Parallelized Kalman-Filter-Based Reconstruction of Particle Tracks on Many-Core Processors and GPUs

Connecting The Dots 2017: March 8, 2017

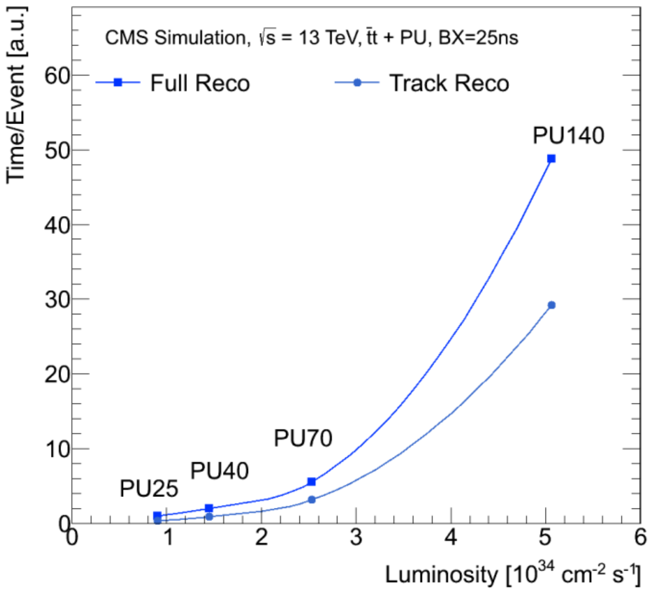
G. Cerati[4], P. Elmer[3], S. Krutelyov[1], S. Lantz[2], M. Lefebvre[3],
M. Masciovecchio[1], K. McDermott[2], D. Riley[2], M. Tadel[1], P. Wittich[2],
F. Würthwein[1], A. Yagil[1]

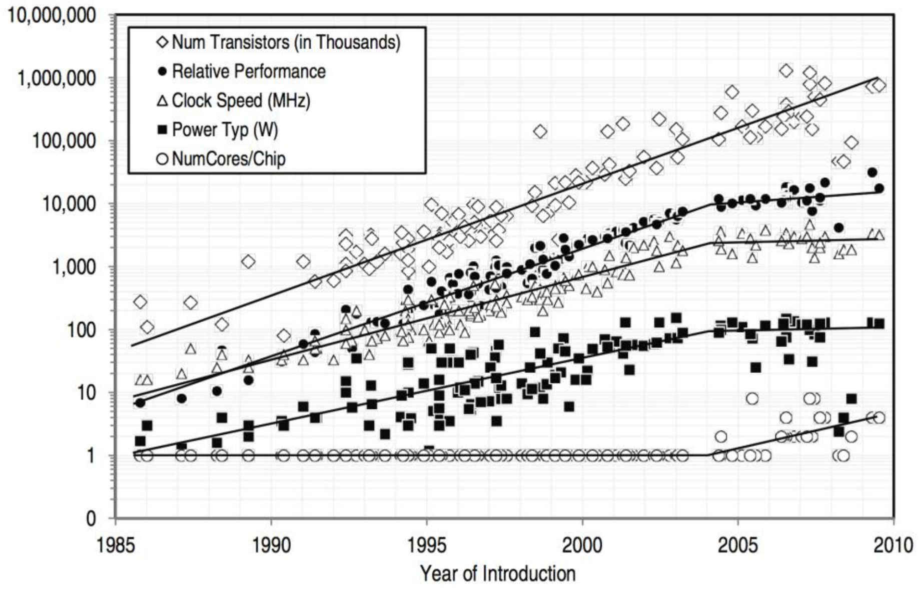
1. University of California – San Diego
2. Cornell University
3. Princeton University
4. Fermilab

Outline

- Problematic & experimental setup
- Parallelizing on x86 processors: Sandy Bridge and Knights Corner
    - Challenges
    - Data structures
    - Algorithmic approaches
    - Results
- Parallelizing on GPUs
    - Porting Strategy
    - Data structures
    - Track fitting: lessons learned
    - Track building: increasing the algorithmic complexity
    - First results
- Avoiding code duplication
- Conclusion & Perspectives

Why Parallelizing?





- By 2025, the instantaneous luminosity of the LHC will increase by a factor of 2.5, transitioning to the High Luminosity LHC
- Increase in detector occupancy leads to significant strain on read-out, selection, and **reconstruction**

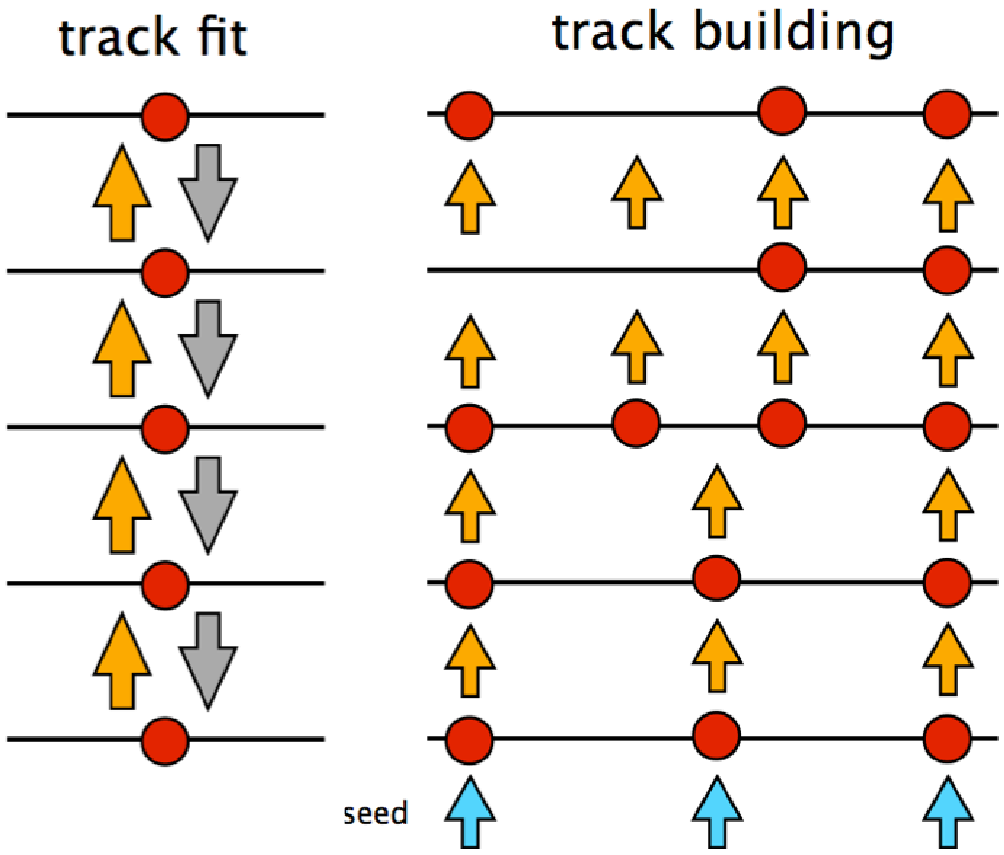
- Clock speed stopped scaling
- Number of transistors keeps doubling every ~18 months

➔ Multi-core architectures

  - E.g. Xeon, MIC, GPUs

KF Track Reconstruction

- Tracking proceeds in three main steps: seeding, building, and fitting
- In fitting, hit collection is known: repeatedly apply the basic logic unit
- In building, hit collection is unknown and requires branching to explore many possible candidate hits after propagation

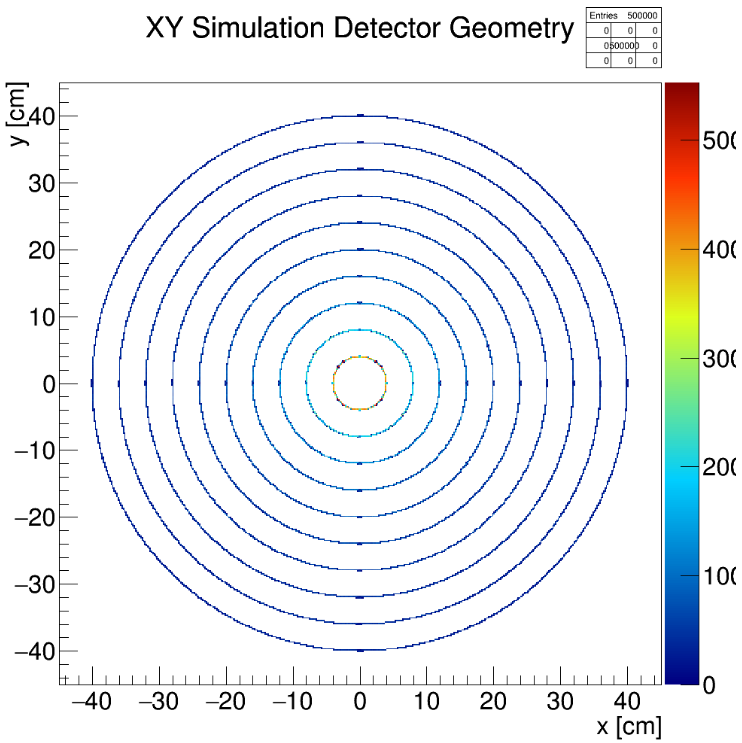


Experimental Setting

Simplified setup

- Detector conditions
  - 10 barrel pixel layers, evenly spaced
  - Hit resolution
    - ➢ $\sigma_{x,y} = 100\mu m$
    - ➢ $\sigma_z = 1.0mm$
  - Constant B-field of 3.8T
  - No scattering/energy loss

- Track conditions
  - Tracks generated with MC simulation uniformly in η,ϕ (azimuthal angle), and p_T
  - Seeding taken from tracks in simulation



Realistic Setup

Options to add material effects, polygonal geometry:

More realistic setup partially built:

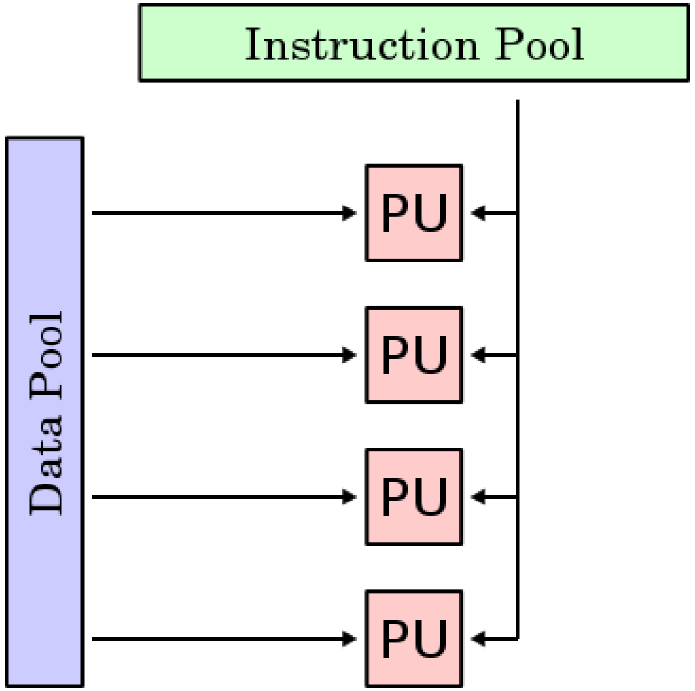
Barrel and Endcap (x86 only)

Selected Parallel Architectures

	Xeon E5-2620	Xeon Phi 7120P	Tesla K20m	Tesla K40
Cores	6 x 2	61	13	12
Logical Cores	12 x 2	244	2496 CUDA cores	2880
Max clock rate	2.5 GHz	1.333 GHz	706 MHz	745 MHz
GFLOPS (double)	120	1208	1170	1430
SIMD width	64 bytes	128 bytes	Warp of 32	Warp of 32
Memory	~64-384 GB	16 GB	5 GB	12 GB
Memory B/W	42.6 GB/s	352 GB/s	208 GB/s	288 GB/s

Challenges to Parallel Processing

- KF tracking cannot be ported in straightforward way to run in parallel
- Need to exploit two types of parallelism with parallel architectures
- **Vectorization**
  - Perform the same operation at the same time in lock-step across different data
  - **Challenge: branching** in track building - exploration of multiple track candidates per seed
- **Parallelization**
  - Perform different tasks at the same time on different pieces of data
  - **Challenge: thread balancing** – splitting the workload evenly is difficult as track occupancy in the detector not uniform on a per event basis

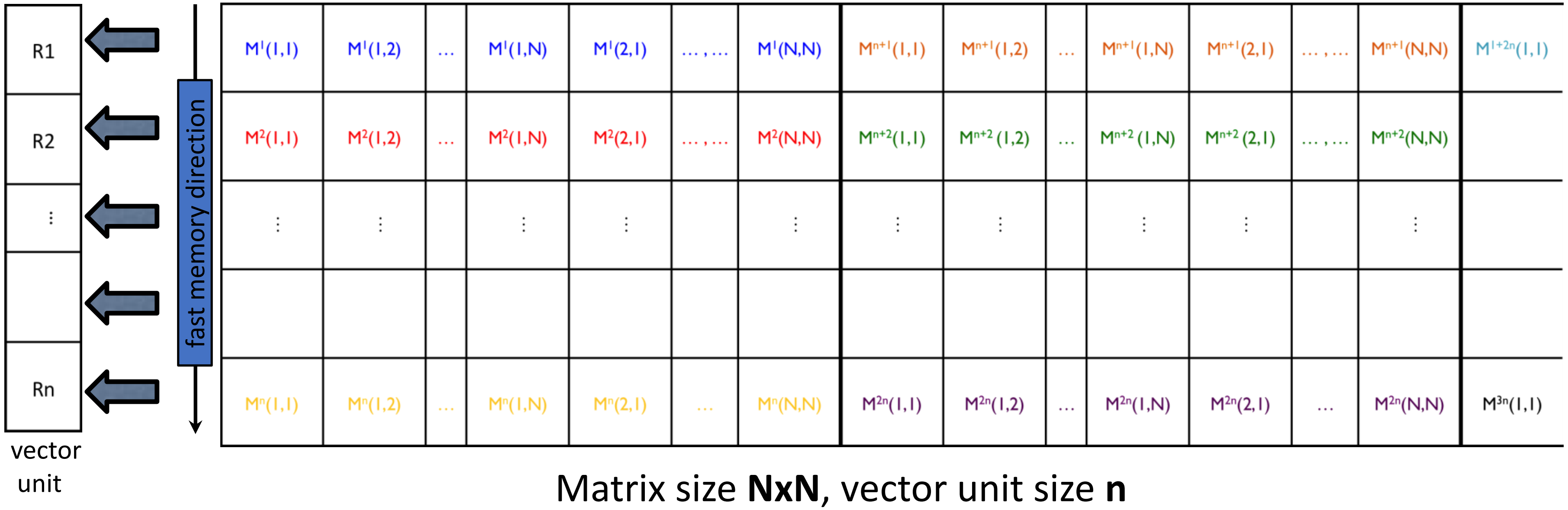


Vectorization

Matriplex

- Matrix operations of KF **ideal for vectorized processing:** however, requires **synchronization** of operations

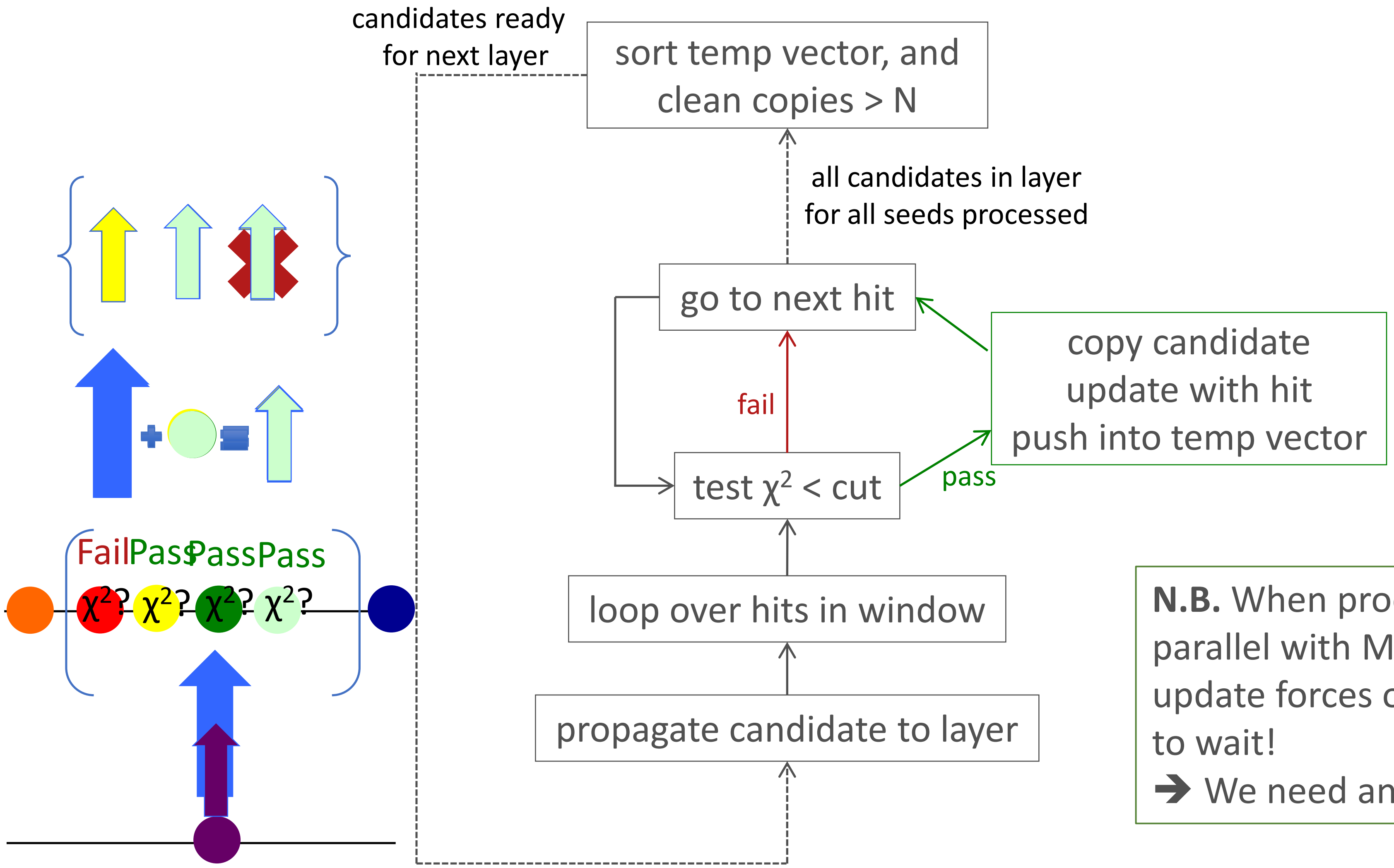
- Arrange data in such a way that it can loaded into the vector units of Xeon and Xeon Phi with ***Matriplex***
  - Fill vector units with the same matrix element from different matrices: **n matrices working in synch on same operation**



R1	$M^1(1,1)$	$M^1(1,2)$	…	$M^1(1,N)$	$M^1(2,1)$	… , …	$M^1(N,N)$	$M^{n+1}(1,1)$	$M^{n+1}(1,2)$	…	$M^{n+1}(1,N)$	$M^{n+1}(2,1)$	… , …	$M^{n+1}(N,N)$	$M^{1+2n}(1,1)$
R2	$M^2(1,1)$	$M^2(1,2)$	…	$M^2(1,N)$	$M^2(2,1)$	… , …	$M^2(N,N)$	$M^{n+2}(1,1)$	$M^{n+2}(1,2)$	…	$M^{n+2}(1,N)$	$M^{n+2}(2,1)$	… , …	$M^{n+2}(N,N)$	
⋮	⋮	⋮		⋮	⋮		⋮	⋮	⋮		⋮	⋮		⋮	
Rn	$M^n(1,1)$	$M^n(1,2)$	…	$M^n(1,N)$	$M^n(2,1)$	…	$M^n(N,N)$	$M^{2n}(1,1)$	$M^{2n}(1,2)$	…	$M^{2n}(1,N)$	$M^{2n}(2,1)$	…	$M^{2n}(N,N)$	$M^{3n}(1,1)$

Matrix size **NxN**, vector unit size **n**

Handling Multiple Track Candidates: First Approach



candidates ready for next layer

sort temp vector, and clean copies > N

all candidates in layer for all seeds processed

go to next hit

copy candidate update with hit push into temp vector

fail

pass

test $\chi^2 <$ cut

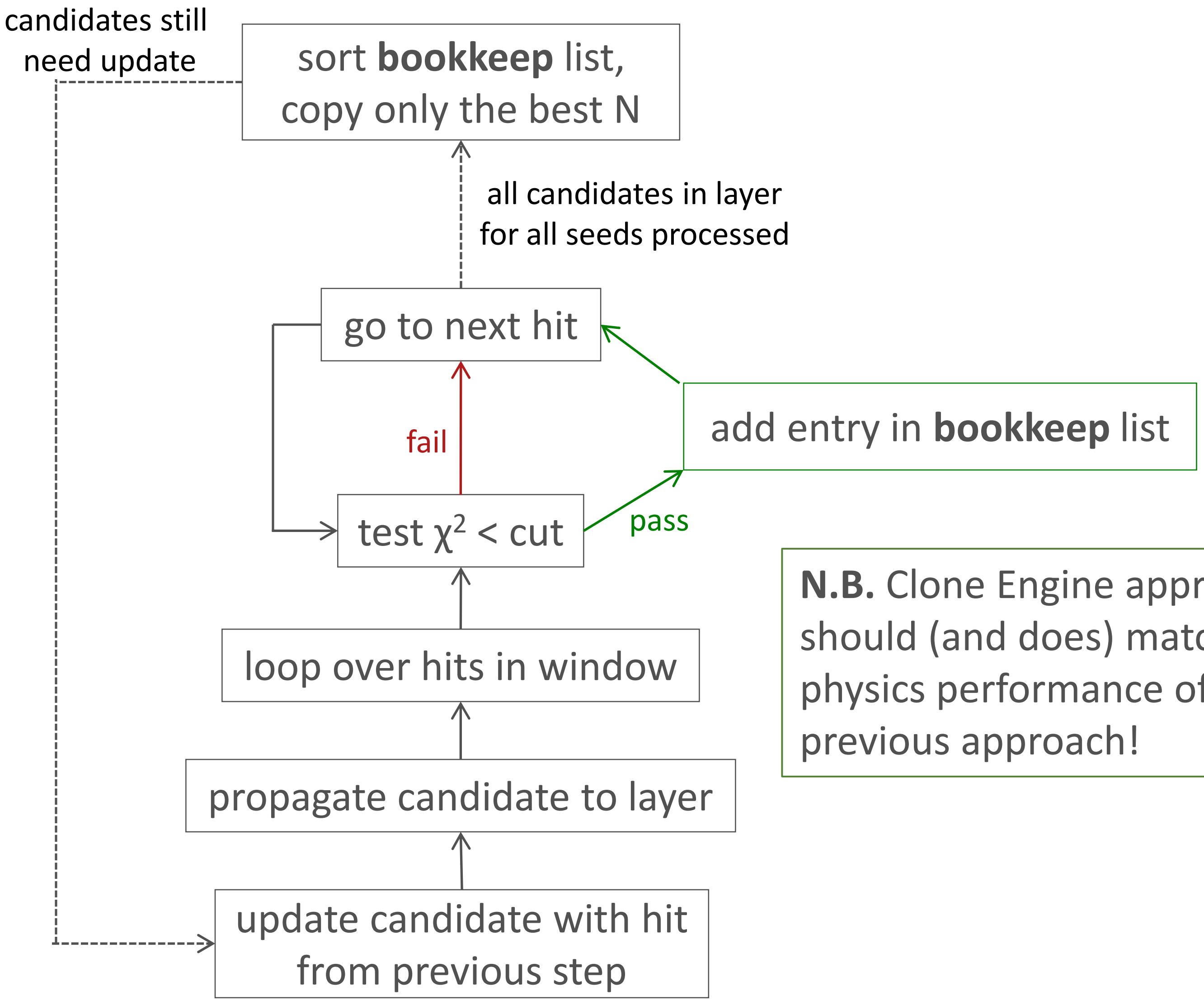
loop over hits in window

propagate candidate to layer

N.B. When processing tracks in parallel with Matriplex, copy + update forces other processes to wait!

➔ We need an other approach

Optimized handling of multiple candidates: “Clone Engine”


candidates still need update
sort **bookkeep** list, copy only the best N
all candidates in layer for all seeds processed
go to next hit
fail
add entry in **bookkeep** list
test χ² < cut
pass
loop over hits in window
propagate candidate to layer
update candidate with hit from previous step

N.B. Clone Engine approach should (and does) match physics performance of previous approach!

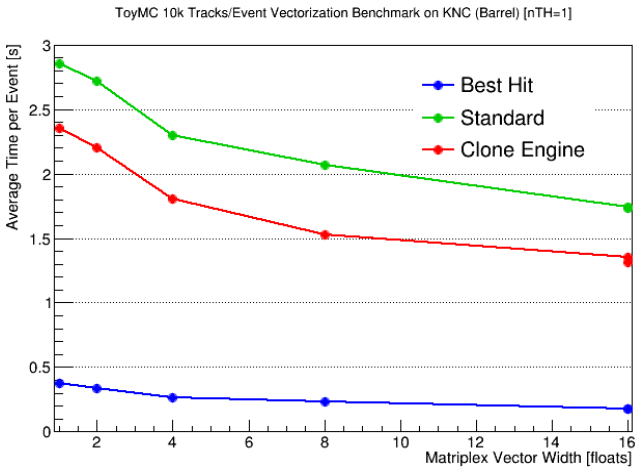
Track Building: Sandy Bridge and KNC

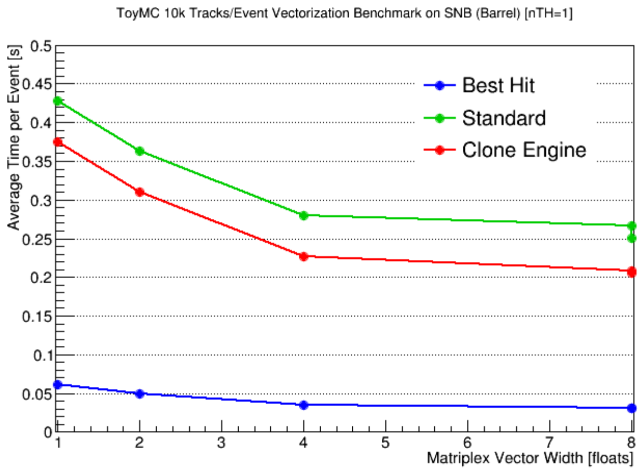
- Toy Monte Carlo experiment
  - Simplified geometry & simulated events
  - Similar trends in experiments with realistic geometry & CMSSW events
- Scaling tests with 3 building algorithms
  - Best Hit - less work, recovers fewer tracks (only one hit *saved* per layer, for each seed)
  - Standard & Clone Engine - combinatorial, penalized by branching & copying
- Two platforms tested
  - Sandy Bridge (SNB): 8-float vectors, 2x6 cores, 24 hyperthreads
  - Knights Corner (KNC): 16-float vectors, 60+1 cores, 240 HW threads
- Vectorization - speedup is limited in all methods
  - Faster by only 40-50% on both platforms
- Multithreading with Intel TBB - speedup is good
  - Clone Engine gives best overall results
  - With 24 SNB threads, CE speedup is ~13
  - With 120 KNC threads, CE speedup is ~65

KNC

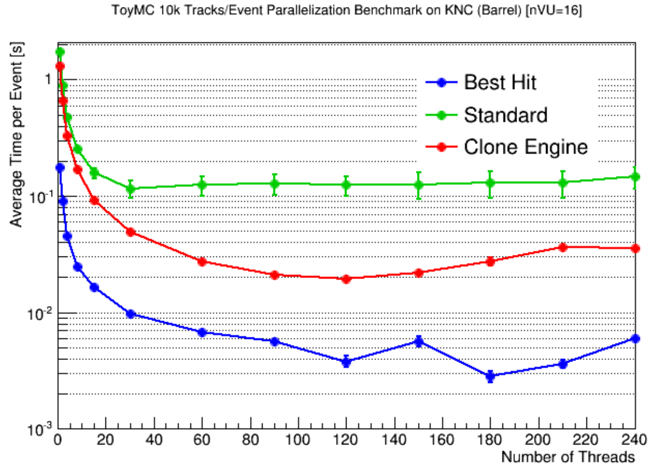
Sandy Bridge

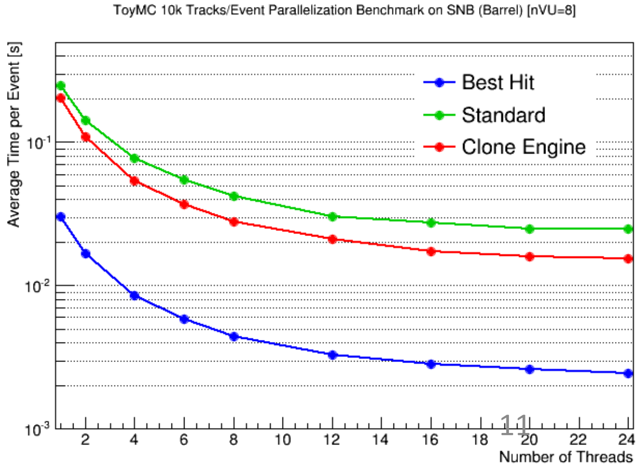
Vectorization





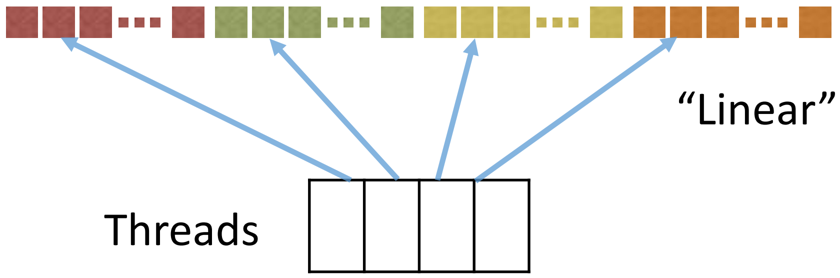
Multithreading

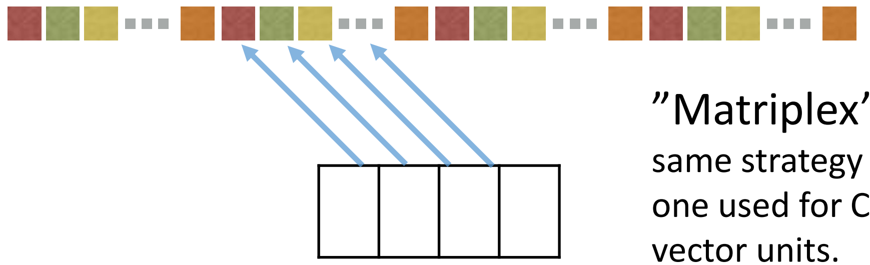




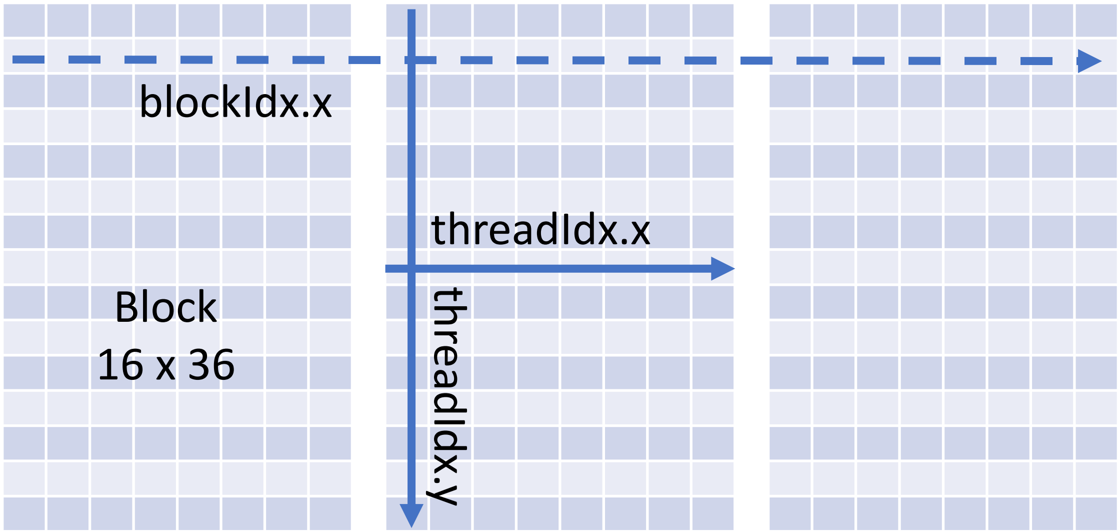
GPU: Finding a Suitable Memory Representation

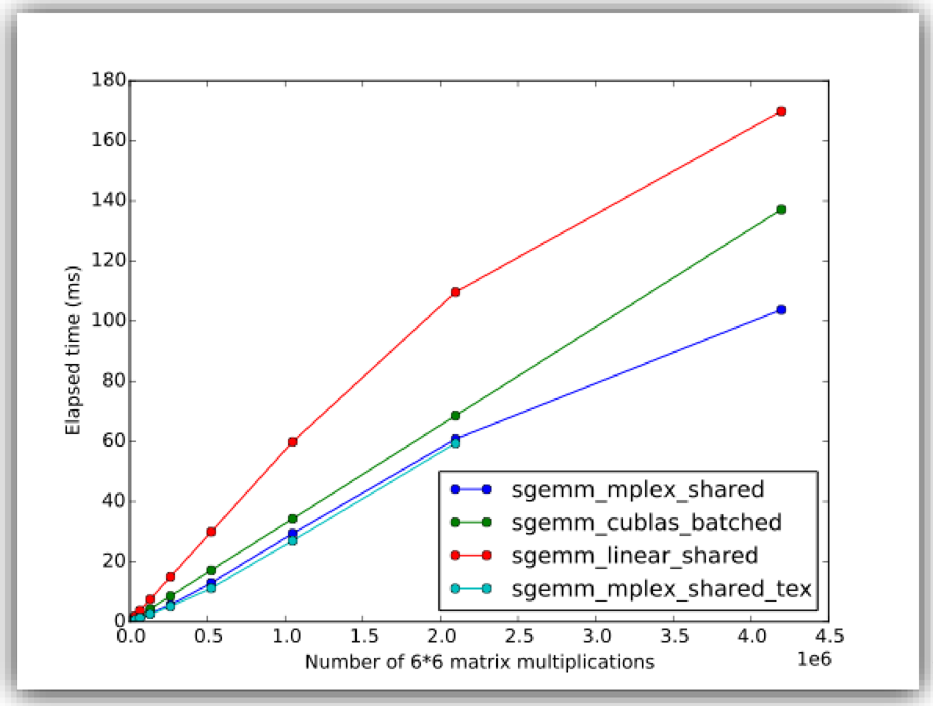
Memory Array





same strategy as the one used for CPUs’ vector units.





GPU Porting Strategy: An Incremental Approach

- Start with fitting:
  - Share a large number of routines with building
  - Simpler: less branching, indirections, ….
  - ➔Draw lessons along the way
- Gradually increase complexity
  - ”Best Hit”: (at most) 1 candidate per seed
    - New issue: Indirections
  - “Combinatorial”: multiple candidate per seed
    - New issue: Branching

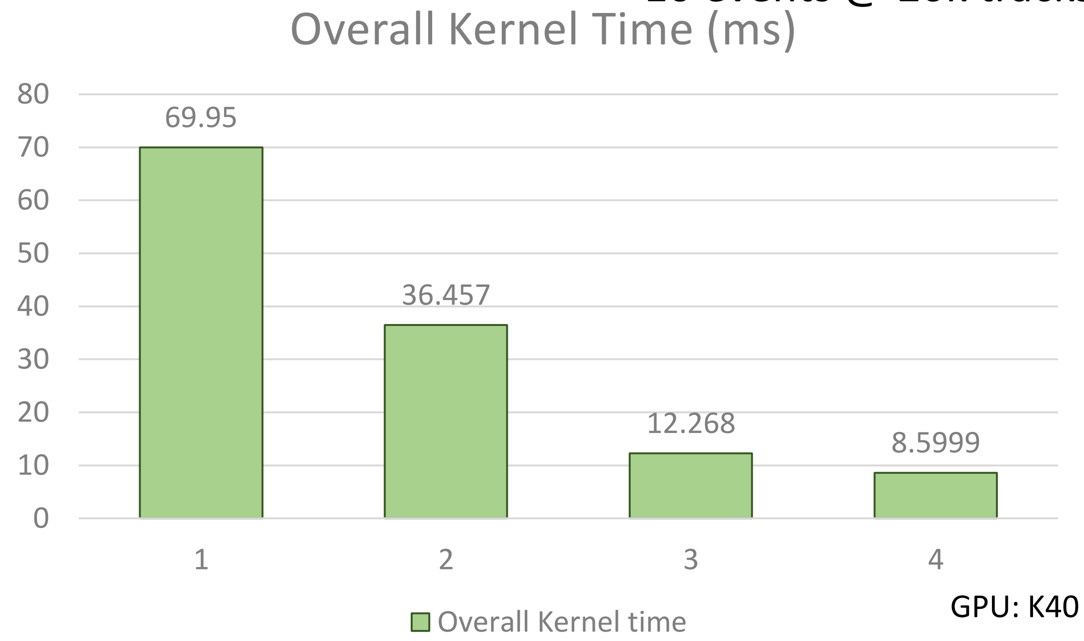
Fitting: Optimizing Individual Events

```
For each event:
    Reorganize tracks
    Transfer tracks
    For each layer:
        Reorganize hits
        Transfer hits
        Compute
        Transfer partial result
        back
```

GPU Propagation & Update Computations

CPU Reorganizing data to Matriplex Numerous indirections

10 events @ 20k tracks

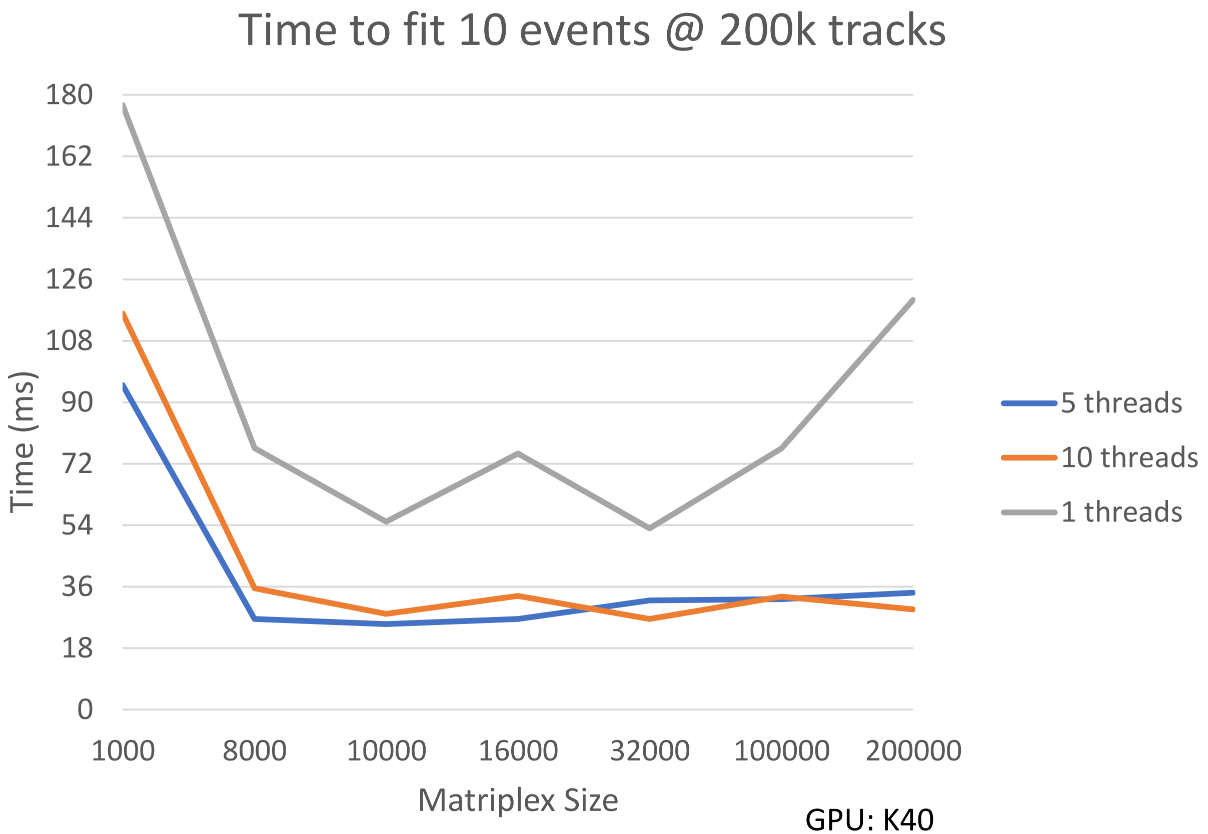


GPU: K40

1. Pre-optimization
2. Better data access: use read only-cache (const __restrict__)
3. Merging kernels (reducing launch overhead)
4. Use registers over shared memory

Fitting: Filling up the GPU

- Larger Matriplex size
  - Faster kernels
  - Longer reorganization
- Smaller Matriplex size
  - “Faster” reorganization
- Concurrent events, different streams
  - Individual kernel instances take longer
  - Overall time shorter
- Compromise:
  - Find Matriplex size so that time(reorg + transfer + kernel) is minimum



GPU: K40

Track Building: GPU Best Hit

- Parallelization: as in Track Fitting
  - Parallelization: 1 **GPU** thread per candidate

- Reorganizing on the CPU is not an option for building
  - Frequent reorganizations ➔ very small kernels
  - Numerous reorganizations ➔no more overlapping possible

- Data Structures
  - Matching CPU and GPU data structures to ease data transfers
    - Later reorganized as Matriplexes on the GPU
  - Static containers directly used on the GPU: Hits, Tracks, …
  - Object composition forces additional trick for classes at the top of the wrapping hierarchy
    - Keep arrays of sub-objects both on the host and on the device to be able to fill copy sub-objects from the CPU and access them from the GPU.
    - ➢ Data transfer overhead from transferring multiple smaller objects

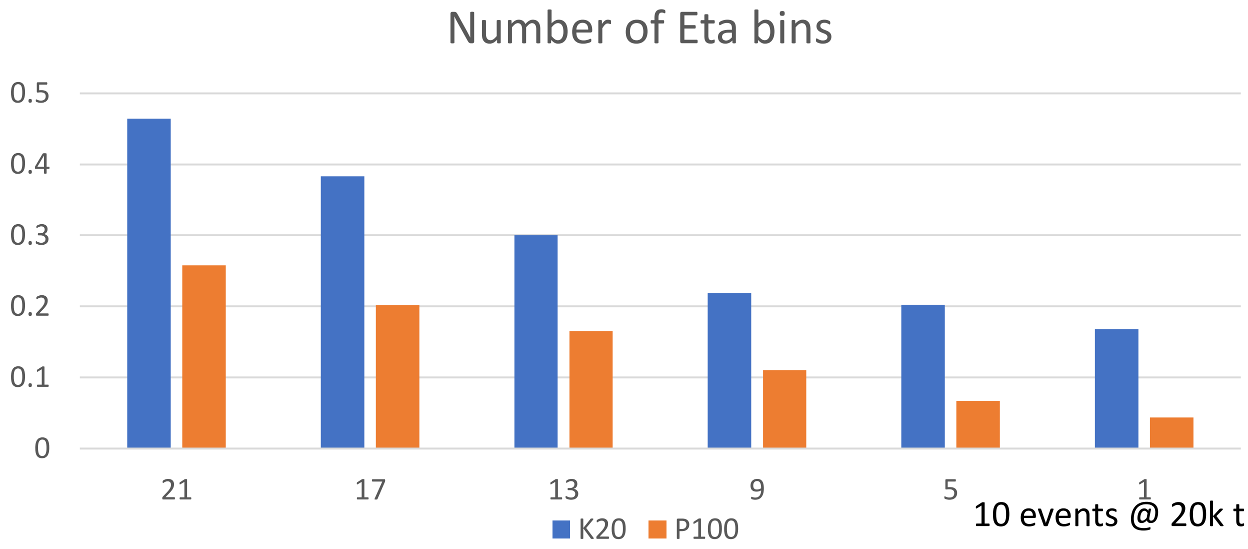
Track Building: Tuning Parameters

- Usual problem, find t_{min} that satisfies:
  - t_{min} = min f(a, b, c, d)

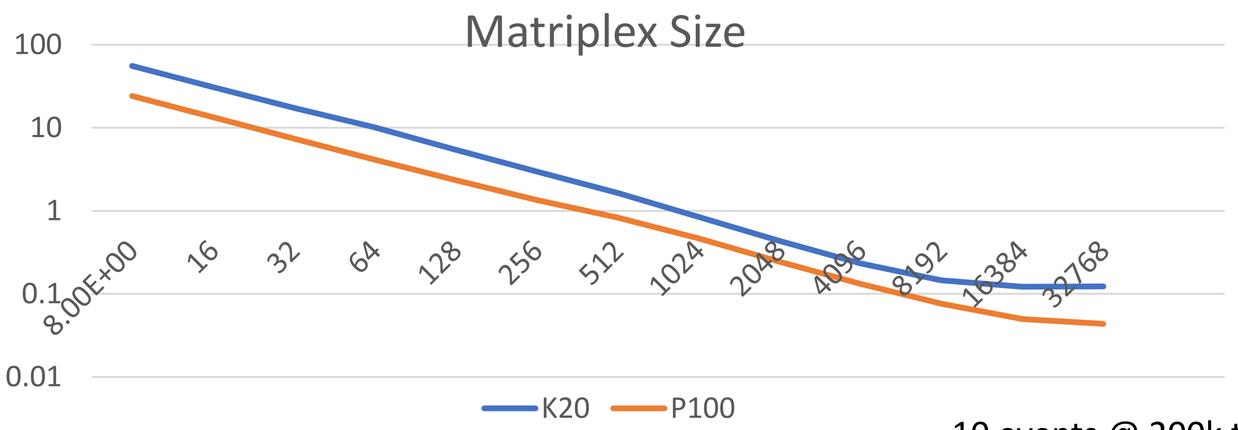
  - (a) Number of Eta bins [*]
  - (b) Threads per block
  - (c) Matriplex width
  - (d) Number of tracks per event

- ”Standard” insight
  - The GPU should be filled with enough threads
  - Even more so with newer GPUs

  ➔Computing multiple events concurrently is mandatory



10 events @ 20k tracks

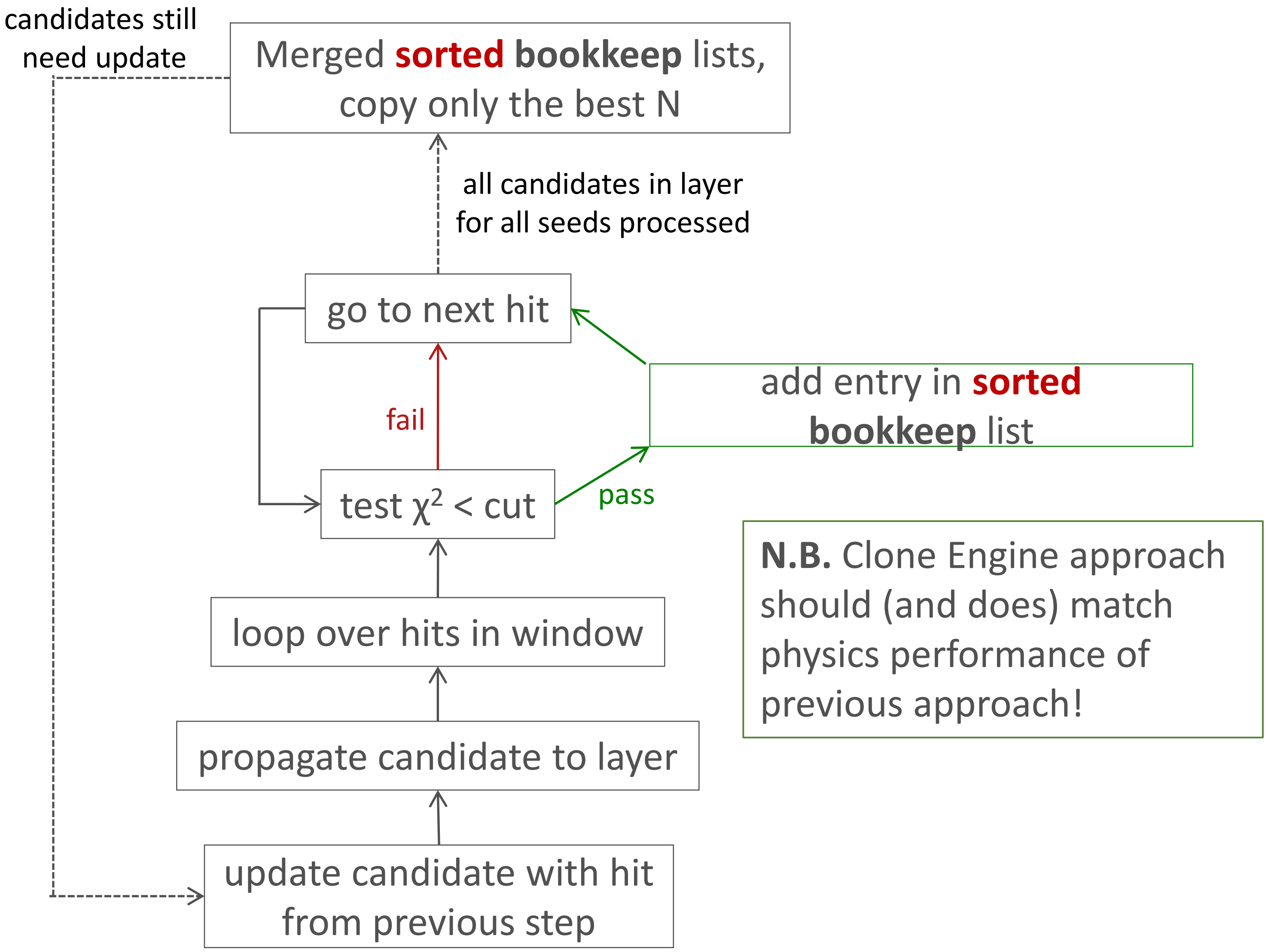


10 events @ 200k tracks

[*] Eta bins: Hits are binned by η to reduced the amount of hits that should be tried for a track

.

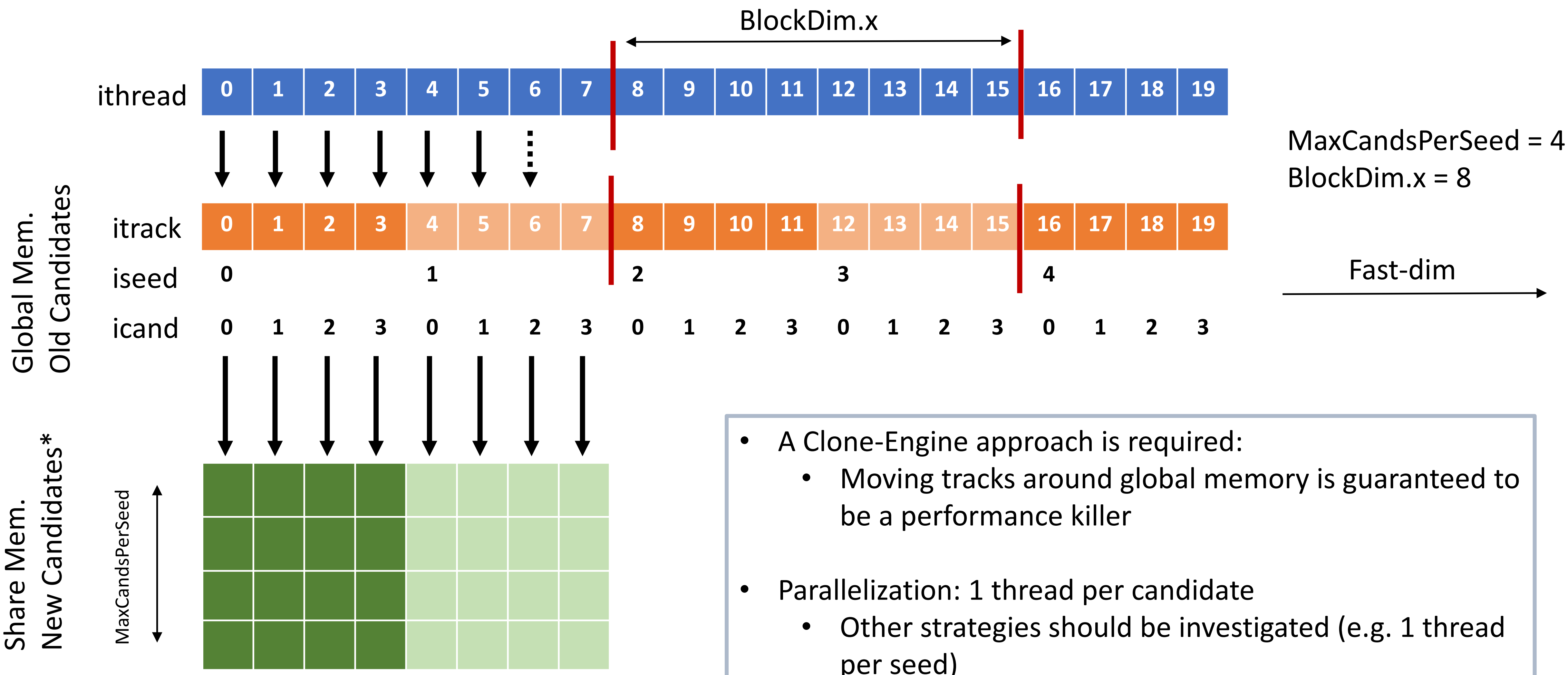
Building with Multiple Candidates: “GPU Clone Engine”



candidates still need update
Merged **sorted bookkeep** lists, copy only the best N
all candidates in layer for all seeds processed
go to next hit
add entry in **sorted bookkeep** list
fail
pass
test $\chi^2 <$ cut
loop over hits in window
propagate candidate to layer
update candidate with hit from previous step

N.B. Clone Engine approach should (and does) match physics performance of previous approach!

Building with Multiple Candidates



ithread	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
itrack	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
iseed	0				1				2				3				4			
icand	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3	0	1	2	3

MaxCandsPerSeed = 4
BlockDim.x = 8

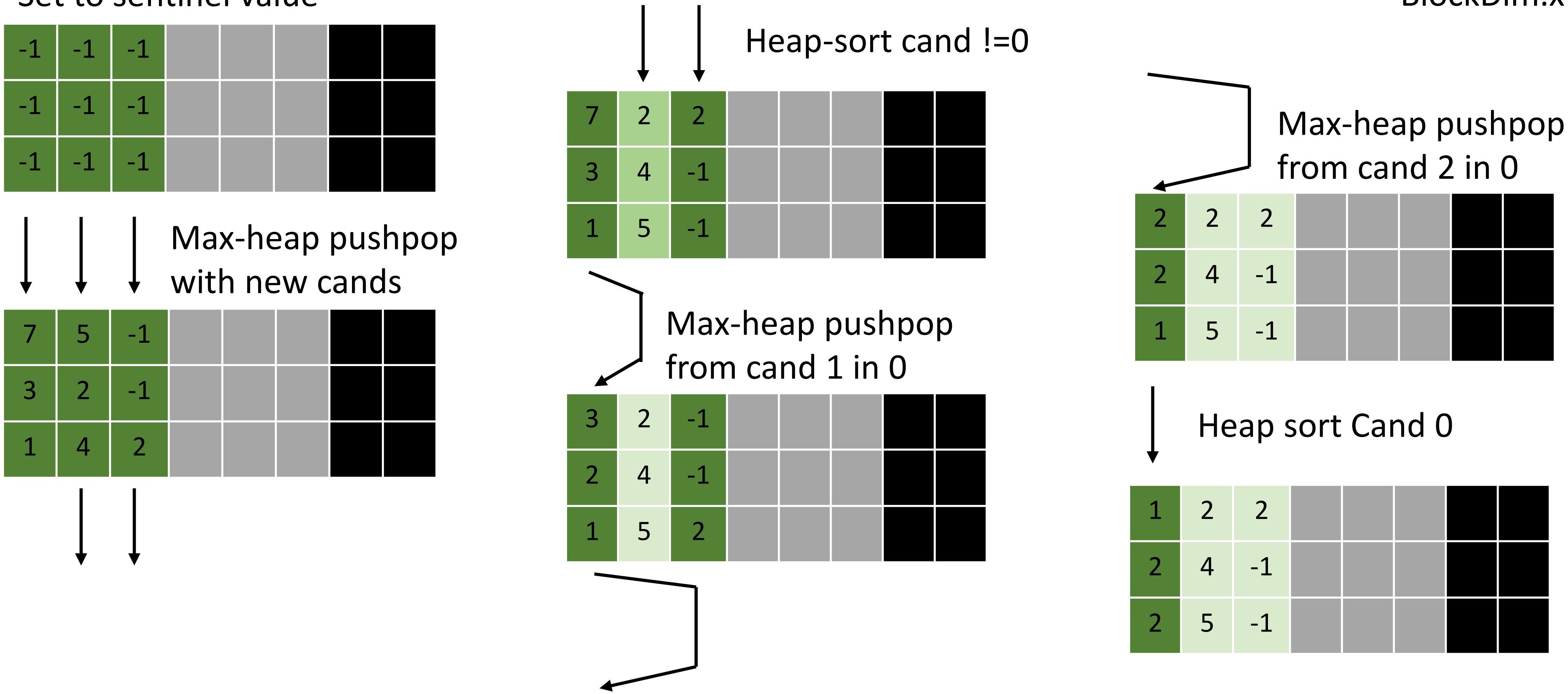
- A Clone-Engine approach is required:
  - Moving tracks around global memory is guaranteed to be a performance killer
- Parallelization: 1 thread per candidate
  - Other strategies should be investigated (e.g. 1 thread per seed)

*Potential next-layer candidates, after adding an acceptable hit from the current layer

Sifting a Seed's New Candidates in Shared Memory

MaxCandsPerSeed = 3
BlockDim.x = 8

Set to sentinel value



-1	-1	-1					
-1	-1	-1					
-1	-1	-1					

Max-heap pushpop with new cands

7	5	-1					
3	2	-1					
1	4	2					

Heap-sort cand !=0

7	2	2					
3	4	-1					
1	5	-1					

Max-heap pushpop from cand 1 in 0

3	2	-1					
2	4	-1					
1	5	2					

Max-heap pushpop from cand 2 in 0

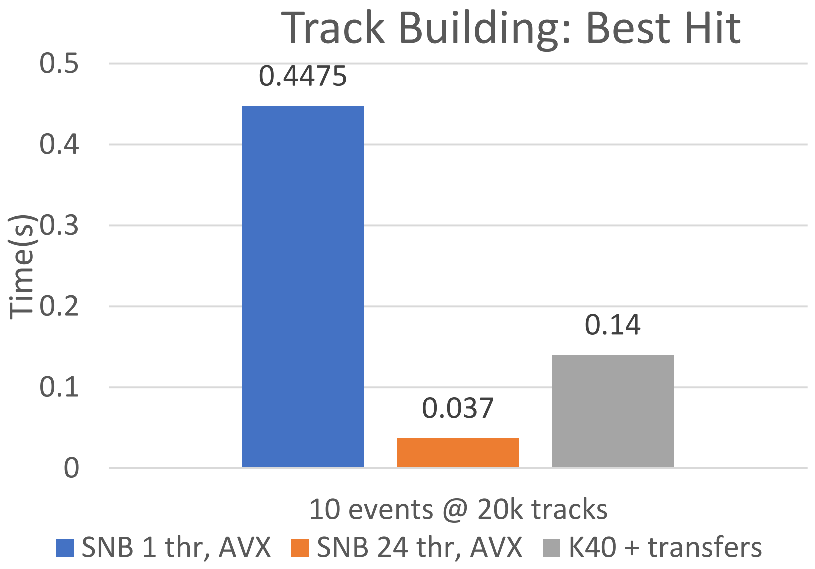
2	2	2					
2	4	-1					
1	5	-1					

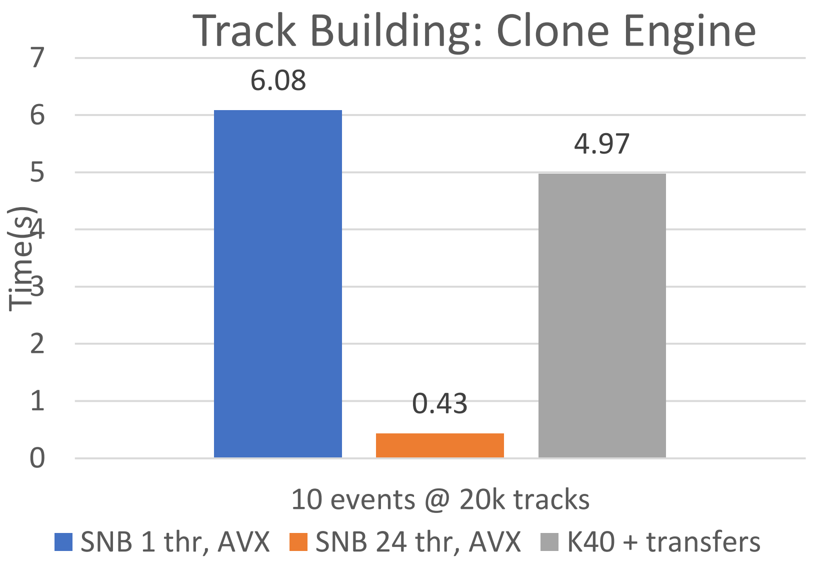
Heap sort Cand 0

1	2	2					
2	4	-1					
2	5	-1					

Notes: Possible optimization using a binary tree approach to sift
The integer in each box represents the **chi-squared** that results from adding a given hit

Track Building: Initial Performance





- 20K tracks per event is not enough to give good performance
- Need to increase the number of events concurrently fed to the GPU by using different streams

- Too many synchronizations
- Sorting’s branch predictions
- Idling threads when number of candidates per seed is not maximum
- Transfer account for 46% of the time

Avoiding Code Duplication

- Keeping two code bases in sync is complicated
- Ability to directly address the architecture
  - At least for this kind of study

- Core routines are very similar in C++ and CUDA

➔Template interface

  - Overloaded operators ([], (,,,))
  - Allow for the same memory accesses

➔Separation between ”logic” and work decomposition

  - C++ ”`for`” loops vs. CUDA “`if (< guard)`”

Track Building on GPU: Improvements and Next Steps

- Stream concurrent events to the GPU
  - Already in place but,
  - Event set-up needs to be moved outside the parallel loop

- Alternative strategies for the clone-engine approach
  - One thread per seed
  - Adaptive strategy depending on the number of candidates per seed

- Move onto newer Pascal GPUs
  - Profiling with Instruction Level GPUs (>= Maxwell)
  - Synchronize only relevant threads, not the full block (>= Pascal)
    - __syncthreads -> sync(…);

Conclusion & Perspectives

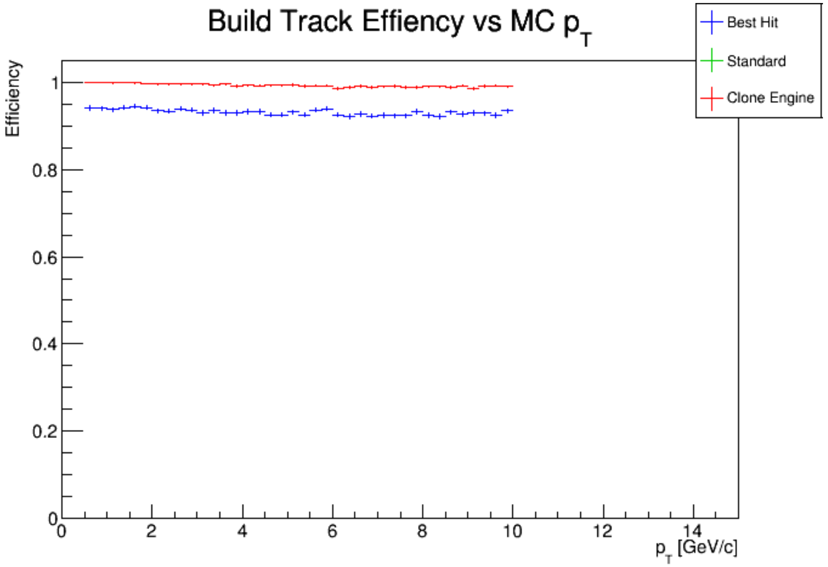
- Track fitting and track building show good performance for both **vectorization** and **parallelization** on **x86** processors (SNB & KNC)

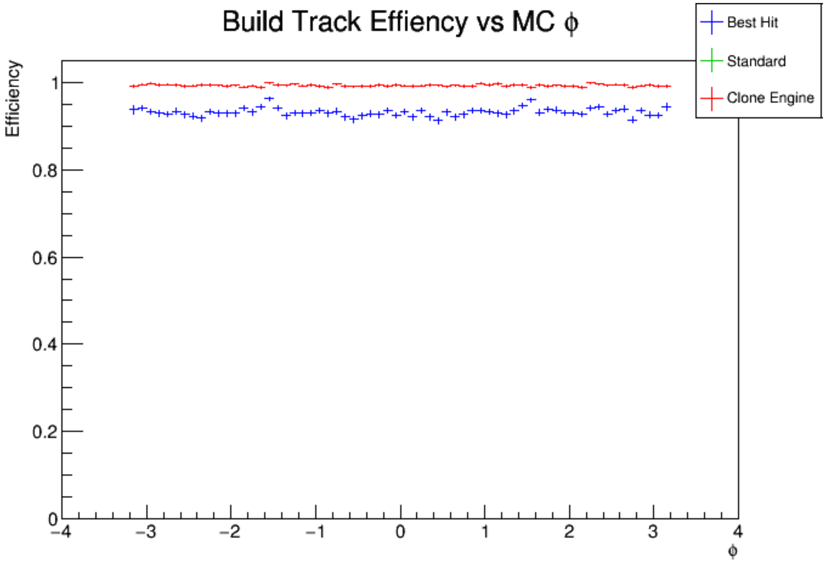
- GPU performances are still behind, particularly in term of $/event
  - Newer GPUs should alleviate some of the issues
  - Better filling through concurrent streams of events seems crucial

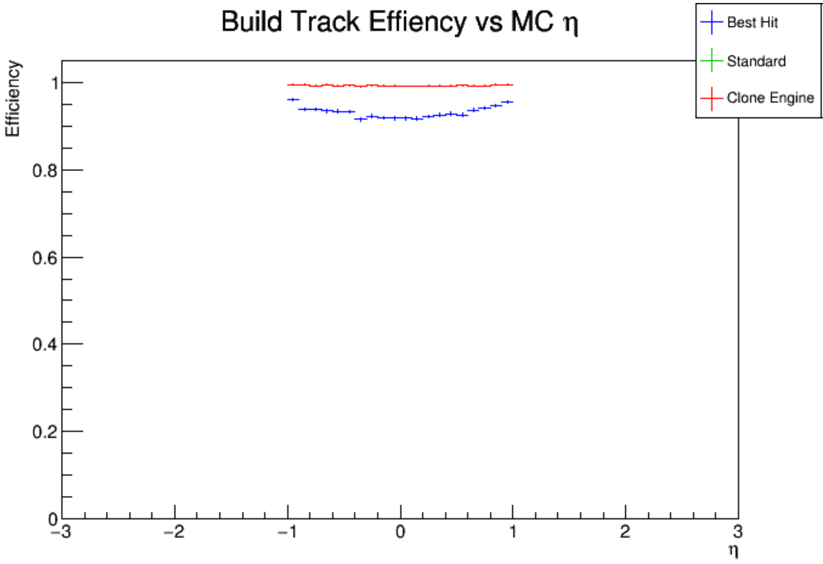
- Lessons learned on one architecture are often valuable to algorithm development on the other one

Backup

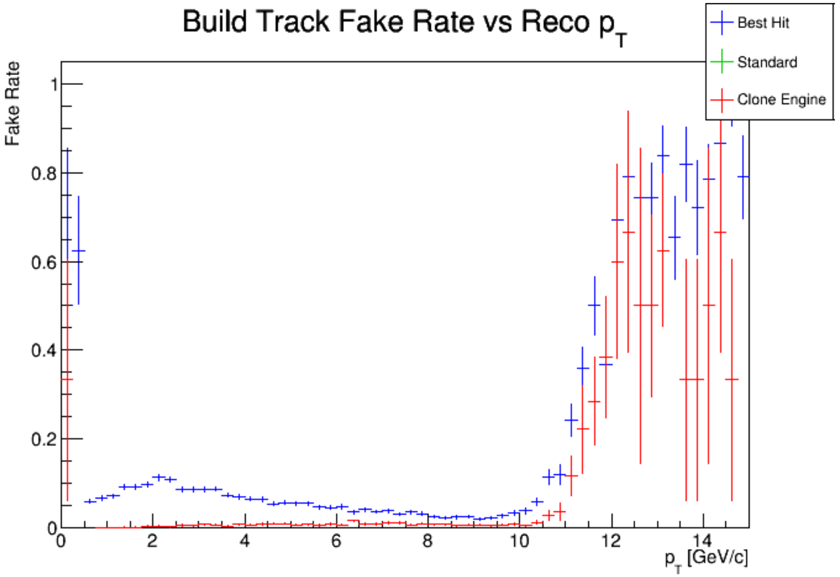
Track Building: Physics Performance – Efficiency

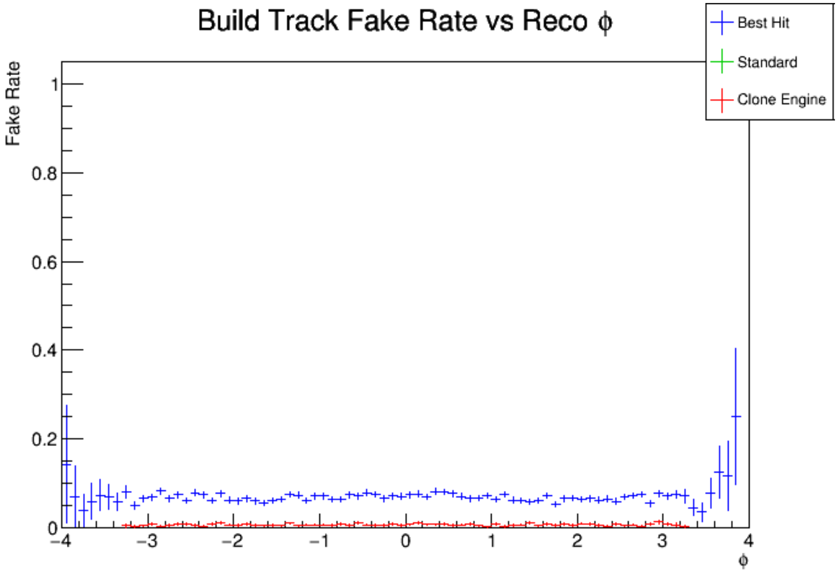

Build Track Effiency vs MC p_T

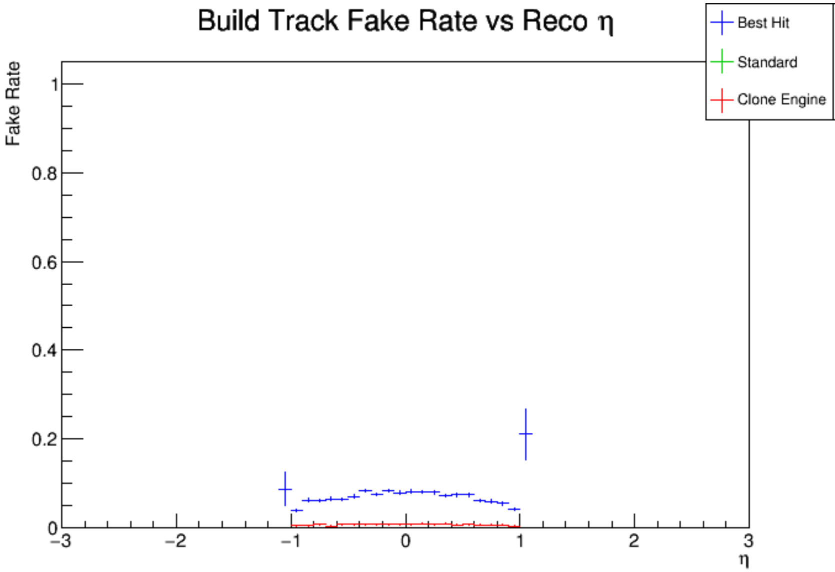

Build Track Effiency vs MC ϕ


Build Track Effiency vs MC η

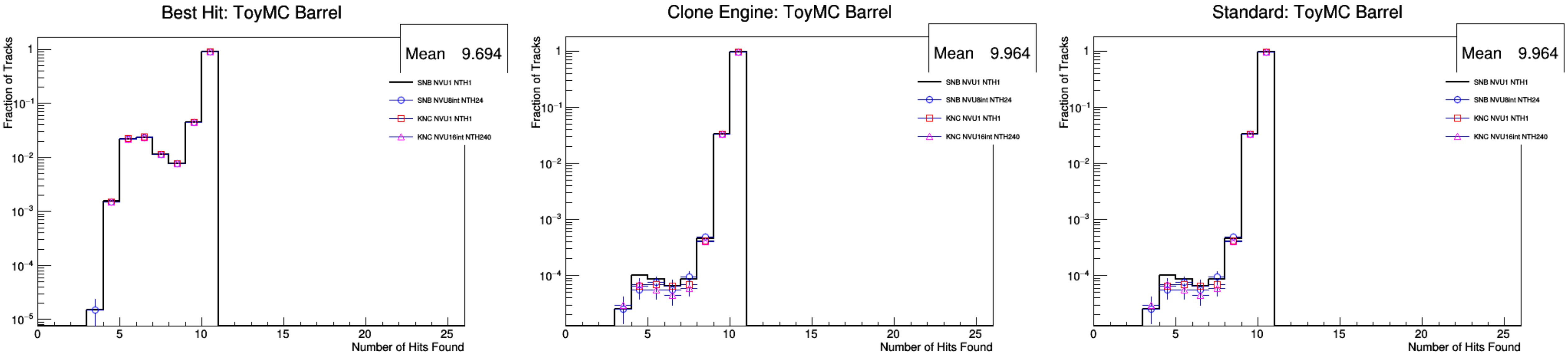
Track Building: Physics Performance – Fake Rate



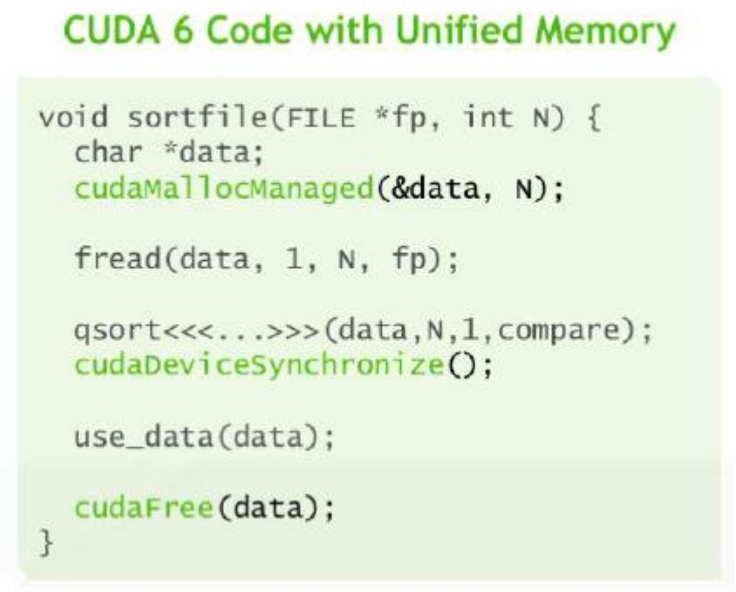




Track Building: Physics Performance
Number of Hits per Track

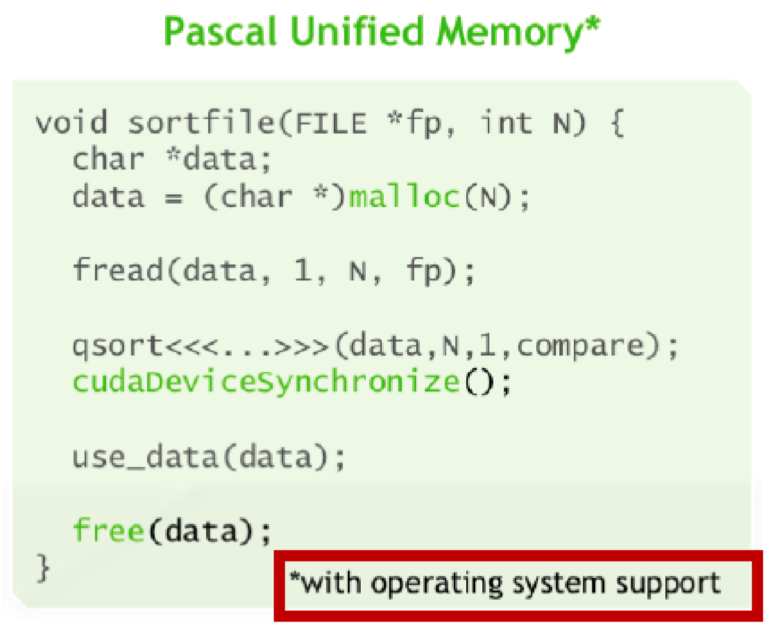


Pascal Managed Memory



CUDA 6 Code with Unified Memory

```
void sortfile(FILE *fp, int N) {
  char *data;
  cudaMallocManaged(&data, N);

  fread(data, 1, N, fp);

  qsort<<<...>>>(data,N,1,compare);
  cudaDeviceSynchronize();

  use_data(data);

  cudaFree(data);
}
```



Pascal Unified Memory*

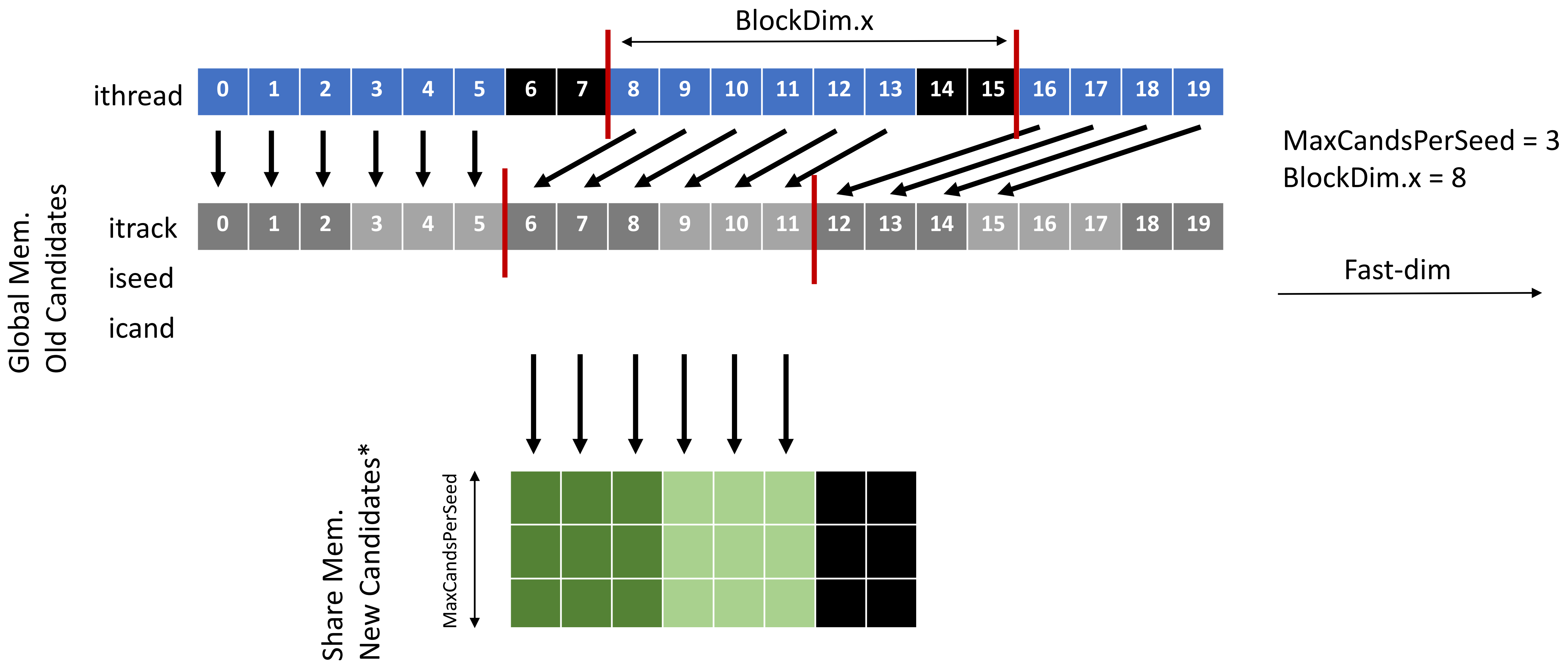
```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);

  fread(data, 1, N, fp);

  qsort<<<...>>>(data,N,1,compare);
  cudaDeviceSynchronize();

  use_data(data);

  free(data);
}
```

*with operating system support

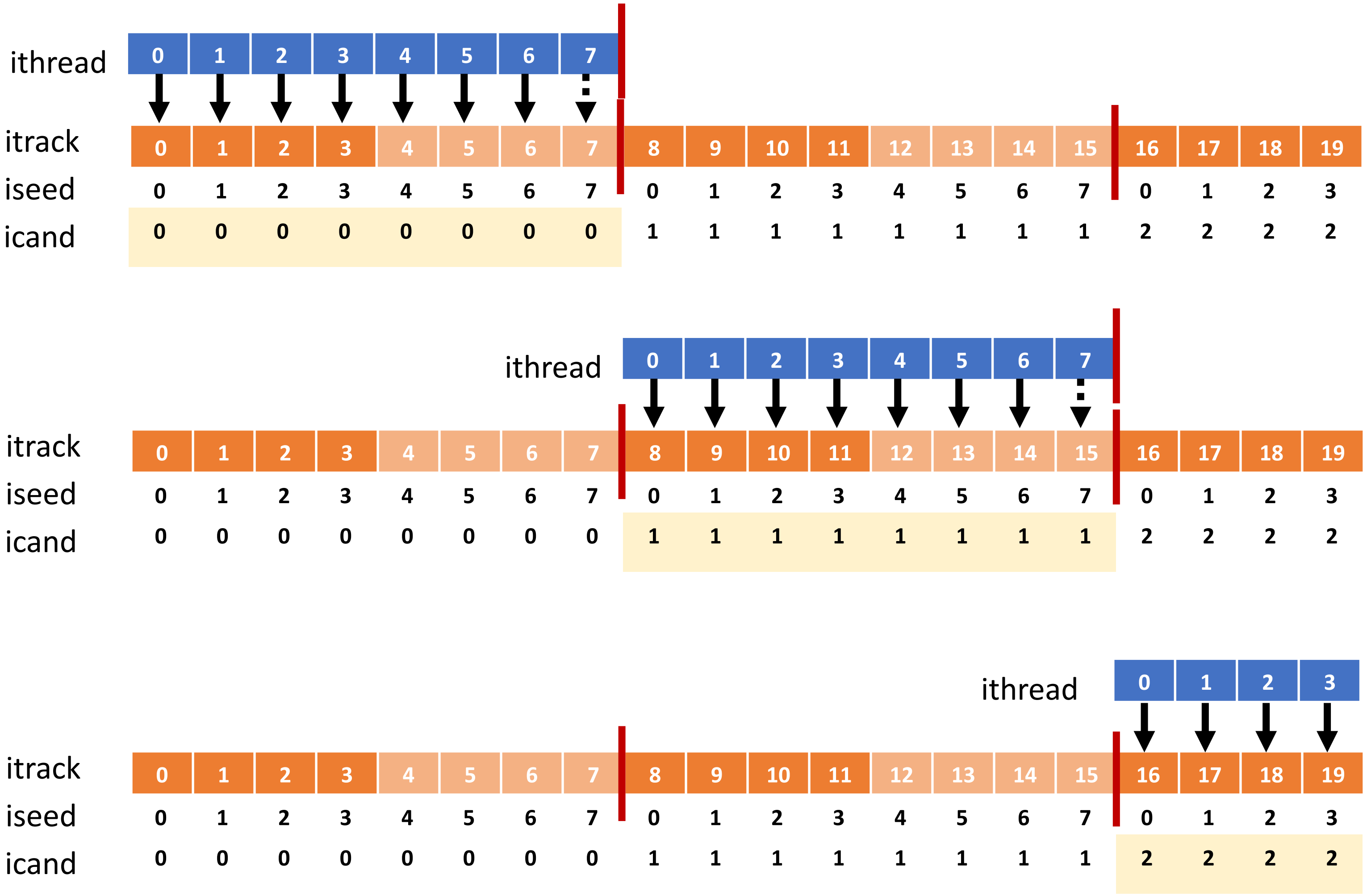
STL?

- But it is not just for beginners; Unified Memory also makes complex data structures and C++ classes much easier to use on the GPU. On systems that support Unified Memory with the default system allocator, any hierarchical or nested data structure can automatically be accessed from any processor in the system. With GP100, applications can operate out-of-core on data sets that are larger than the total memory size of the system.

Mapping: Threads, Tracks, CandList



Clone Engine: Different Strategy



ithread	0	1	2	3	4	5	6	7												
itrack	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
iseed	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3
icand	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	2	2	2	2

ithread									0	1	2	3	4	5	6	7				
itrack	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
iseed	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3
icand	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	2	2	2	2

ithread																	0	1	2	3
itrack	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
iseed	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3
icand	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	2	2	2	2

- One thread per seed
  - Less idling threads due to low numbers of candidates per seed
  - Breaks the Matriplex aligned memory accesses
    - Unless 32 candidates per seed
    - Or significant padding
  - Less threads per events

Avoiding Code Duplication
…while keeping the C++ code clean

- STL is still not available in CUDA code
- C-arrays (T*) are the main container in CUDA code (even if encapsulated)

- E.g. std::vector are hard to adapt
  - Resize(), push_back(),…

➔Allocate larger arrays

➔Many cudaMemcpys to transfer complex, irregular data structure
  - E.g. vector<vector<T>>

Realistic Geometry with CMSSW events (x86 only)

- Single muon gun
- t-tbar + pileup, where the mean pileup per event is 35
- Phase0 geometry
- Efficiency and nHits/track for CMS events
    - comparable to CMS for this particular set of seeds/tracks