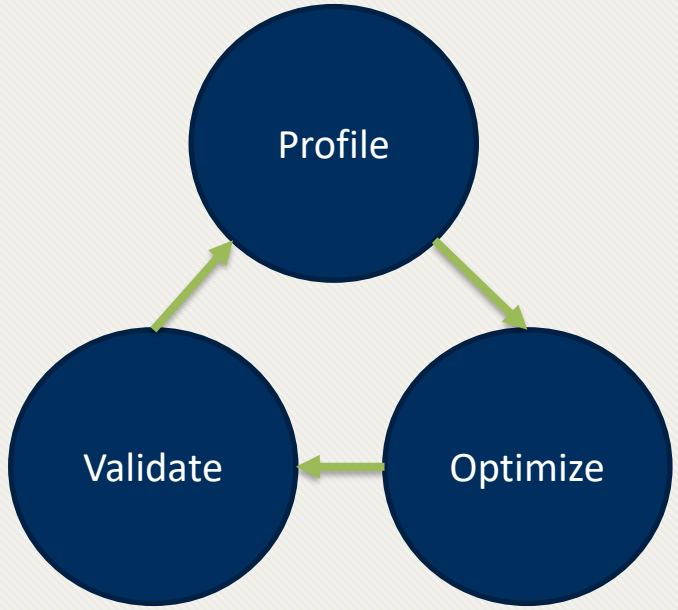
LeGendre Pairs,
A Hate Story Across the HPC Universe:
The Threads of Destiny

Distributive Book Club

August 16th , 2023

LeGendre Pairs?!

- Again, this talk is not about LeGendre Pairs!
  - Thank you, Alex for helping!
- It is about optimizing it.
  - But how to improve a program without fully understanding it?
- Only realize fundamental features, e.g.:
  - Various sections
  - Code semantics
  - Data and control flow
  - Repetition
  - Memory access pattern
  - Memory packing/unpacking
  - …
- Let’s see the code together!



LeGendre Pairs – General Observations

Length 45

- 3800+ lines of code
- Several nested loops -> huge search space
  - $126 * 84^4$
- Very large inner-most loop.
- Two main sections:
  1. Populate an array => fixed length (45)
  2. Perform computation and check some conditions on them
- Loop iterations:
  - Imbalanced and independent
- Deterministic
  - No random variable
- Several variables that can be packed

Length 117

- 8500+ lines of code
- Several nested loops -> huge search space
  - $126 * 84^{12}$
- Even larger inner-most loop.
- Two main sections:
  1. Populate an array => fixed length (117)
  2. Perform computation and check some conditions on them
- Loop iterations:
  - Imbalanced and independent
- Deterministic
  - No random variable
- Several variables that can be packed

Reminder: Multi-threaded Version and Parallel Efficiency

- How does increasing cores affect performance?
- Let's call OpenMP for help!
  - It is an API for shared-memory multiprocessing programming in C/C++, and Fortran.
- How to tell if increasing cores will definitely improve the performance?
  - **Parallel efficiency**

Reminder: Parallel Efficiency

- The parallel efficiency of a program is the ratio of the speedup factor S(n) and the number of processors. **Efficiency = S(n) / n**

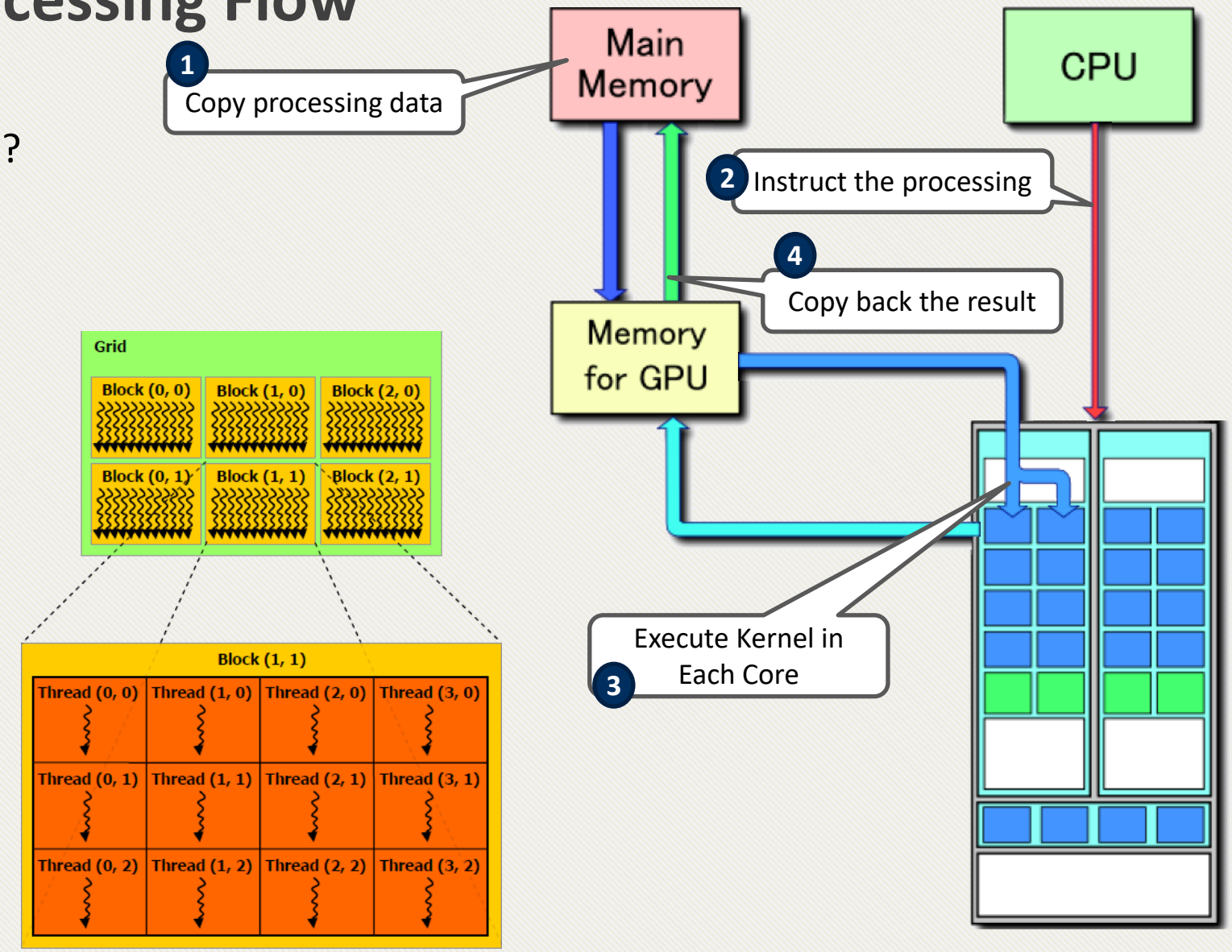
#	Version (LP 45)	No of CPU cores	Speedup vs. #1	Parallel Efficiency
1	with **-O3** compiler flag	1	1	-
2	#1 with **2** OMP threads (outermost loop)	2	1.91x	0.95
3	#1 with **4** OMP threads (outermost loop)	4	3.76x	0.94
4	#1 with **6** OMP threads (outermost loop)	6	5.66x	0.94

- This virtually means that as long as we have free cores, we can improve the performance.

Reminder: GPU Processing Flow

- How can we run a code in GPU?

- Keywords to remember:
  - SM (Streaming Multiprocessor)
  - ThreadBlock & GridBlock
  - Warp (SIMD execution)
  - Kernel



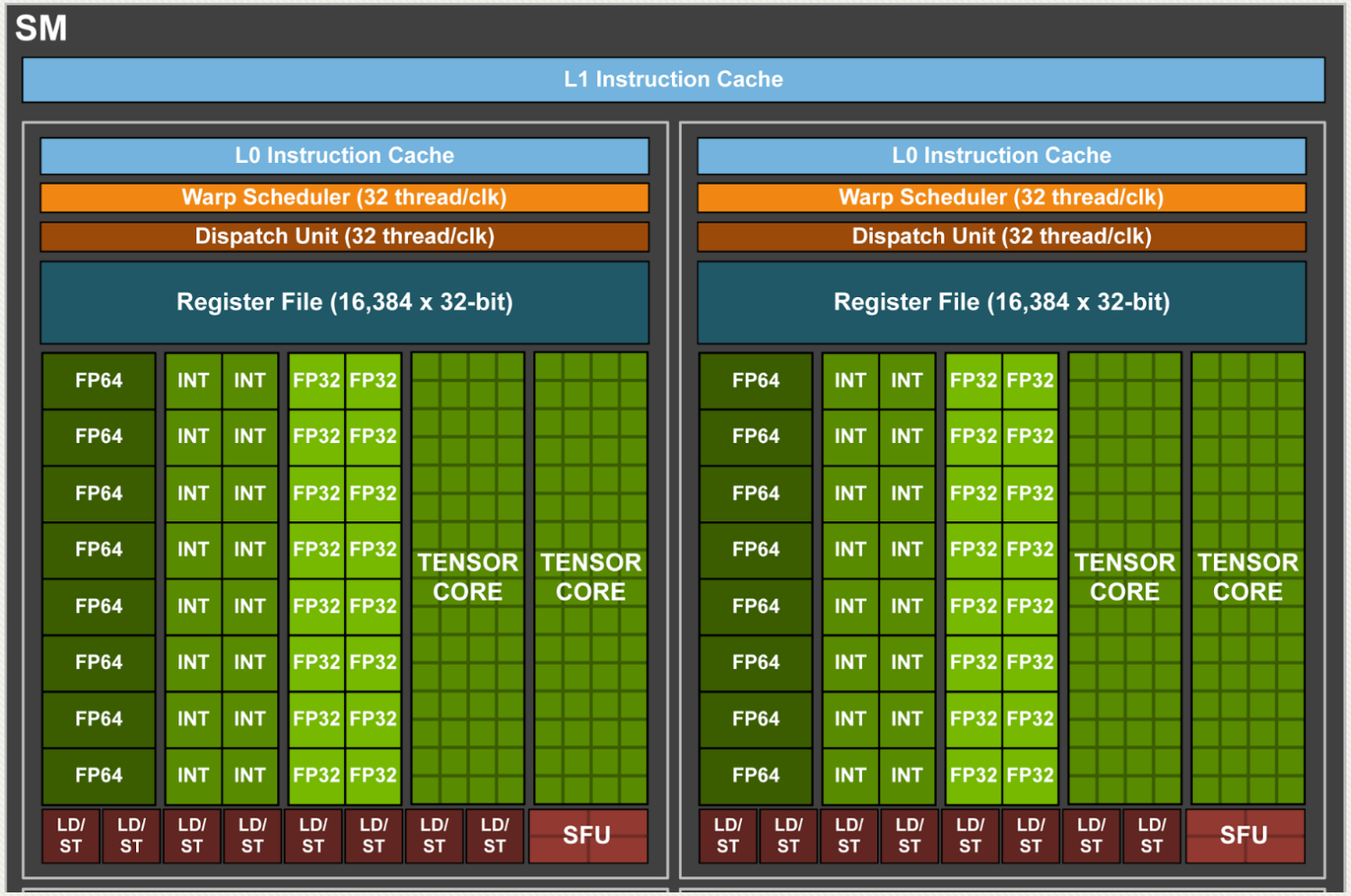
LP 45 - CUDA versions

- Profile and monitor GPU codes:
  - Nvtop
  - NVIDIA Visual Profiler (nvvp)
  - NVIDIA Nsight Systems (nsys)
  - NVIDIA Nsight Compute (ncu)

- Results are gathered from NVIDIA GeForce GTX 1060 3GB, and RTX 3070 8GB
  - Need to measure data transfers, too.
  - Table of results

Branch Divergence

- Simultaneous Multiprocessors
- SIMD (or SIMT) Architectures
- Warp scheduling

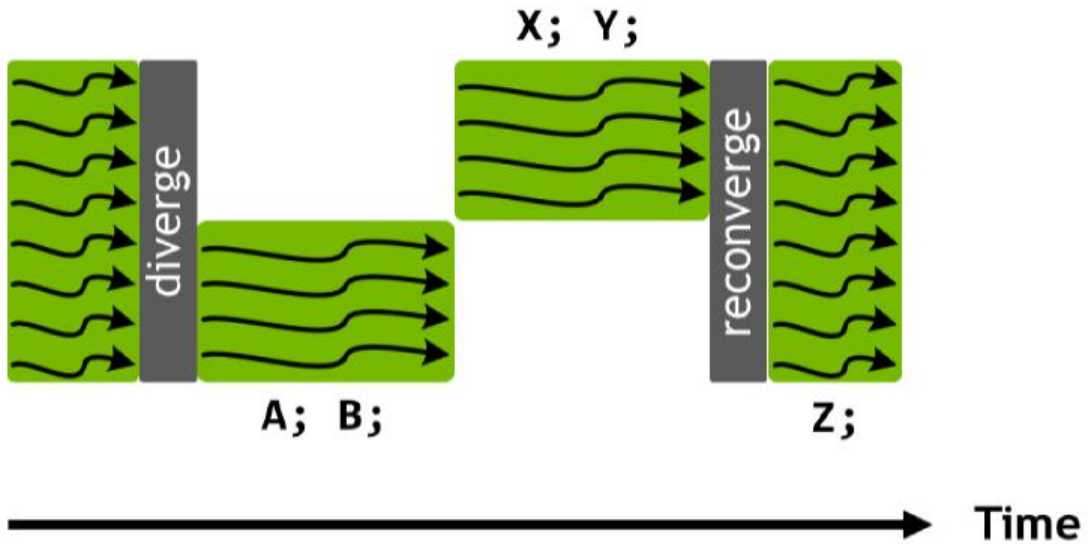


Images' source: NVIDIA and Illawarra Dragon Boat Club

Branch Divergence

- SIMT architecture and warp execution model

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```



- Why is branch divergence important?!

Image source: NVIDIA

Optimization – Remove Branch Divergence

- Figure out permutations: 2 options
  - Dynamically – by each thread
  - Statically and inform all threads through (constant) memory
    - But some parts of constant memory is already used. Solution?

LP 117 Optimizations – CUDA V2, Constant Memory

- GPUs have a dedicated memory section for constant memory.
- Utilizing constant memory significantly decreases the frequent memory accesses latency
- Let’s see the code.

CUDA V3, Shared Memory

- Problem with local variables and register spilling
  - Compiler utilizes Global Memory to store local variables
  - Global memory is cached, but if we want to specifically cache some variable, we should use Shared Memory

CUDA V4, Loop Unrolling

- Some of the operations are redundantly done by each thread.
- Unrolling and moving some loops around would be helping with that.

CUDA V5, Fuse PSD Calculations

- It turns out that inner-loop computations were not random!
- Take PSDs[4] into account as an example.

CUDA V6, Removing Shared Memory!

- NVIDIA Visual Profiler to the rescue
- GPU Occupancy is low because we have used up the shared memory available

CUDA V7, Cache A3 and A13

- Some of the calculations are redundant
- We can store and re-use those as intermediate computations.

CUDA V8, Ctrl + Z !

- Other than some minor optimization, revert everything.
- Why did this happen?

- Curse of optimization before validation

Several other optimizations that did not help

- Utilizing FFTW library on CPU
- Moving another dimension inside the kernel
- Move two other dimensions inside the kernel
- Moving A to global memory
- Perform reduction using shared memory.
  - This is really cool by the way, but it didn’t help… pff

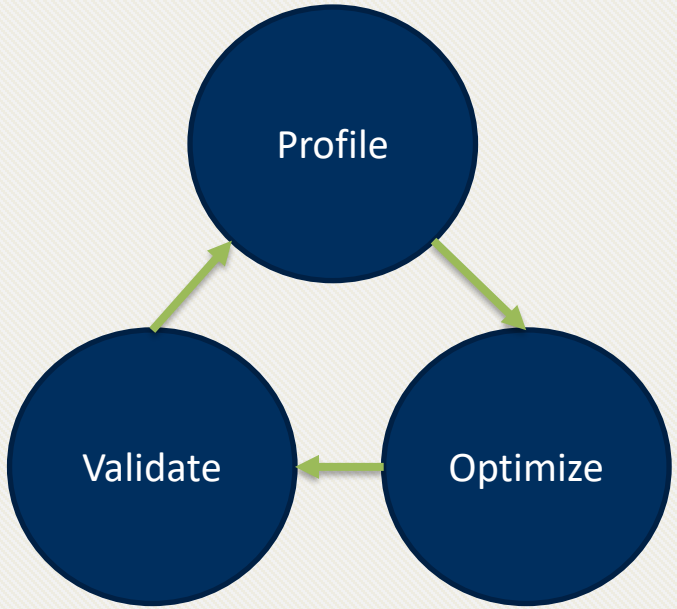
WebGPU v1

- Let’s dive right into the code!

Conclusions

- The same optimizations apply to L117 version.
- Some of them are device-specific, but they have little share of performance.

- Git and Make are your friends
- Profilers and debuggers are your best friends
- Never give up when it comes to optimization!
  - Just remember the cycle for each update



Thank You ☺

Instead of blaming darkness, let’s light a candle!