Automatic FFT Kernel Generation for CUDA GPUs.

Akira Nukada

Tokyo Institute of Technology

FFT (Fast Fourier Transform)

FFT is a fast algorithm to compute DFT (Discrete Fourier Transform).

$$X'(k) = \sum_{j=0}^{N-1} X(j) e^{-2\pi i jk/N}$$

When the input size N can be factorized into M and L, N-point FFT is replaced by L x M-point FFTs, M x L-point FFT, and multiplications by twiddle factors.

$$X'(k_0 + k_1 L) = \sum_{j_0=0}^{M-1} e^{-2\pi i j_0 k_1 / M} \; e^{-2\pi i j_0 k_0 / N} \sum_{j_1=0}^{L-1} X(j_0 + j_1 M) e^{-2\pi i j_1 k_0 / L}$$

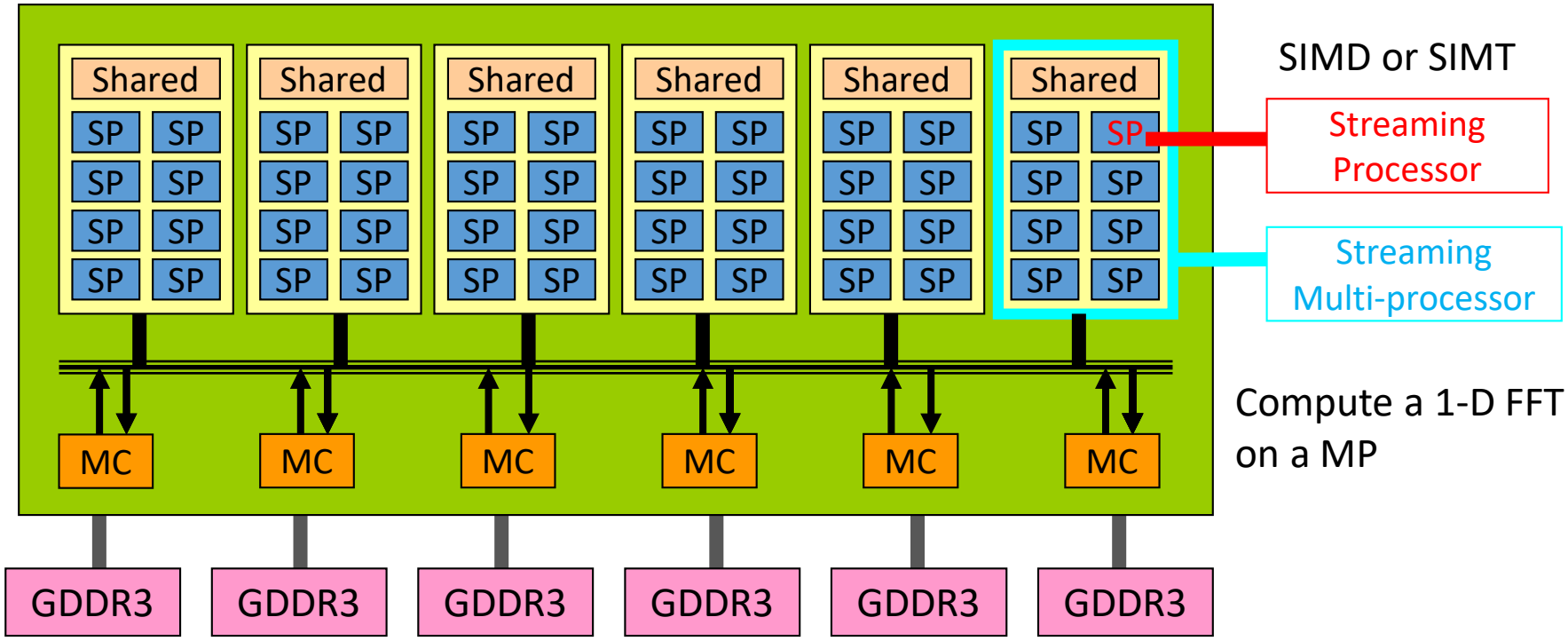
twiddle factors

L-point FFT

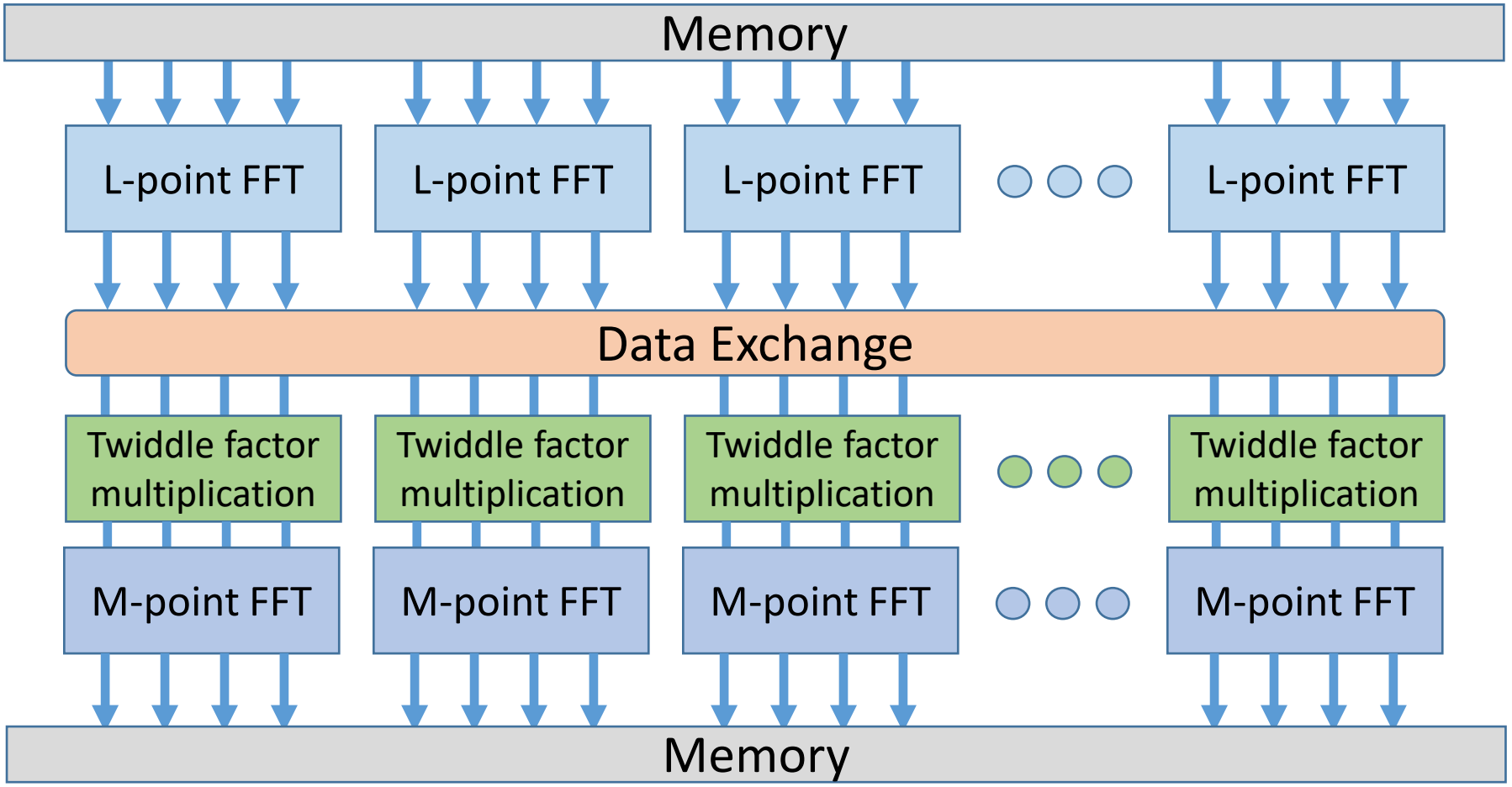
M-point FFT

NVIDIA CUDA GPU Architecture

Target: computing batched 1-D FFTs using GPU.



Fine-grain parallel computation of FFT



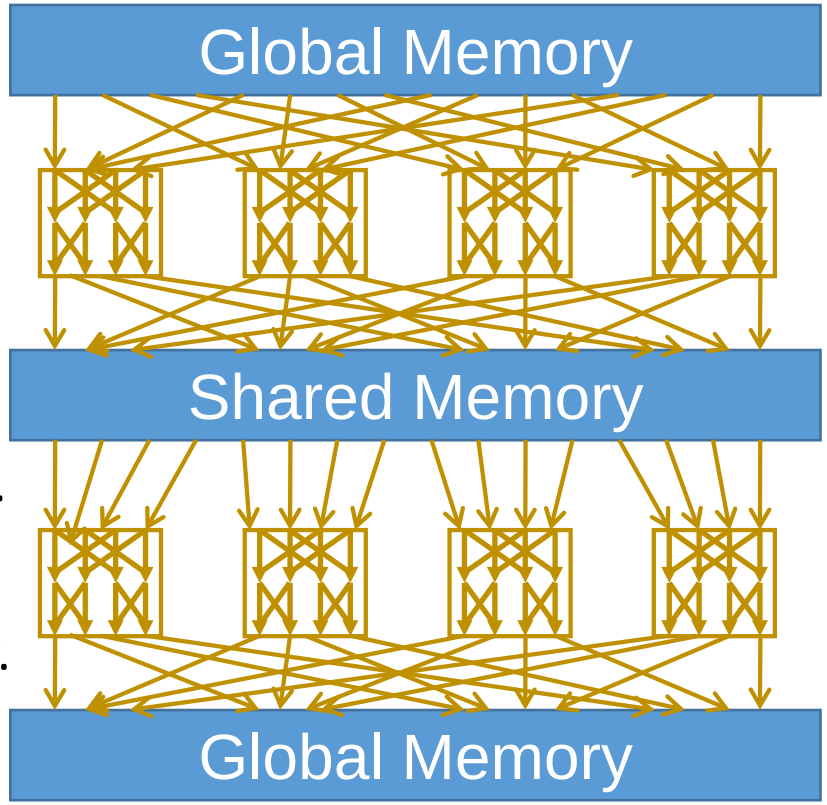
1-D FFT on CUDA GPUs

Data is read from global memory.

Many threads simultaneously compute small FFTs using registers.

Threads exchange data using shared memory.

Finally, data is written back to global memory.



Three CUDA FFTs in 2008 based on this fine-grain parallel implementations

(1) Ours (2) N. Govindaraju, et. al. (3) V. Volkov, et. al.

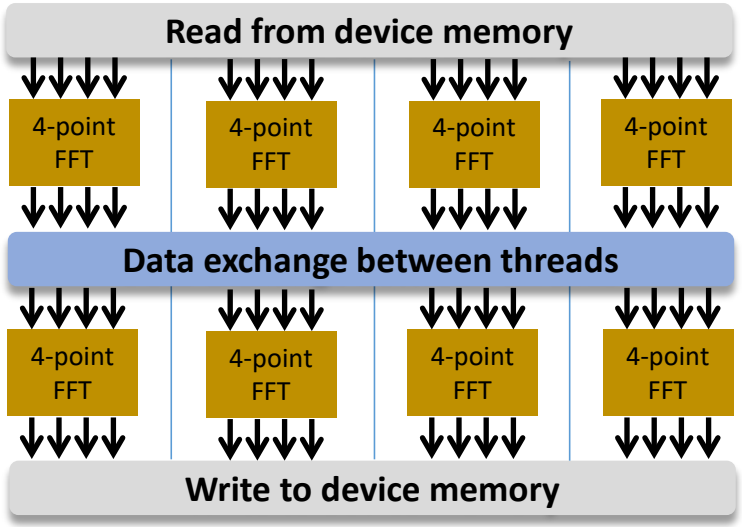
Twiddle factor multiplication in CUDA FFT

Twiddle Factors are triangular functions,
and thread-dependent value.
In CUDA, they should come from one of
(1) registers.
(2) table on constant (cache) memory.
(3) table on texture (cache) memory.
(4) table on shared memory.
(5) calculate using SFU each time.
We selected ‘texture plan’ to reduce the
number of instructions and registers.

Fast Fourier transform on AMD GPUs

Implementation with RADEON / OpenCL is similar to NVIDIA / CUDA

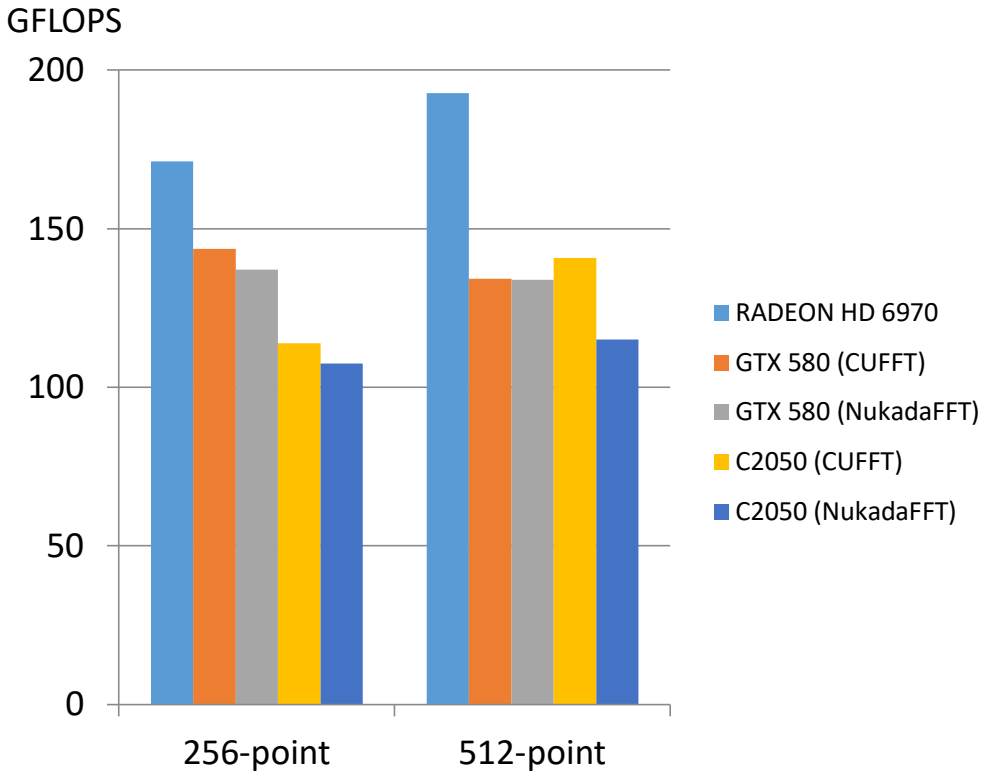
- Each thread computes a small FFT
- Data exchange between threads
  - Via shared memory (CUDA)
  - Via local memory (OpenCL)
- Twiddle factor (cos&sin) table
  - On texture memory (CUDA)
  - On constant memory (OpenCL)



Example of 16-point FFT using 4 threads

Performance comparison with NVIDIA GPUs

- GeForce and Tesla
  - Intel Core i7 CPU
  - X58 Express Chipset
  - CUDA 4.0
  - CUFFT library 4.0, or NukadaFFT
- RADEON
  - AMD Phenom 9500 CPU
  - AMD APP SDK 2.4
  - Custom FFT code in OpenCL



Bottle-neck & efficiency | ratio to theoretical peak

- DP performance
  - Bottle-neck on GeForce
- Memory access efficiency
  - Double-complex (double2) data
  - Good for RADEON, Bad for GeForce

	AMD RADEON HD 6970	NVIDIA GeForce GTX 580	NVIDIA Tesla C2050
Peak DP performance	675 GFLOPS	197GFLOPS	515GFLOPS
Achieved performance	171GFLOPS (25.3%)	144GFLOPS (73.1%)	114GFLOPS (22.1%)
Peak Memory B/W	176GB/s	192GB/s	144GB/s (128GB/s*9/8)
Achieved B/W	137GB/s (77.8%)	115GB/s (59.9%)	91GB/s (71.1%)
Bottle-neck	Memory	DP perf.	Memory

Number of floating point operations

$5N \log_2 N$ is *pseudo* number of FP operations

Powers-of-two FFT

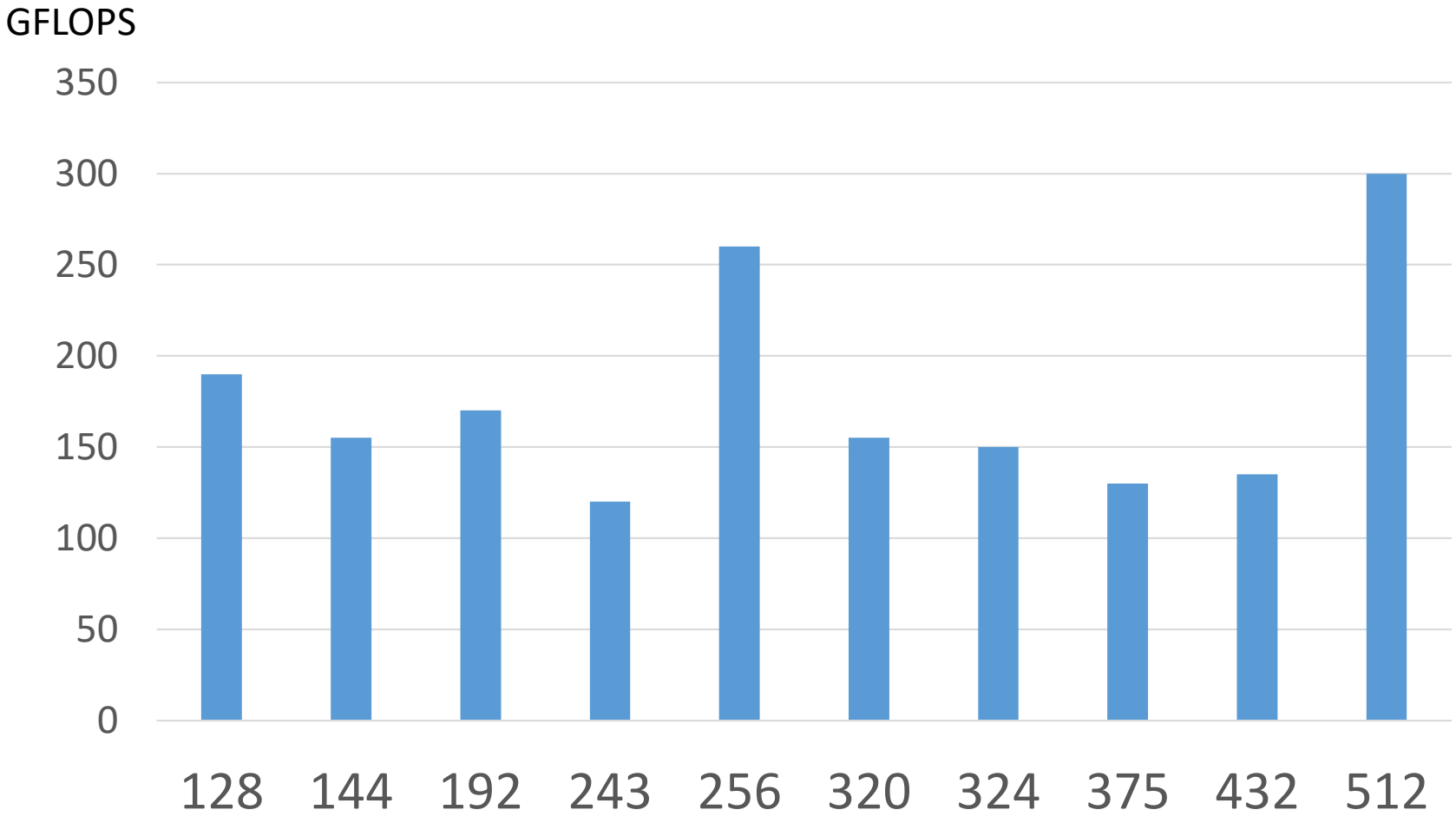
- Large number of FPADD/FPSUB ops.
- Low ratio of FPMAD combination

- RADEON GPU architecture can execute one of the following instructions in a cycle
  - 2 FPADD/FPSUB
  - 1 FPMUL
  - 1 FPMAD

Real number of FP ops. and FP instructions

	256-point	512-point
$5N \log_2 N$	10,240	23,040
ADD/SUB ops.	4,672	11,776
MUL ops.	2,304	4,352
FPADD/FPSUB	3,520	9,984
FPMUL	1,152	2,560
FPMAD	1,152	1,792
Min. FP cycle (AMD)	4,064	9,344
Min. FP cycle (NVIDIA)	5,824	14,336

Performance of 1-D FFT (GTX280)



Auto-Tuning parameters of FFT on CUDA GPUs

Many varieties in computing environment:

- Generation of GPU (G8X/G9X, GT2XX)
  - # of registers per SM
- # of cores, clock frequency (shader, memory)
- Compiler version (optimization of PTXJIT)
  - Actual register usage is determined by compiler.
- Driver version (CUDA & display driver)

Tuning Parameters

(1) Selection of radices of FFT kernels

- combination & ordering

ex) 240-point => 4, 4, 3, 5

In CPUs, this parameter determines
# of floating-point ops
# of load/store to cache memory

In GPUs, this parameter also determines
# of threads per thread block
In above example, radix 3 is the smallest.
Then, # of threads is set to 80 (240/3).

Tuning Parameters

(1) Selection of radices of FFT kernels

- combination & ordering

ex) 240-point => 4, 4, 3, 5

(2) Selection of number of threads (generic for GPU)

Sufficient thread blocks to exploit memory bandwidth

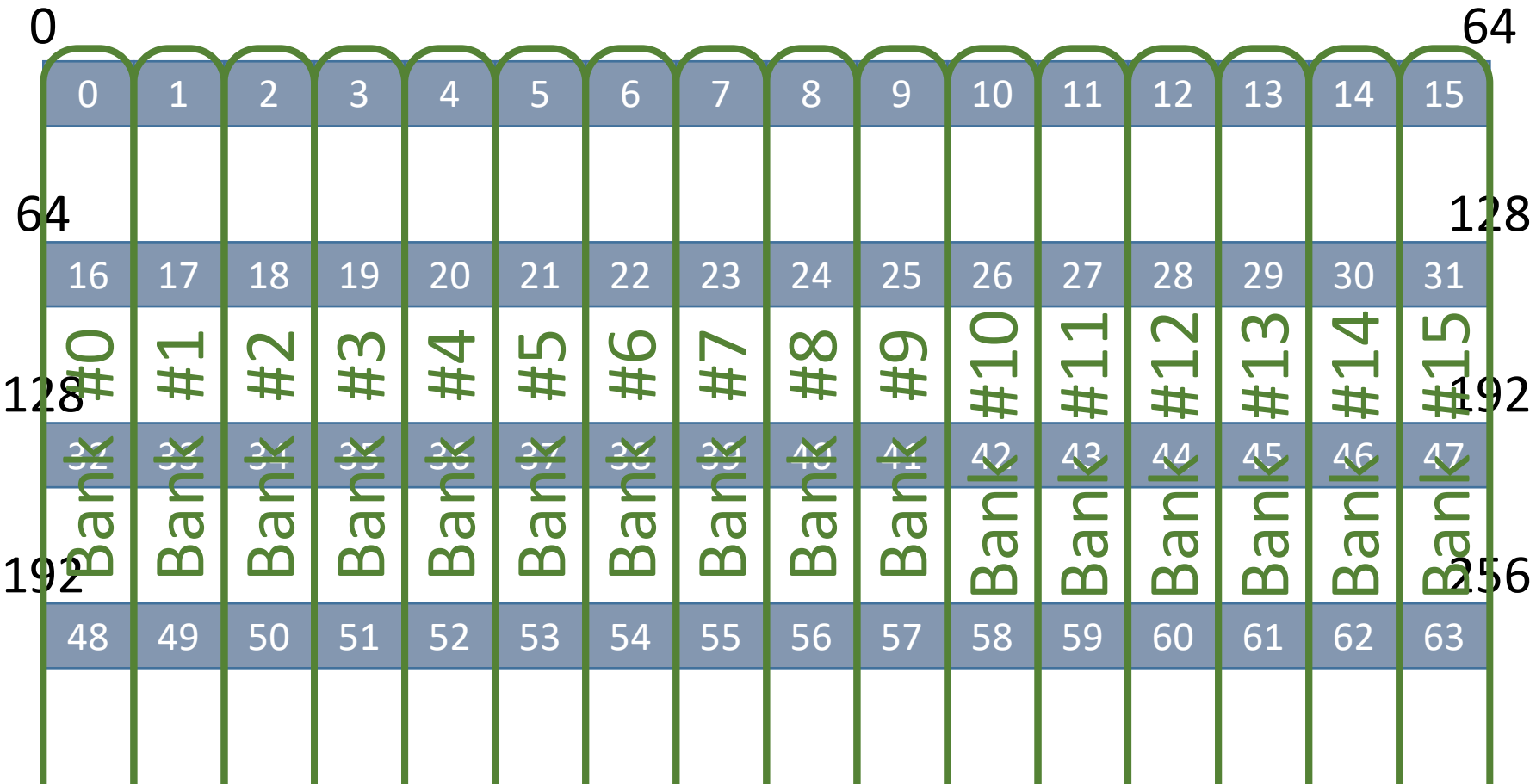
(3) Avoid bank conflicts on shared memory (CUDA & FFT)

Insert padding in a specific pattern/rule.

Shared memory access

- as fast as registers.
- consists of 16 banks of 32-bit.

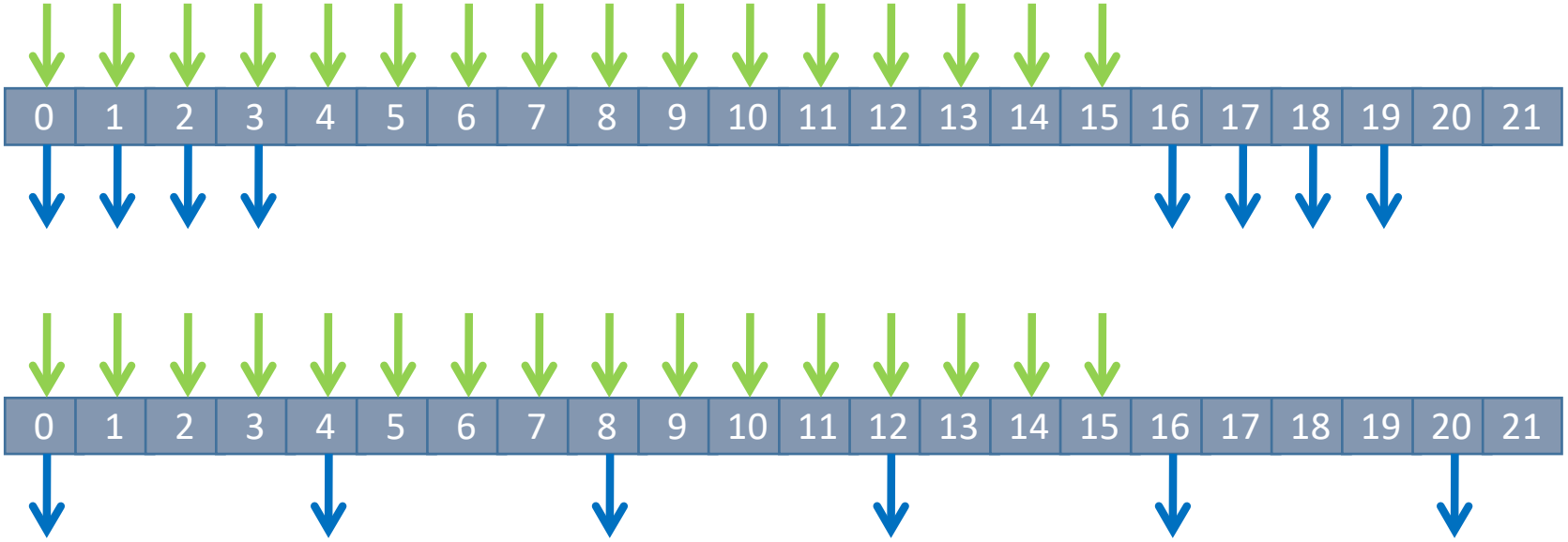
Up to 16 threads can access simultaneously, if there are no bank conflicts.



Shared memory access pattern in FFT

Implementation based on Stockham auto-sort algorithm.

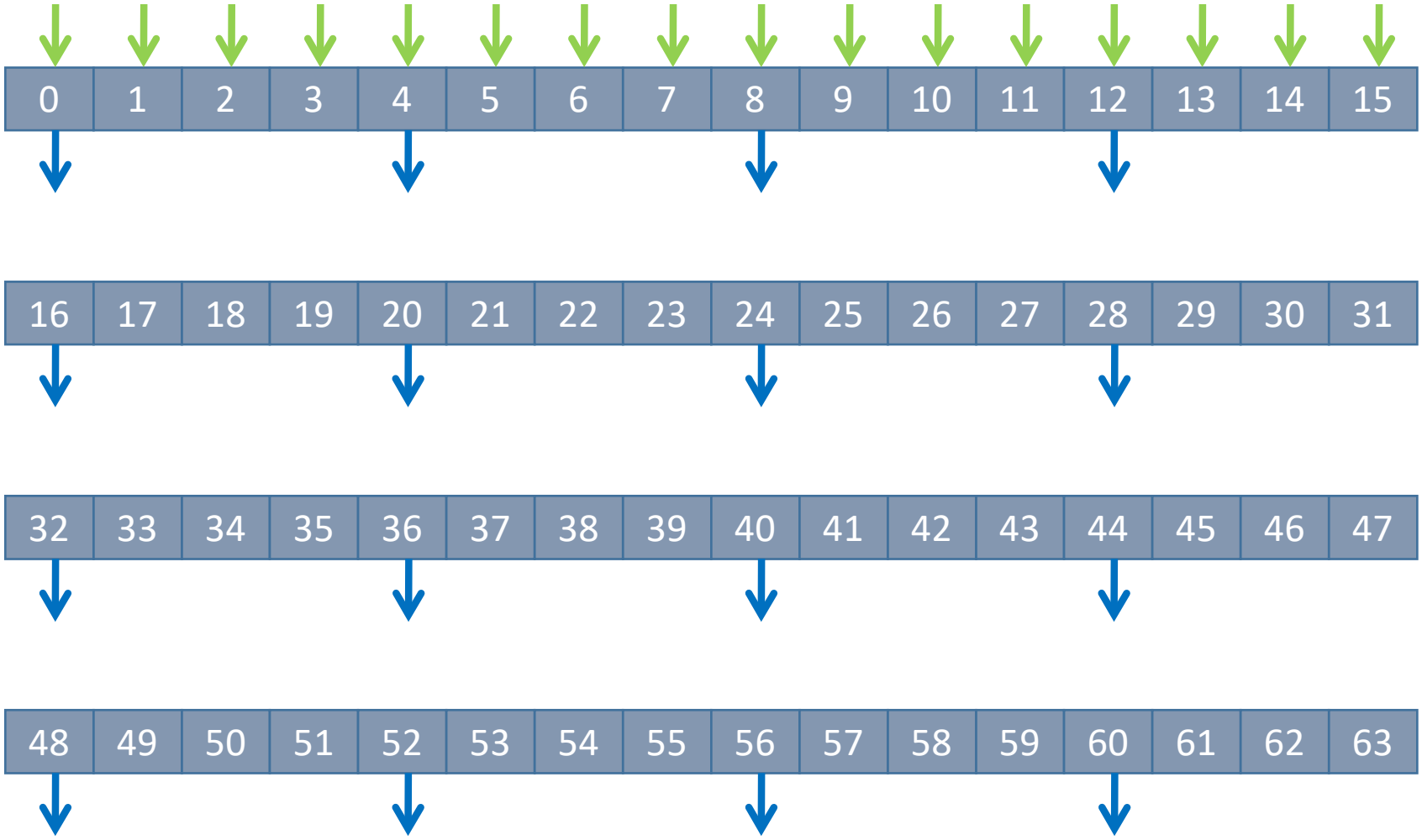
64-point FFT using 16 threads is described as follows.



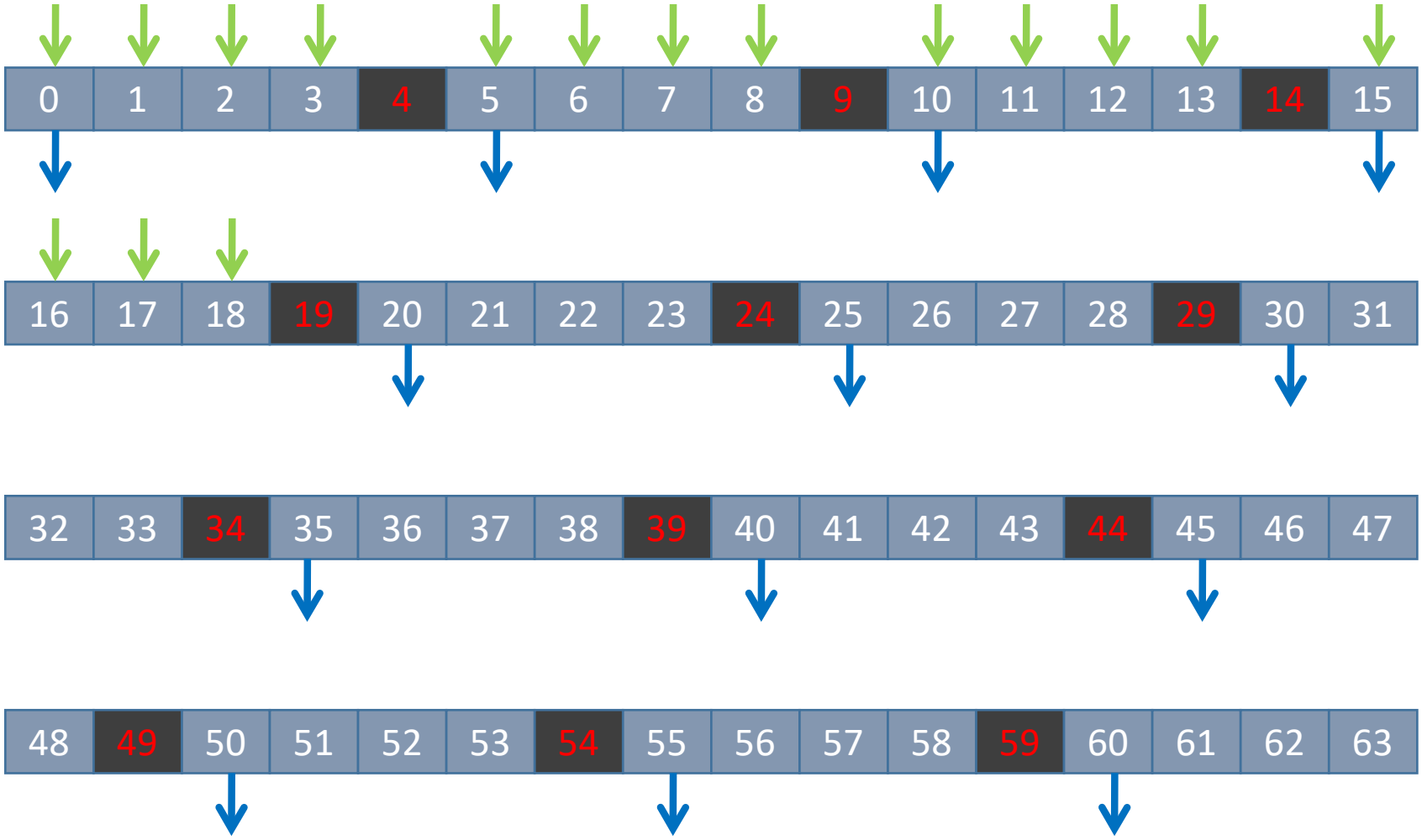
No bank conflicts in WRITEs

READs may cause some bank conflicts

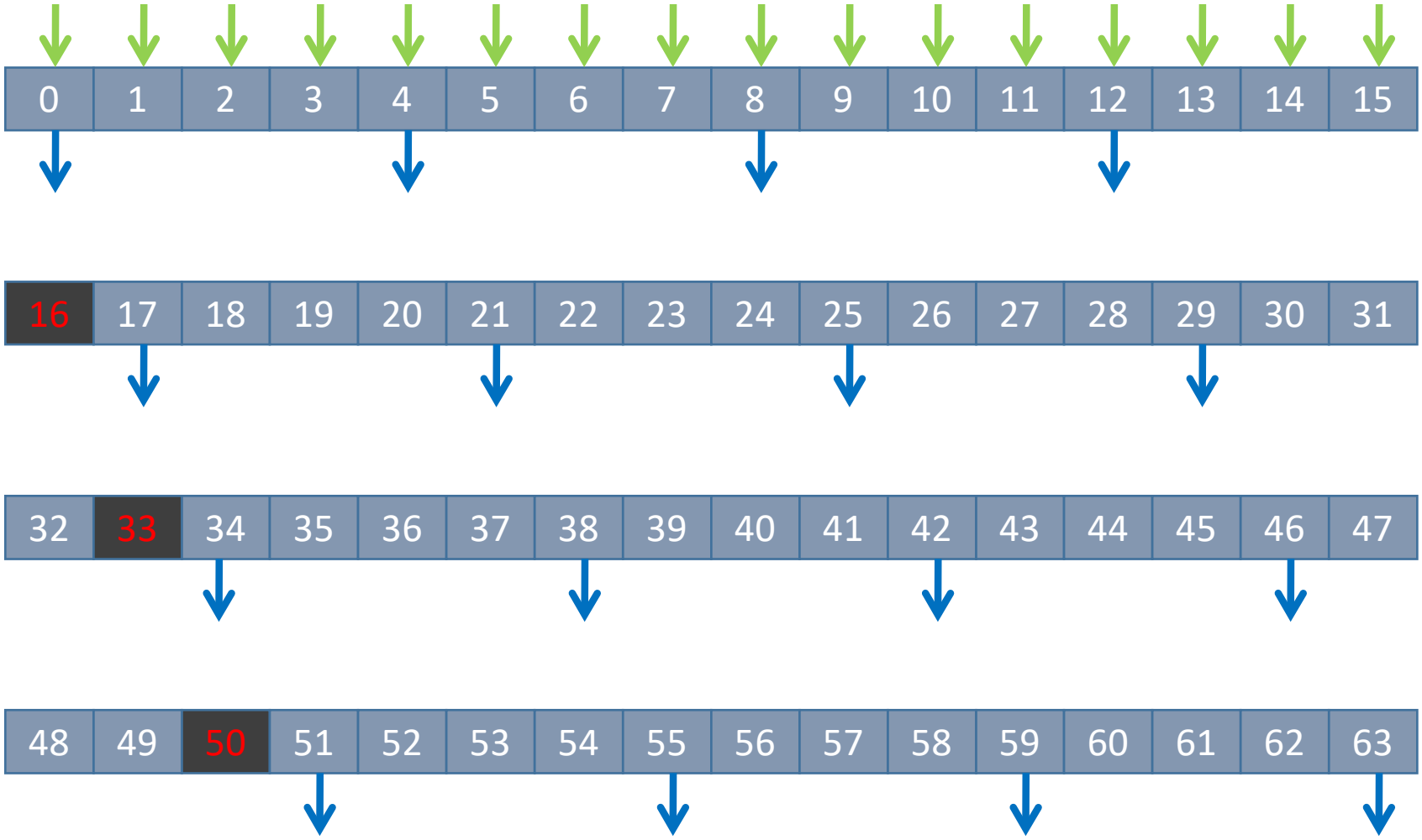
Default access pattern



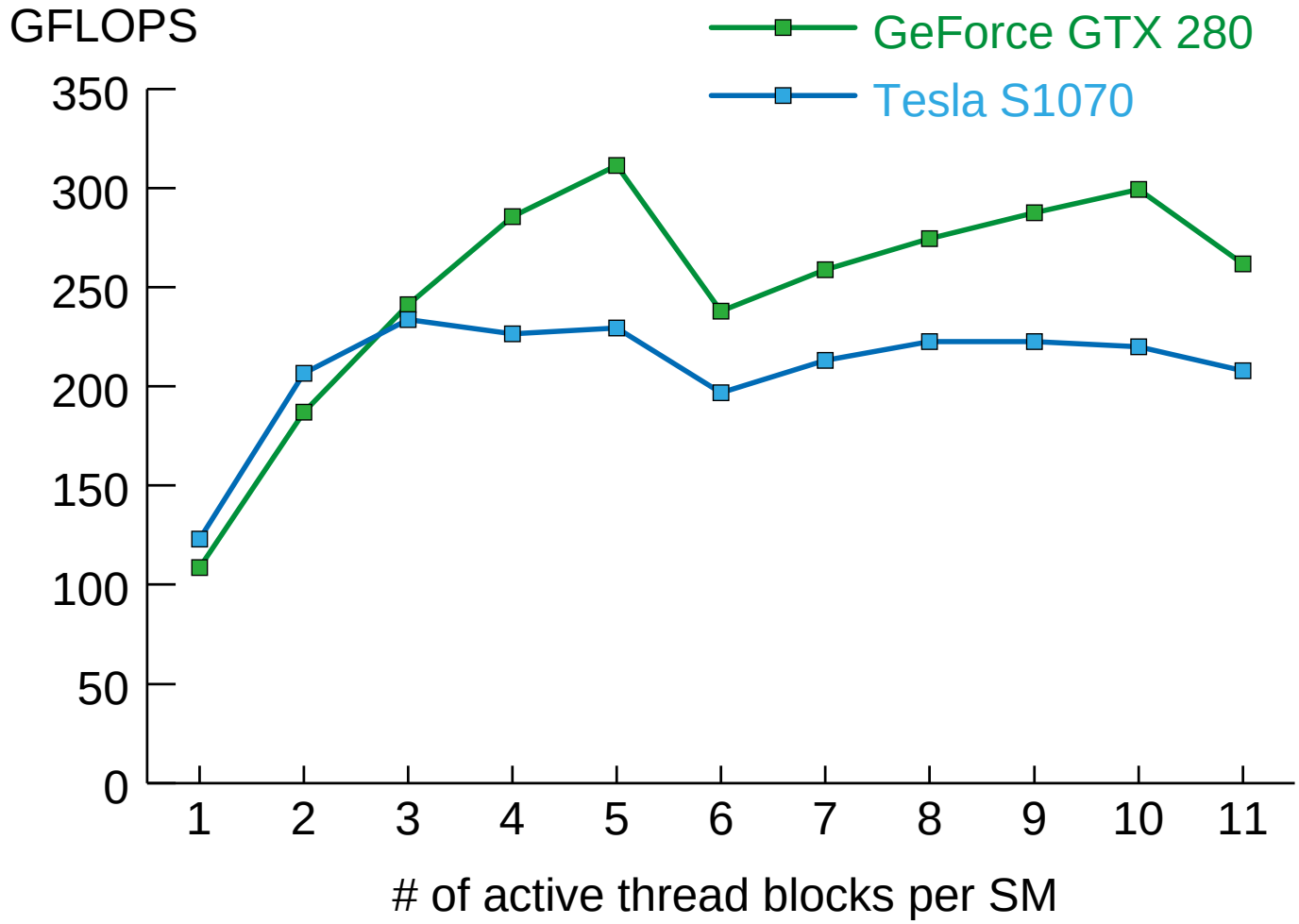
Padding (1) : after every block stride



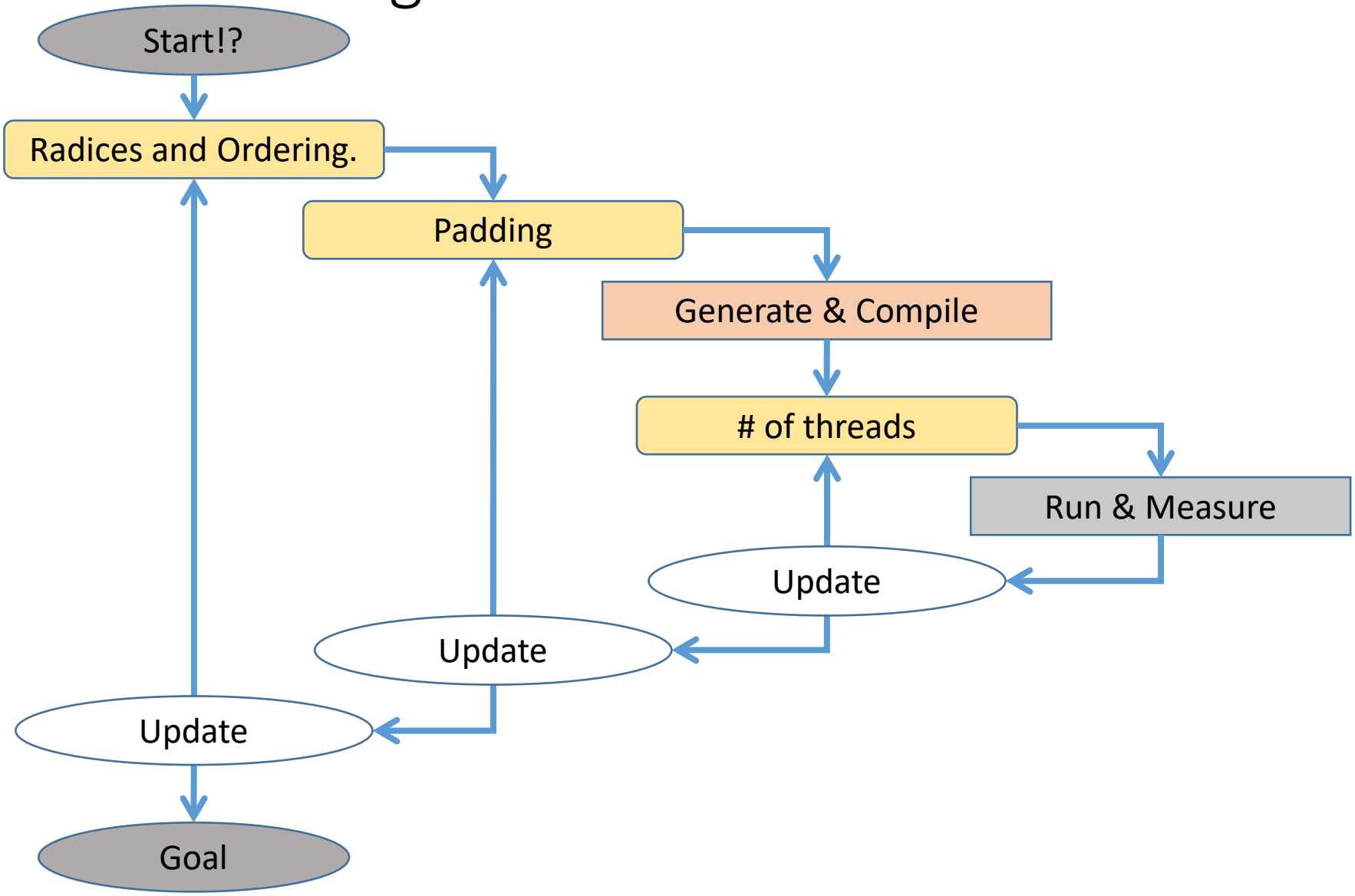
Padding (2) : after every 16n elements.



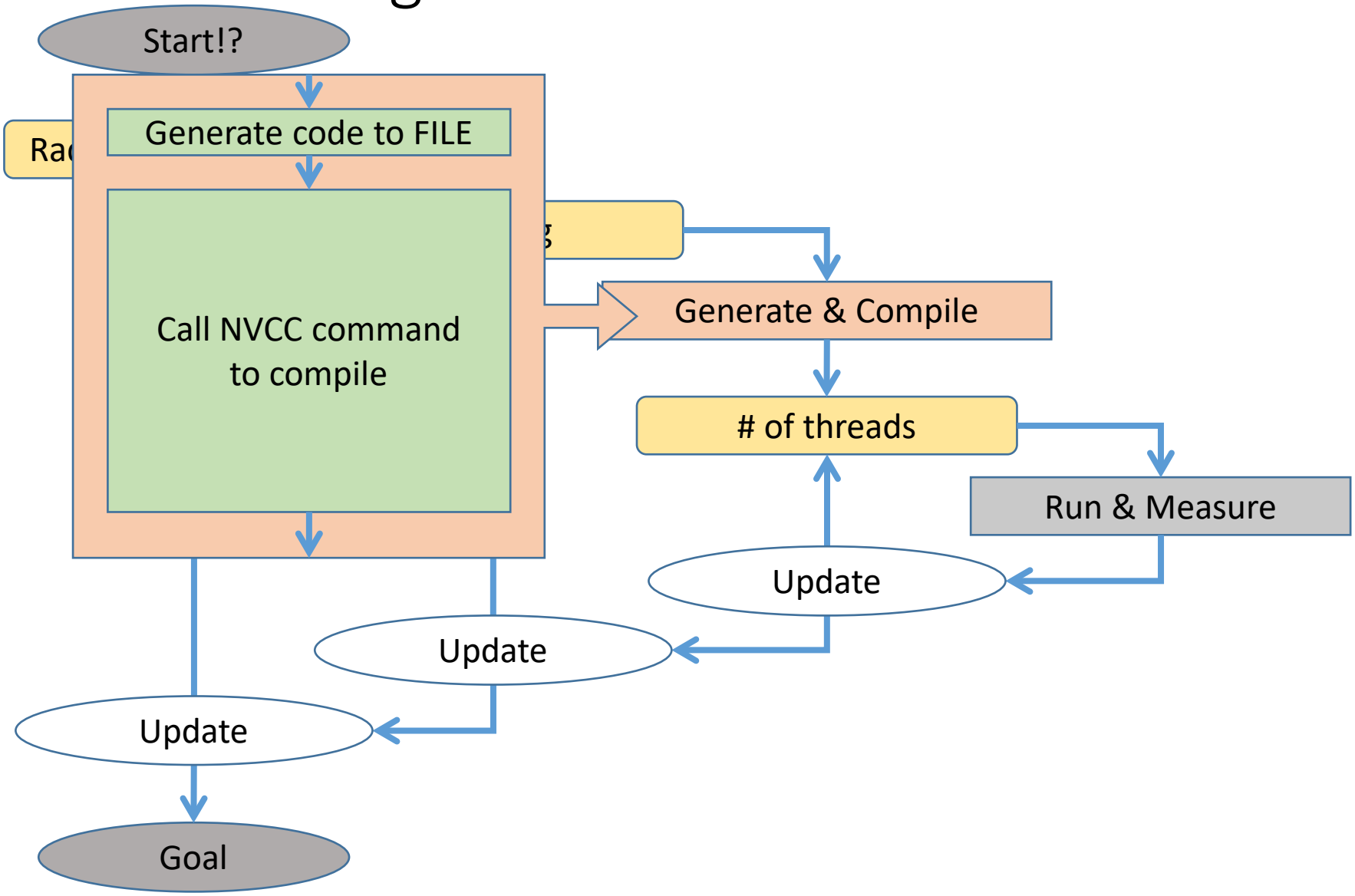
Adjustment of number of thread blocks



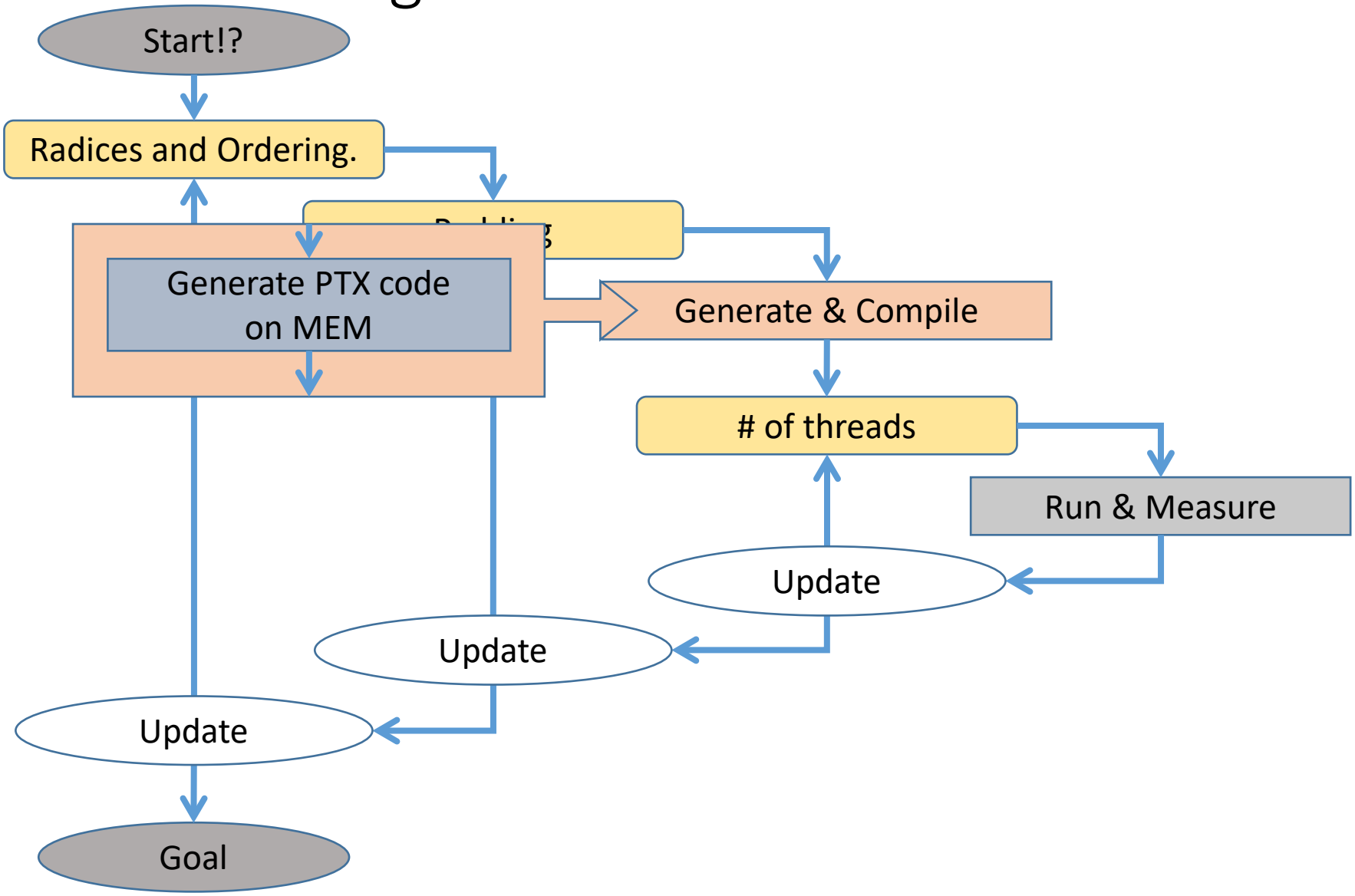
Auto-Tuning FFT for CUDA



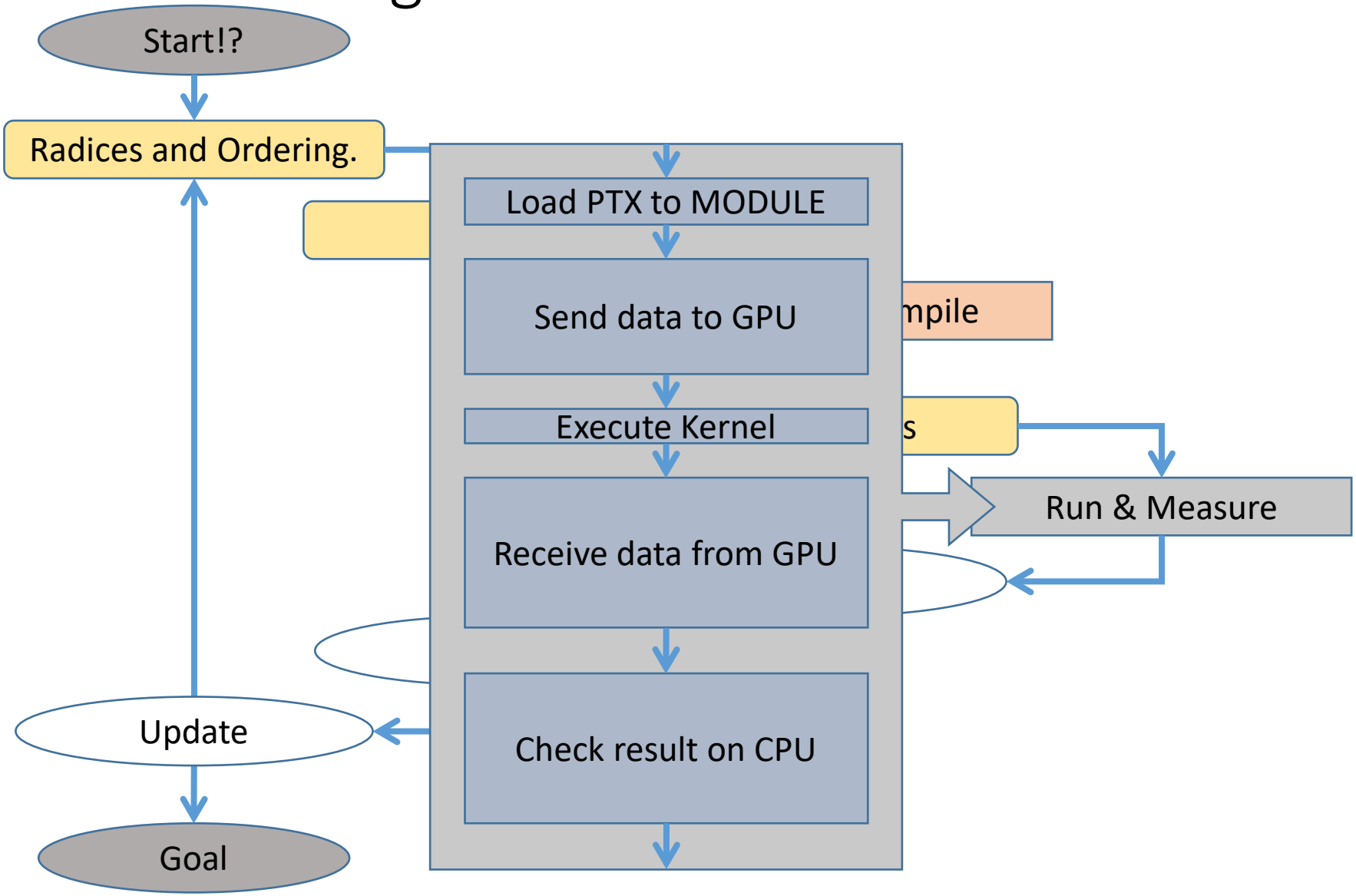
Auto-Tuning FFT for CUDA



Auto-Tuning FFT for CUDA



Auto-Tuning FFT for CUDA



CUDA PTX ~ Intermediate lang. ~

```
mov.u32        %r1, %ctaid.x;
mov.u32        %r2, %ntid.x;
mov.u32        %r3, %tid.x;
mad.lo.s32     %r4, %r2, %r1, %r3;
mul.wide.s32   %rd3, %r4, 4;
add.s64        %rd4, %rd2, %rd3;
ld.global.u32  %r5, [%rd4];
and.b32        %r6, %r4, 3;
setp.eq.s32    %p1, %r6, 0;
shl.b32        %r7, %r5, 1;
selp.b32       %r8, %r7, 0, %p1;
setp.eq.s32    %p2, %r6, 1;
mul.lo.s32     %r9, %r5, 3;
selp.b32       %r10, %r9, 0, %p2;
add.s32        %r11, %r8, %r10;
setp.eq.s32    %p3, %r6, 2;
shl.b32        %r12, %r5, 2;
selp.b32       %r13, %r12, 0, %p3;
add.s32        %r14, %r11, %r13;
setp.eq.s32    %p4, %r6, 3;
mul.lo.s32     %r15, %r5, 5;
selp.b32       %r16, %r15, 0, %p4;
add.s32        %r17, %r14, %r16;
st.global.u32  [%rd4], %r17;
```

CUDA PTX code

Parsing
Optimizations

```
__global__ void ex2(int *data) {
   int tmp, tmp2=0;
   tmp = data[x];
   tmp2 += ((x & 3) == 0)*(tmp * 2);
   tmp2 += ((x & 3) == 1)*(tmp * 3);
   tmp2 += ((x & 3) == 2)*(tmp * 4);
   tmp2 += ((x & 3) == 3)*(tmp * 5);
   data[x] = tmp2;
}
```

CUDA C code

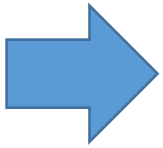
Register assignment
Instruction mapping

```
  XMAD R18, R23.reuse, R21.reuse, R28;              /* 0x5b000e0001571712 */
  XMAD.PSL.CBCC R18, R23.H1, R24.H1, R18;           /* 0x5b30091801871712 */
  XMAD R19, R22.reuse, R21.reuse, R19;              /* 0x5b00098001571613 */
  XMAD.MRG R21, R22, R21.H1, RZ;                    /* 0x5b007fa801571615 */
{ XMAD.PSL.CBCC R4, R22.H1, R21.H1, R19;            /* 0x5b30099801571604 */
   DEPBAR.LE SB5, 0x7;       }                      /* 0xf0f0000034770000 */
   XMAD.MRG R5, R15.reuse, R13.H1.reuse, RZ;        /* 0x5b007fa800d70f05 */
   XMAD R18, R15.reuse, R13.reuse, R18;             /* 0x5b00090000d70f12 */
   XMAD R4, R14.reuse, R13.reuse, R4;               /* 0x5b00020000d70e04 */
XMAD.MRG R13, R14, R13.H1, RZ;                      /* 0x5b007fa800d70e0d */
   XMAD.PSL.CBCC R15, R15.H1, R5.H1, R18;           /* 0x5b30091800570f0f */
{  XMAD.PSL.CBCC R14, R14.H1, R13.H1, R4;           /* 0x5b30021800d70e0e */
   DEPBAR.LE SB5, 0x1, {0};       }                 /* 0x181fc8c0fe2007f3 */
   XMAD.MRG R2, R25.reuse, R16.H1.reuse, RZ;        /* 0x5b007fa801071902 */
   XMAD R15, R25.reuse, R16.reuse, R15;             /* 0x5b0007800107190f */
```

CUDA assembly code

Compilers' optimization

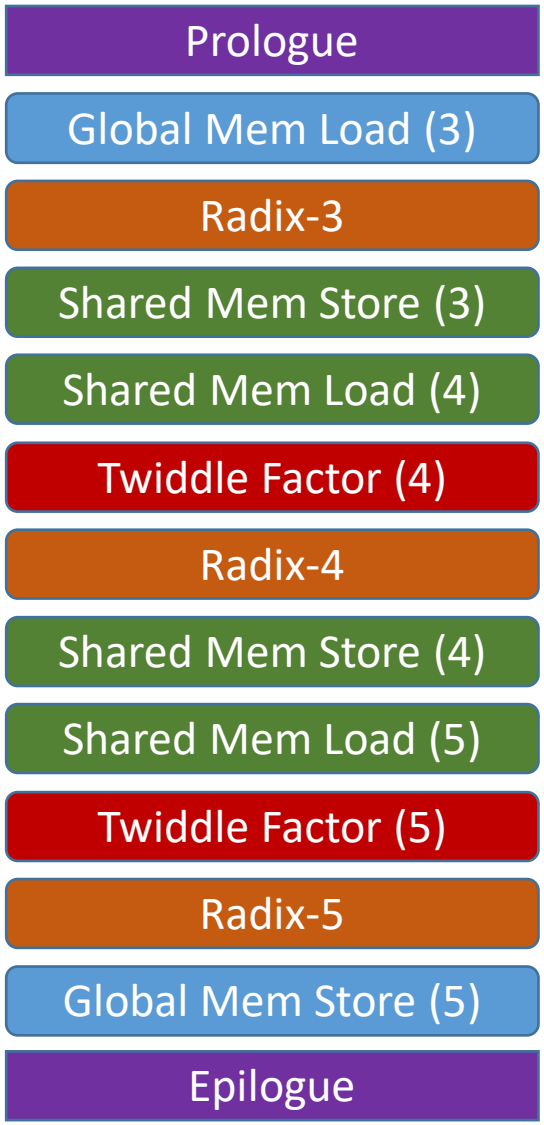
```
for (….) {
    global_load();
    radix4_fft();
    shmem();
    radix5_fft();
    shmem();
    radix8_fft();
    global_store();
}
```



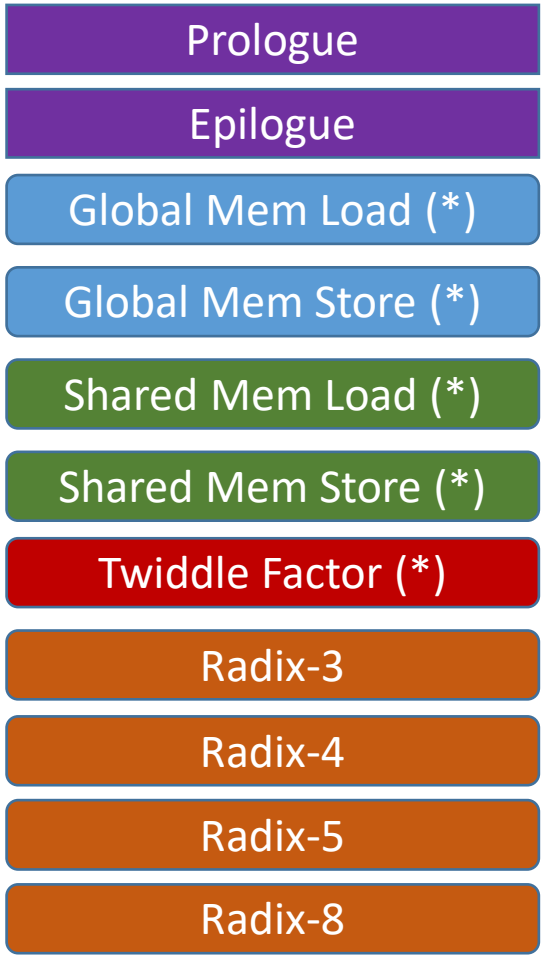
```
//Calculation of loop-independent values

for (….) {
    global_load2();
    radix4_fft2();
    shmem2();
    radix5_fft2();
    shmem2();
    radix8_fft2();
    global_store2();
}
```

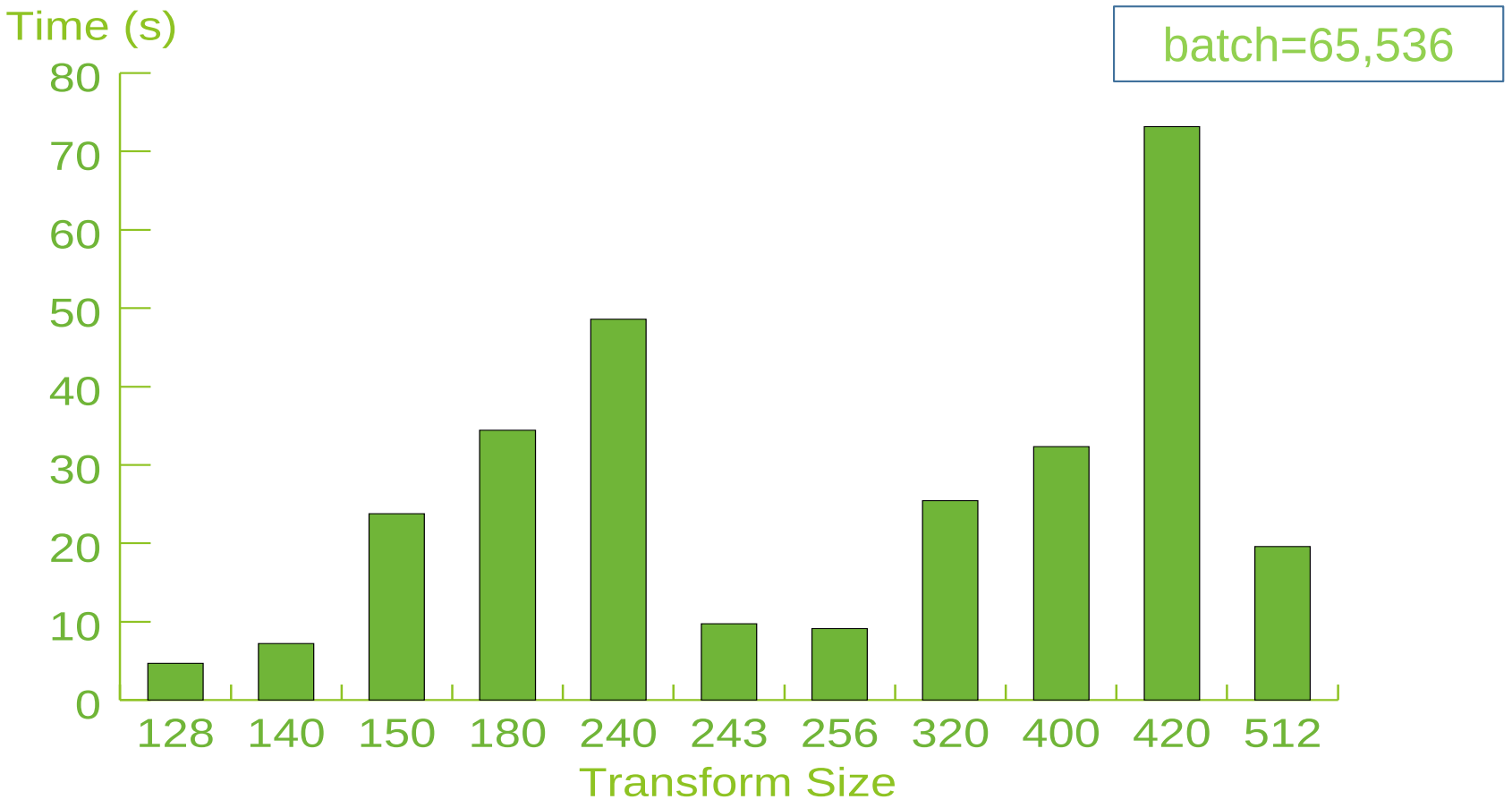
PTX generation



code blocks



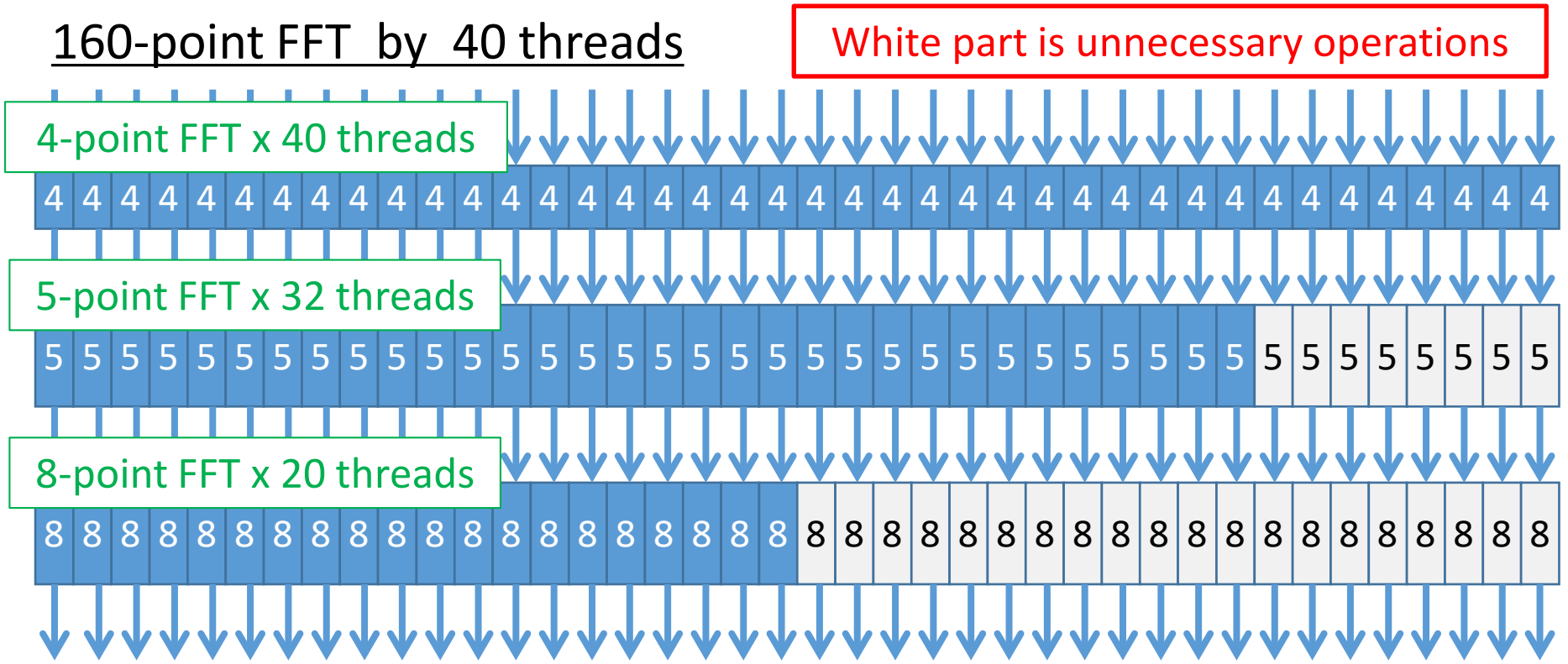
Time of auto-tuning for 1-D FFTs



Warp-level optimizations

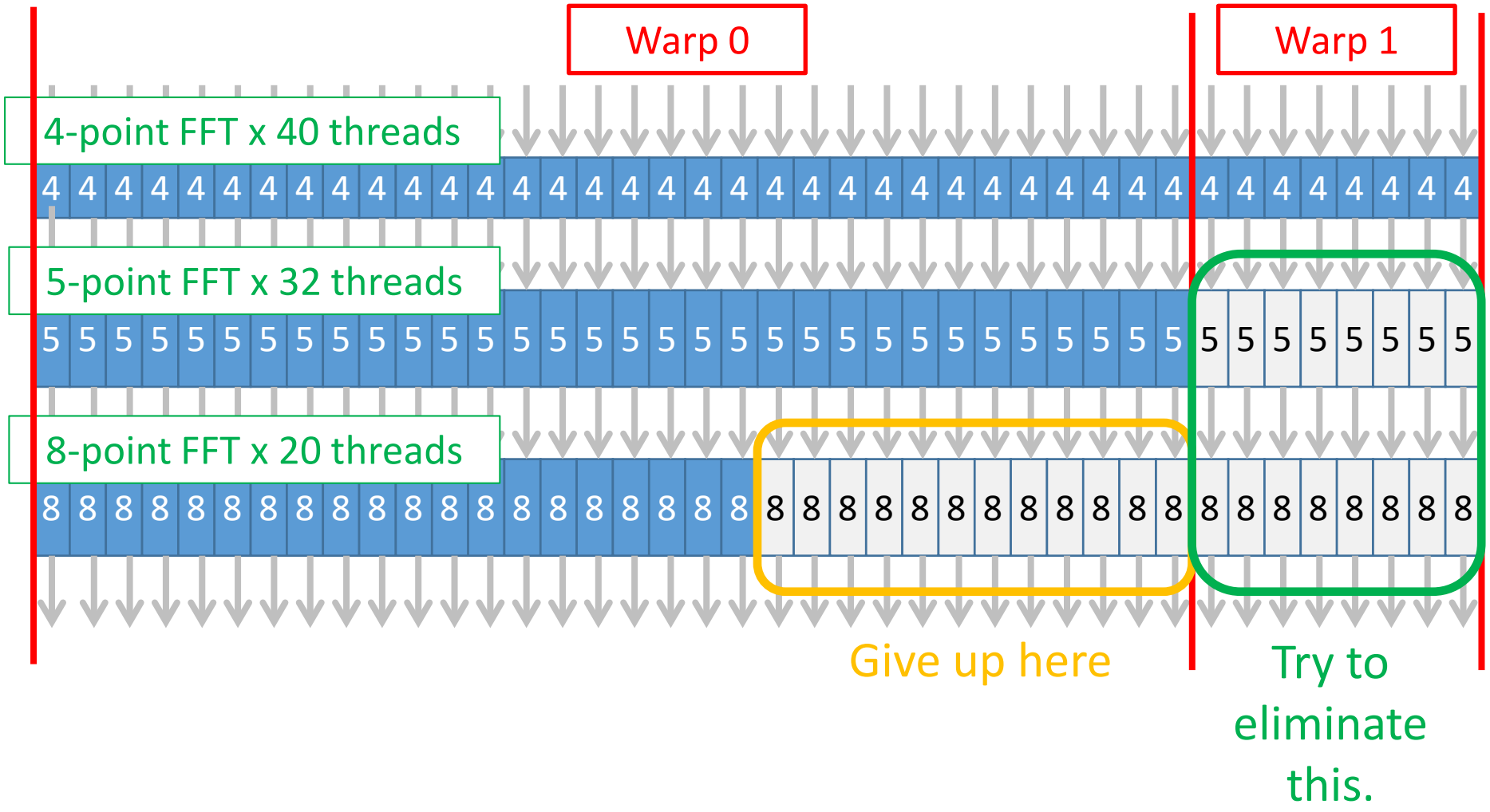
of threads per thread block is constant in kernel

→ Some threads may perform unnecessary ops.



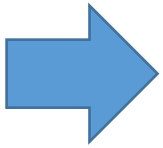
In warp level : 32 threads share instructions.

160-point FFT by 40 threads



Warp-level flow control.

```
for (….) {
    global_load();
    radix4_fft();
    shmem();
    radix5_fft();
    shmem();
    radix8_fft();
    global_store();
}
```



Warp 0

```
for (….) {
    global_load();
    radix4_fft();
    shmem();
    radix5_fft();
    shmem();
    radix8_fft();
    global_store();
}
```

Warp 1

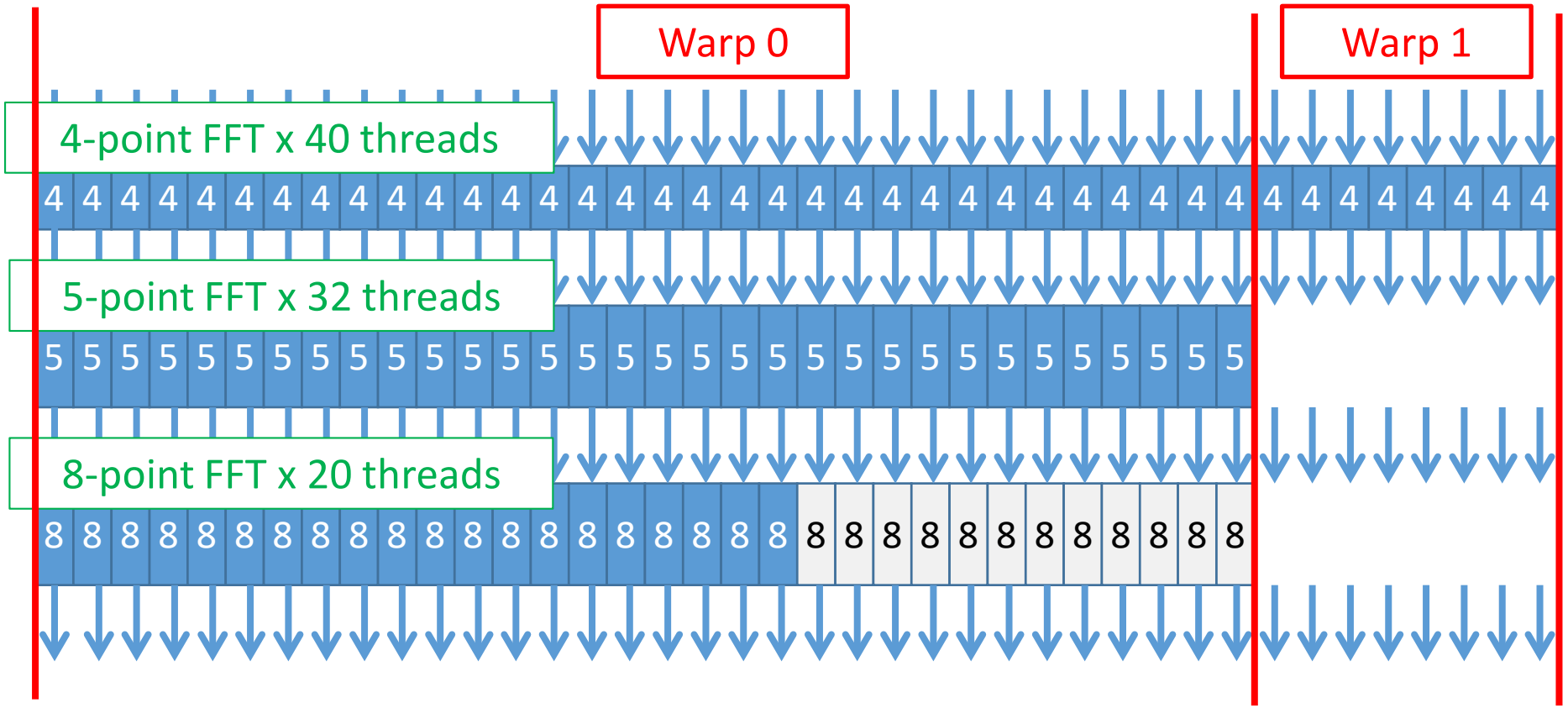
```
for (….) {
    global_load();
    radix4_fft();
    shmem();

    barrier();


}
```

Warp-level

160-point FFT by 40 threads



NukadaFFT library and Updates

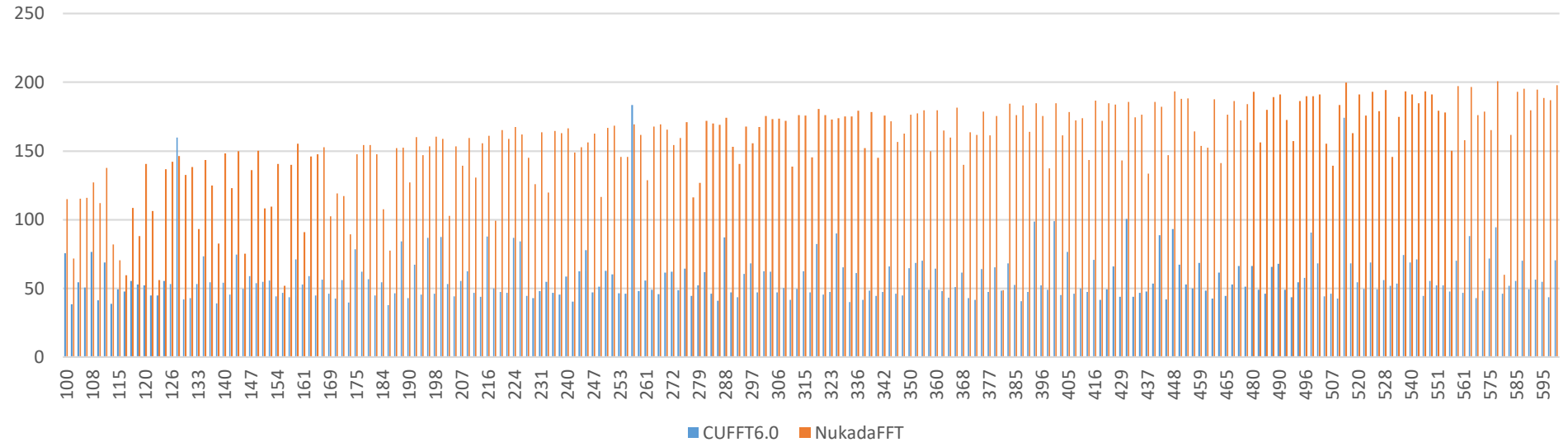
- Initial version for GT200 GPUs in 2010.
- Minor update (from PTX 1.4 to 2.0)
- Recompile for CUDA 4.0
- Recompile for CUDA 5.0
- Recompile for CUDA 6.0

……

On Maxwell GPU (GeForce 750 Ti)

65,536 batched 1-D FFT (N=100~600).

	Counts
NukadaFFT fails	274 (sizes which need prime factors > 32)
CUFFT fails	31 (several sizes greater than 512)
Both work and NukadaFFT is faster	225 (up to x4.48 speed-up)
Both work and CUFFT is faster	2 (128, 256)



Summary

- Auto-tuning enables long-life of performance library, while tuning time is another issue.
- On memory PTX generation & compile is fast, and maybe easier than you think.
- Cut-off algorithm may be able to reduce the tuning time.