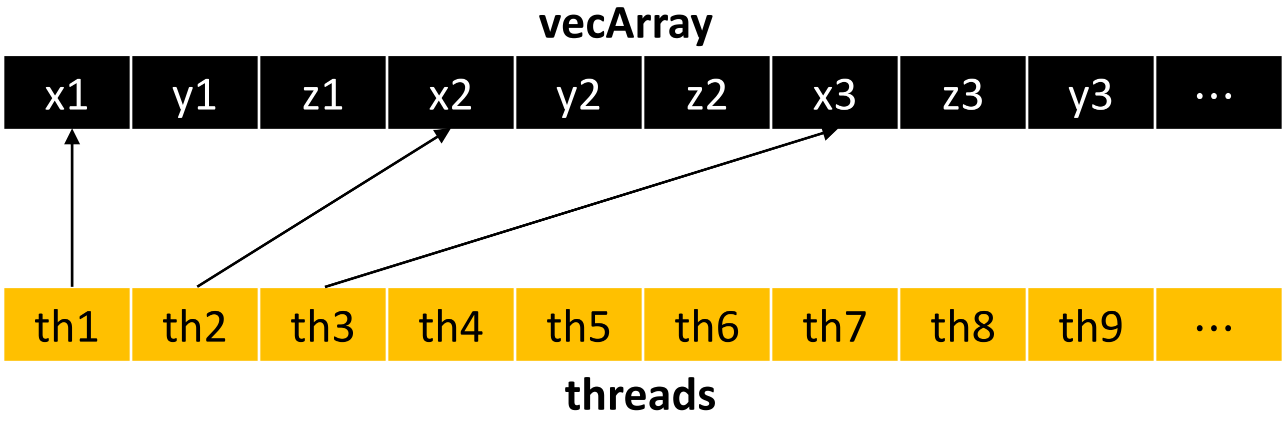
- When accessed data is aligned about threads, the number of data transaction required is reduced (it gets faster)

- Each thread of CUDA can access only 1, 2, 4, 8, 16 Bytes at once

Ex1) 3D vector (x,y,z) handling (misaligned memory access)

```
3dVectorHandle(Vector3D* vecArray){
  Vector3D v = vecArray[threadIdx.x];
  // this is similar to…
  // float x = vecArray[threadIdx.x][0];
  // float y = vecArray[threadIdx.x][1];
  // float z = vecArray[threadIdx.x][2];
}
```

Memory is NOT aligned against thread access



Data structure for fast memory access (cont.)

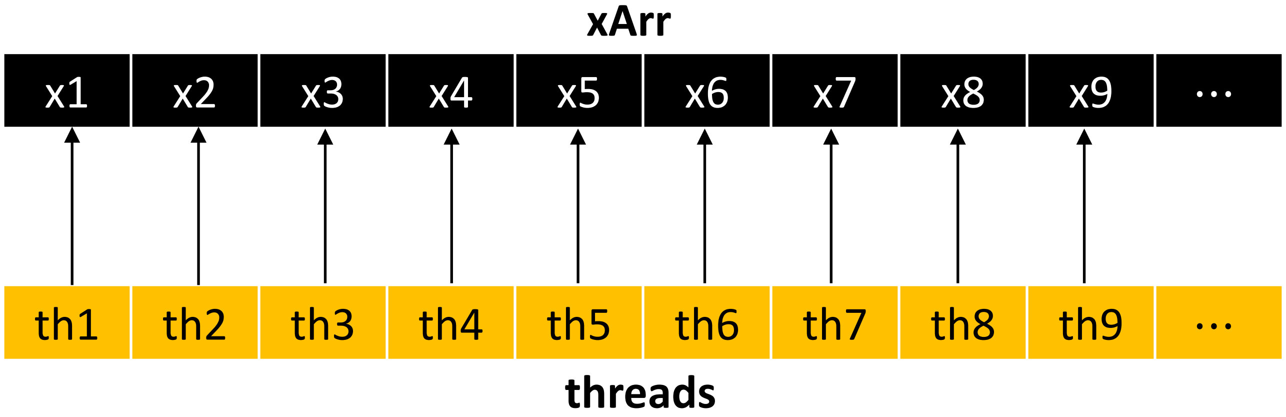
- When accessed data is aligned about threads, number of data transaction is reduced (it gets faster)

- Each thread of CUDA can access only 1, 2, 4, 8, 16 Bytes at once

Ex2) Aligned memory access

```
3dVectorHandle(float* xArr, float* yArr, float* zArr){
  float x = xArr[threadIdx.x];
  float y = yArr[threadIdx.x];
  float z = zArr[threadIdx.x];
}
```

Memory is Aligned against thread access



xArr

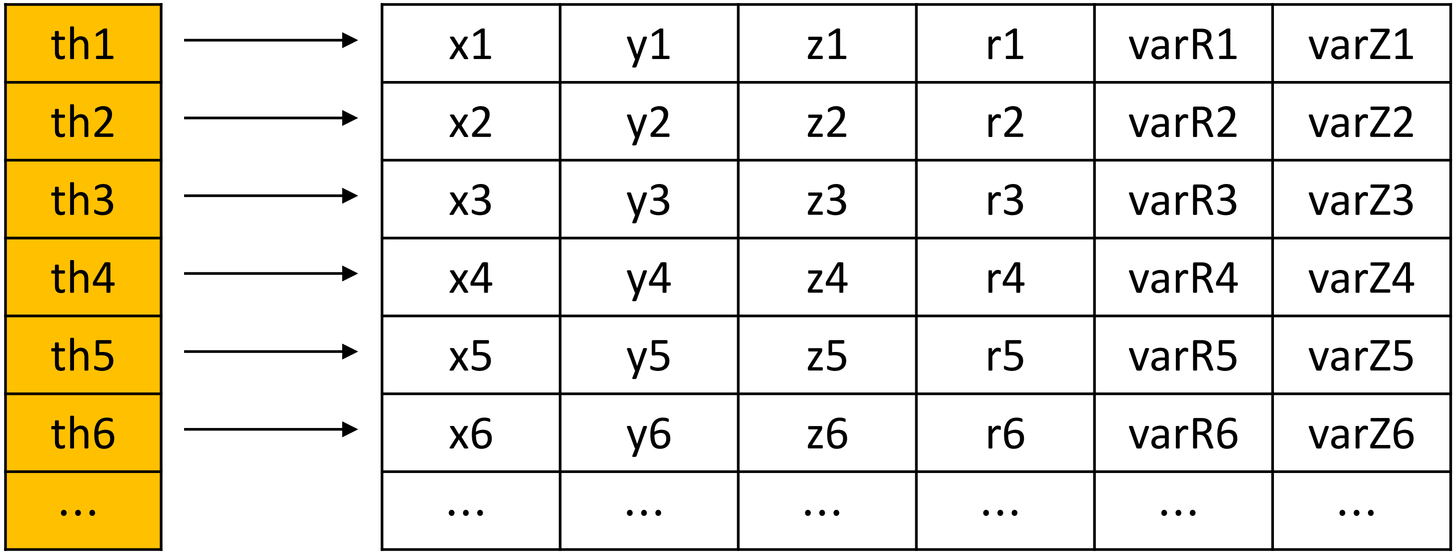
x1 x2 x3 x4 x5 x6 x7 x8 x9 …

th1 th2 th3 th4 th5 th6 th7 th8 th9 …

threads

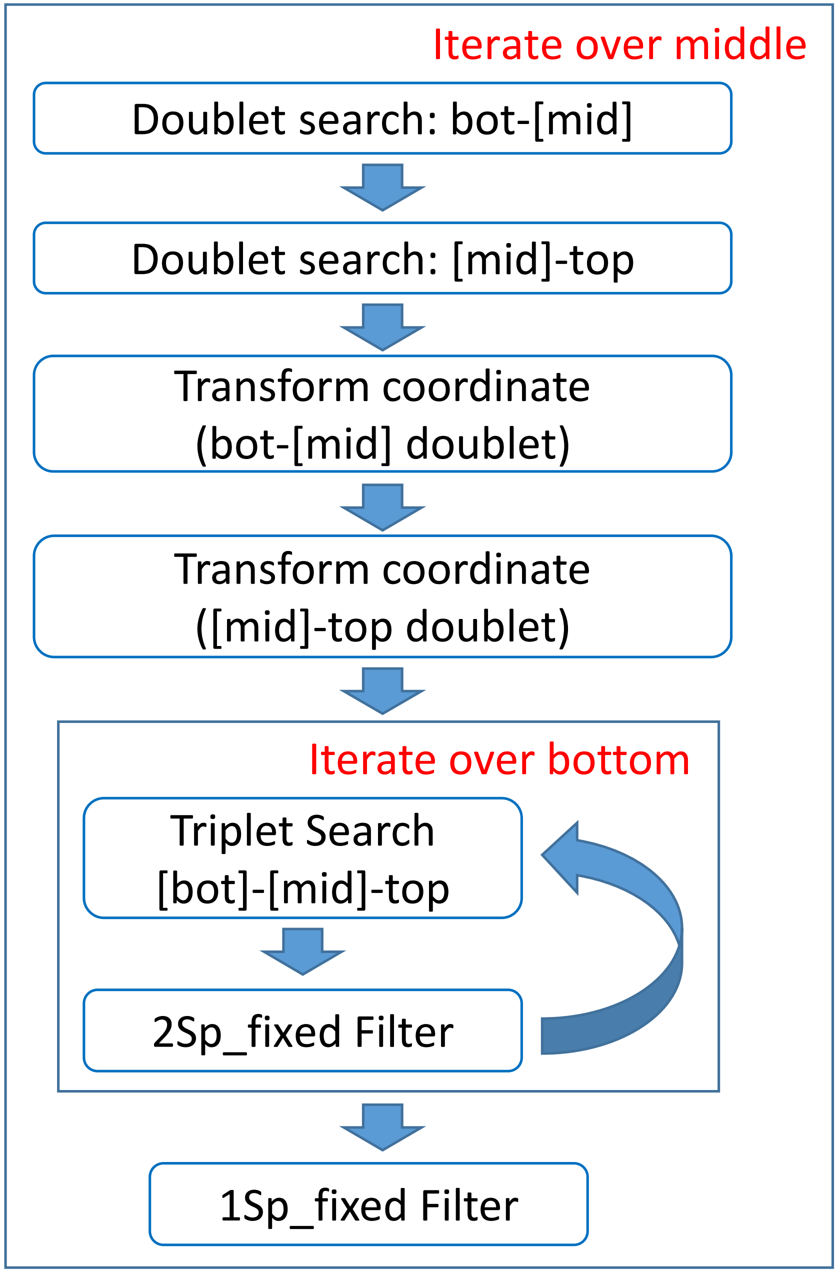
Data structure for fast memory access (cont.)

- All spacepoints data were flattened into the matrix (column-major) which respects the aligned data structure



th1	→	x1	y1	z1	r1	varR1	varZ1
th2	→	x2	y2	z2	r2	varR2	varZ2
th3	→	x3	y3	z3	r3	varR3	varZ3
th4	→	x4	y4	z4	r4	varR4	varZ4
th5	→	x5	y5	z5	r5	varR5	varZ5
th6	→	x6	y6	z6	r6	varR6	varZ6
…		…	…	…	…	…	…

CPU algorithms for a group of space points



Iterate over middle

Doublet search: bot-[mid]

Doublet search: [mid]-top

Transform coordinate (bot-[mid] doublet)

Transform coordinate ([mid]-top doublet)

Iterate over bottom

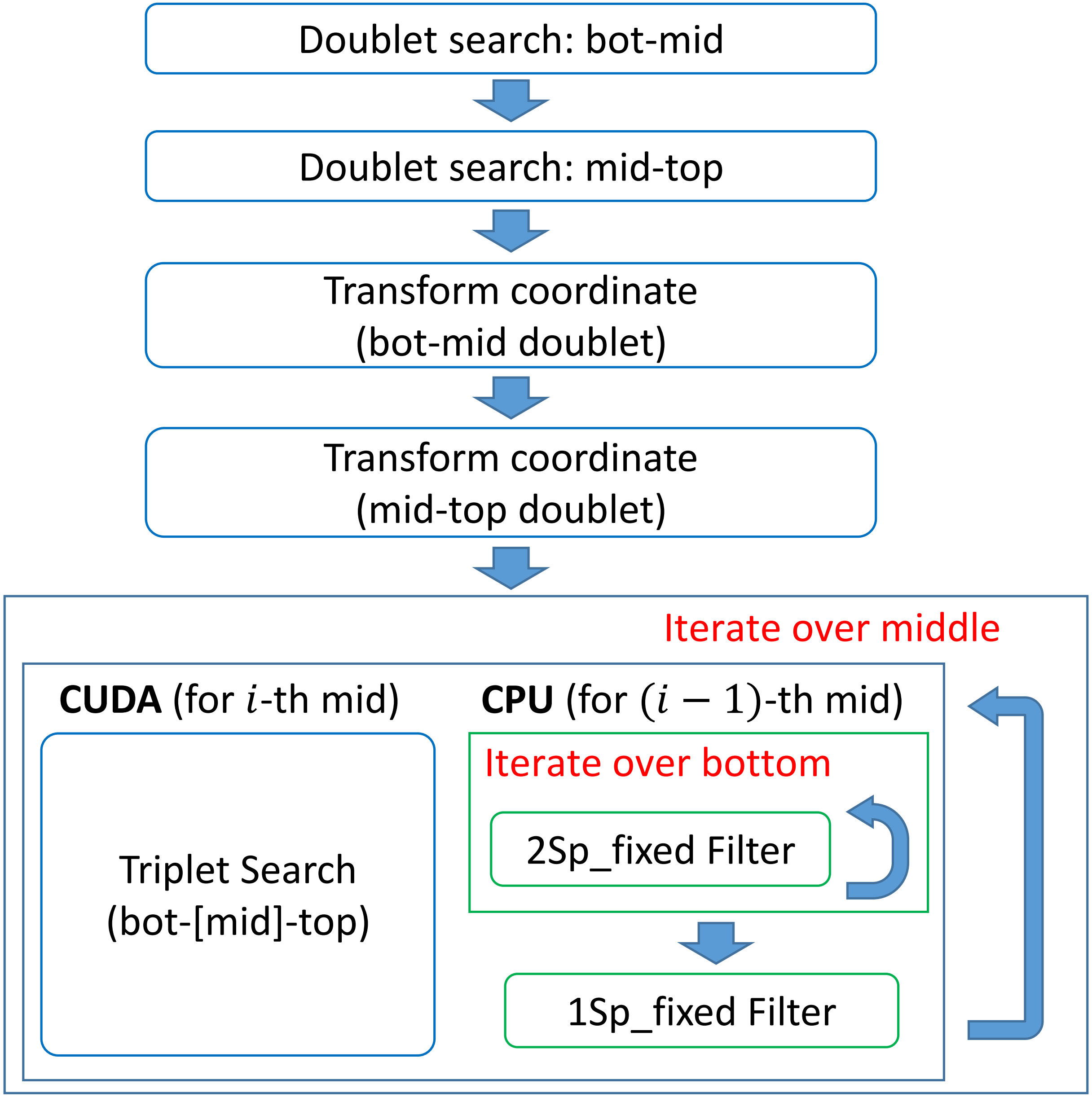
Triplet Search [bot]-[mid]-top

2Sp_fixed Filter

1Sp_fixed Filter

[*]: fixed

CUDA algorithms for a group of space points



Doublet search: bot-mid

Doublet search: mid-top

Transform coordinate (bot-mid doublet)

Transform coordinate (mid-top doublet)

Iterate over middle

CUDA (for i-th mid)

Triplet Search (bot-[mid]-top)

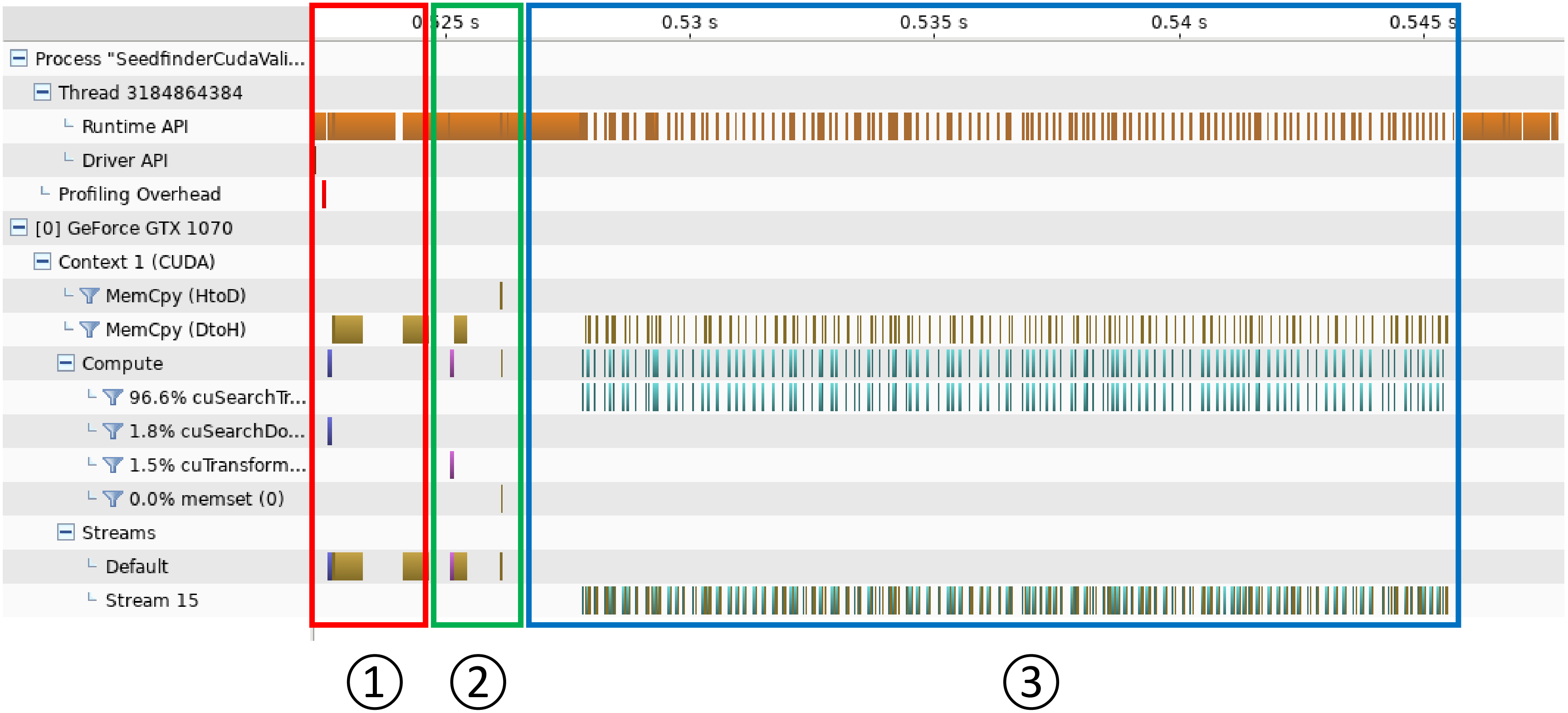
CPU (for $(i-1)$-th mid)

Iterate over bottom

2Sp_fixed Filter

1Sp_fixed Filter

Timeline for CUDA



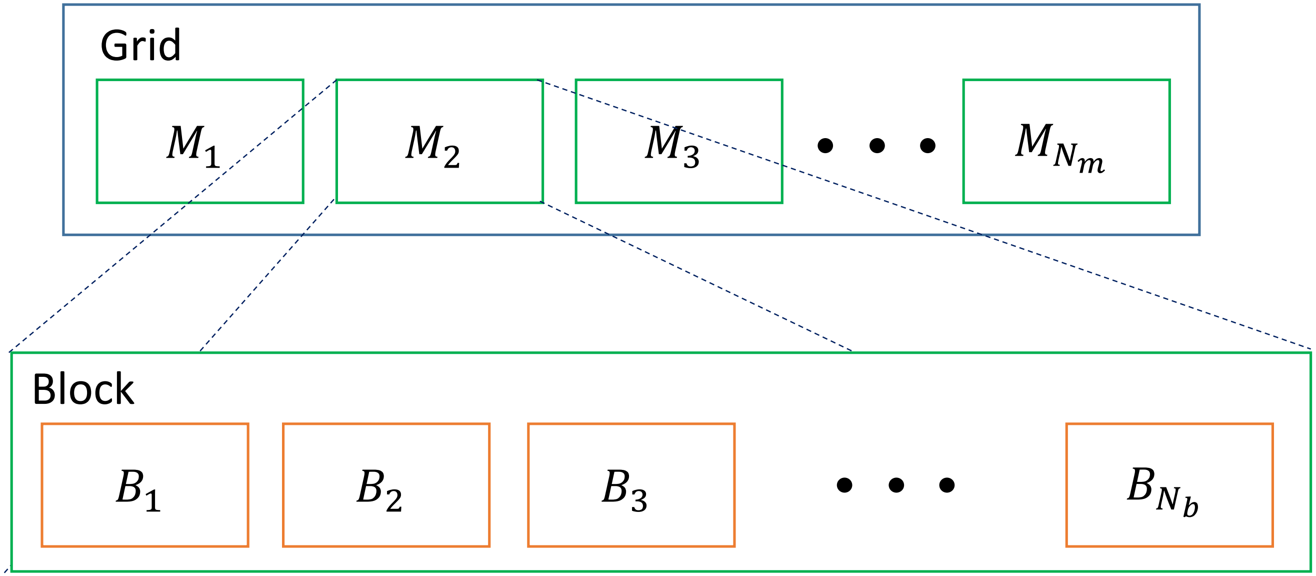
① Doublet Search

② Transform coordinate

③ Triplet Search + SeedFilter (CPU)

Doublet Search (for middle-bottom)

- Input : spacepoint data of middle and botton hits
- Output: index of compatible hits that form doublets



Grid

M_1 M_2 M_3 • • • M_{N_m}

Block

B_1 B_2 B_3 • • • B_{N_b}

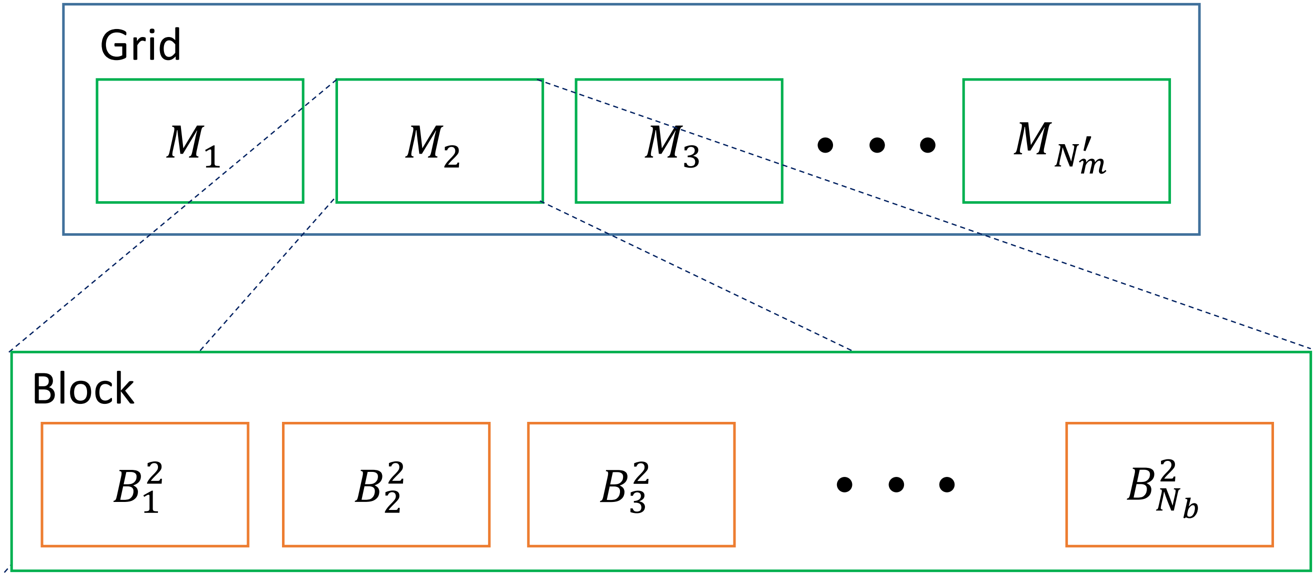
- Since the maximum number of threads per block is 1024,
the same kernel is iterated over different sets of bottom hits if $N_b > 1024$

Transform coordinate (for middle-bottom)

Input : spacepoint data middle and bottom hits
index of compatible hits,
Output: (Reduced) spacepoint data transformed into circle

· The matrix size of the output is reduced from the original data to the number of doublets

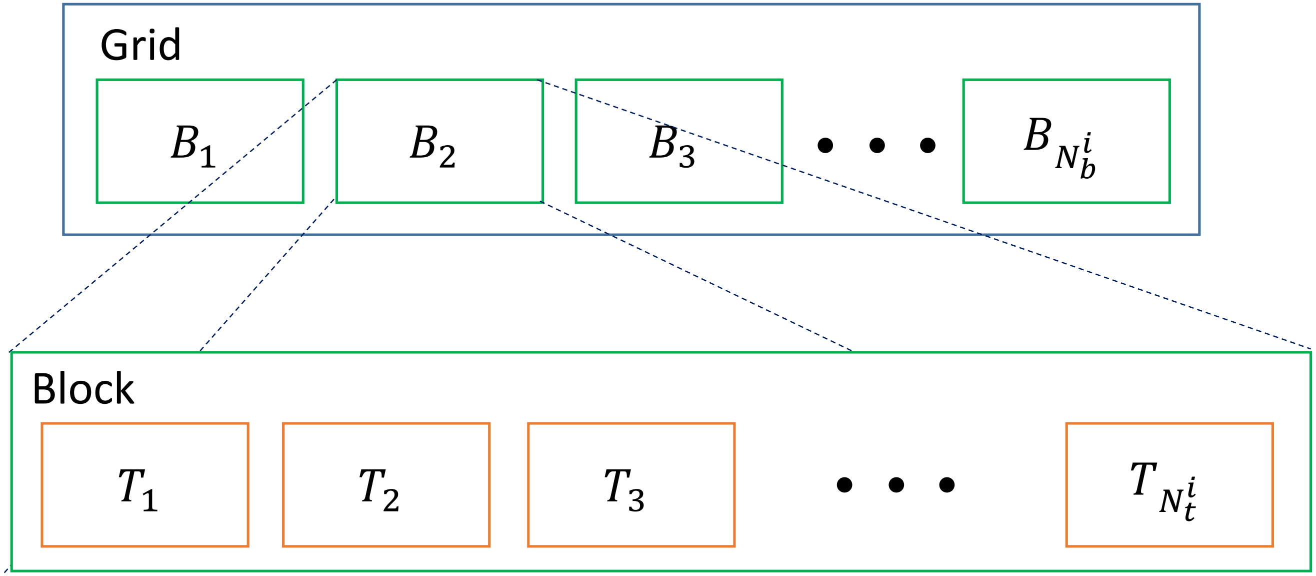
· The structure is similar to the doublet search



Triplet search (for a middle hit)

Input for the triplet search of i-th middle hit:

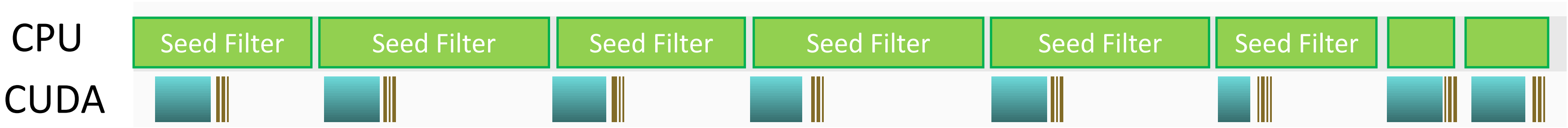
$\left(N_b^i\right)$ hit & circle matrix and $\left(N_t^i\right)$ hit & circle matrix



output for the triplet search of i-th middle hit:

1) Number of top hits which form triplets for every bottom hit
2) curvature and impact parameters of triplets

Timeline of triplet search



- Triplet search for i-th mid and seed filtering for $(i-1)$-th mid are done asynchronously

- Seed filtering done by Cpu is the biggest bottleneck

results comparison (CUDA vs. CPU)

- CPU model: i7-5820K CPU @ 3.3 GHz
- GPU model: GTx 1070

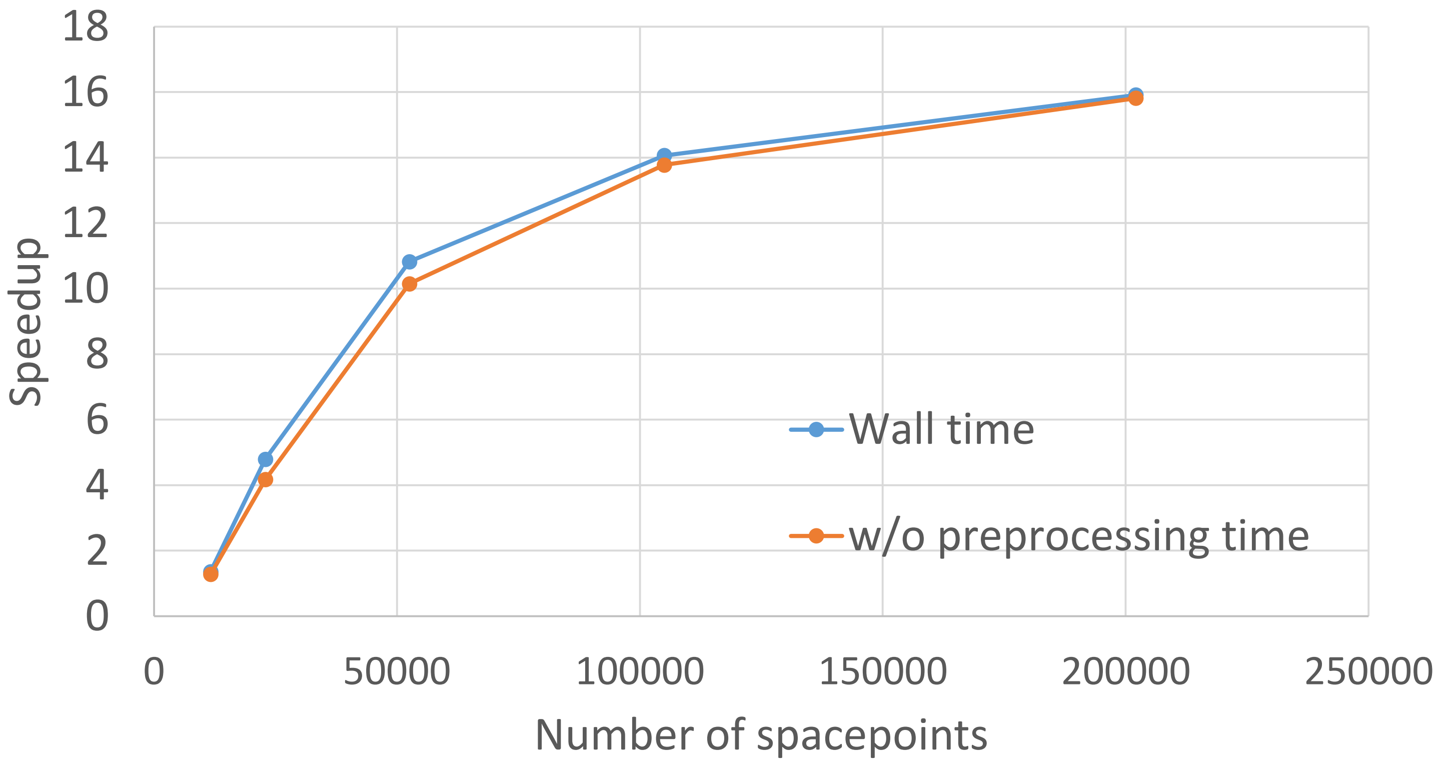
# space points	# of seeds (CUDA)	# of seeds (CPU)	Seed matching ratio
10k	2690	2690	100%
20k	5512	5512	100%
50k	12805	12805	100%
100k	25572	25572	99.99%
200k	49302	49302	99.97%

- Seed matching ratio is almost 100%
- the mismatches happen due to the different rounding policy between CPU and GPU

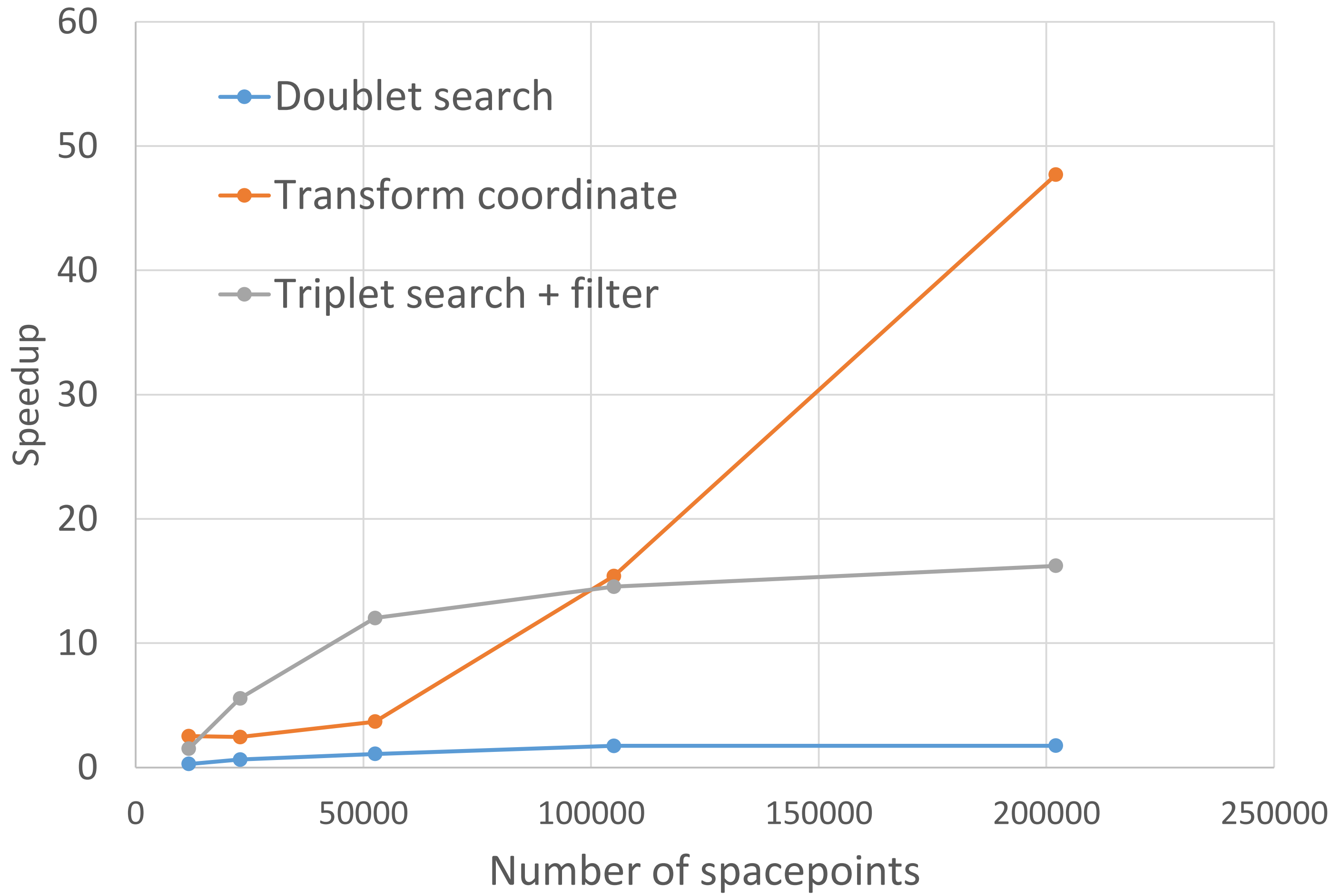
Wall-time speedup (Release mode (-O2))

- Speedup was measured by comparing the wall time:
(preprocessing + CPU seed finding) / (preprocessing + CUDA seed finding)

- Preprocessing includes data reading and grouping



Speedup for each algorithm (Release mode (-O2))



Summary

- Parallelization on the seed finding was done successfully

- Validation tests showed that the seed matching ratio is $\sim 100\ \%$

- Achieved one order of magnitude improvement in speedups for >50k space points