LeGendre Pairs, a Love Story Across the HPC Universe:

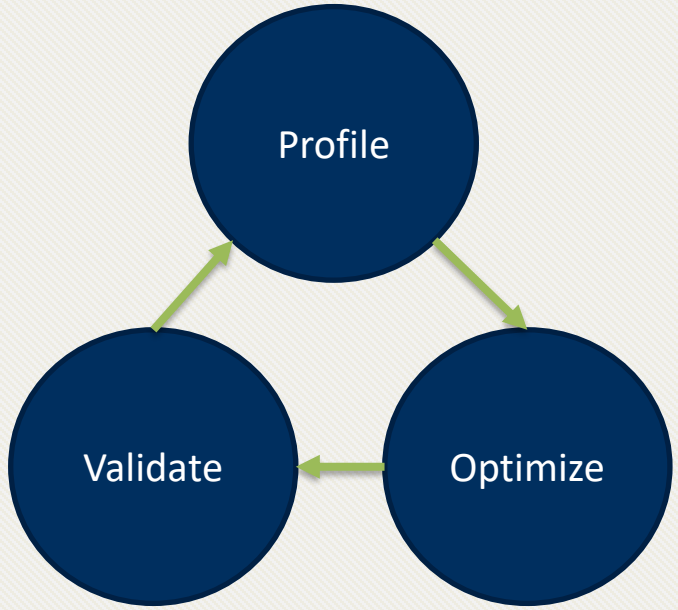
The Cores of Power!

Distributive Book Club

November 9th , 2022

LeGendre Pairs?!

- Actually, this talk is not about LeGendre Pairs!
- 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 L45 – General Observations

- 3800+ lines of code
- Several nested loops -> huge search space
- Very large inner-most loop.
- Two main sections:
  1. Populate an array
  2. Perform computation and check some conditions on it
- Loop iterations:
  - Imbalanced
  - Independent
- Deterministic
  - No random variable
- No race condition (yet!)
- Several variables that can be packed

LeGendre Pairs L45 – Analysis

- Runtime profile and analysis using runtime tools, e.g. **gprof**

```
$ gcc source -o executable $(FLAGS) -g –pg
$ ./executable
$ gprof --all-lines --line -b executable gmon.a.out > analysis.a.out
```

- Let's see the output and some data
- Section 1:
  - takes 30% of the whole execution time
  - is memory-bound. So, utilizing Cache, Shared memory, or Constant Memory improves its performance.
  - (On GPU) Coallesced memory access is also very important, as there are some patterns in Section A.

LeGendre Pairs L45 – Analysis

- Section 2:
  - is compute-bound (if caching is utilized properly).
  - Having each thread performing each iteration makes sense. However, the iterations are wildly imbalance.
  - Less than 0.01 percent of the iterations makes to the final condition check; (dynamic task assignment?).
  - DFT computations can be handled by multiple threads in reduction-like pattern. O(logn)

- Print operations
  - They do not take considerable time
  - When porting to GPU, the results should be buffered and gets returned to the CPU, either at the end of each slice, or frequently during running the kernel.

LeGendre Pairs L45 – Analysis

- Let's cleanup the code
- Execution time on **Polydeuces**:

#	Version	Duration of part A (ms)	Speedup vs. #1	Duration of Part B	Speedup vs. #1
1	Single CPU core – cleaned up	1188047.15	1x	578017.02	1x
2	#1 with Buffered Results	1178997.37	1.01x	576614.09	1.01x
3	#2 with **-O3** compiler flag	725519.95	1.63x	374696.34	1.54x

Multi-threaded Version

- Let's call OpenMP for help!
  - It is an API for shared-memory multiprocessing programming in C/C++, and Fortran.

- Pay attention to shared data and local data
- Watch out race conditions

- Let's see the code.

#	Version	Duration of part A (ms)	Speedup vs. #1	Duration of Part B	Speedup vs. #1
1	Single CPU core – cleaned up	1188047.15	1x	578017.02	1x
2	#1 with Buffered Results	1178997.37	1.01x	576614.09	1.01x
3	#2 with **-O3** compiler flag	725519.95	1.63x	374696.34	1.54x
4	#2 with **2** OMP threads (outermost loop)	642096.61	1.85x	335816.28	1.72x
5	#4 with **-O3** compiler flag	380146.53	3.12x	185300.15	3.11x

Multi-threaded Version

- How does increasing cores affect performance?

#	Version	Duration of part A (ms)	Speedup vs. #1	Duration of Part B	Speedup vs. #1
1	Single CPU core – cleaned up	1188047.15	1x	578017.02	1x
2	#1 with Buffered Results	1178997.37	1.01x	576614.09	1.01x
3	#2 with **-O3** compiler flag	725519.95	1.63x	374696.34	1.54x
4	#2 with **2** OMP threads (outermost loop)	642096.61	1.85x	335816.28	1.72x
5	#4 with **-O3** compiler flag	380146.53	3.12x	185300.15	3.11x
6	#5 with **4** OMP threads (outermost loop)	192162.96	6.19x	107076.45	5.3x

- How to tell if increasing cores will definitely improve the performance?
  - **Parallel efficiency**

Parallel Efficiency

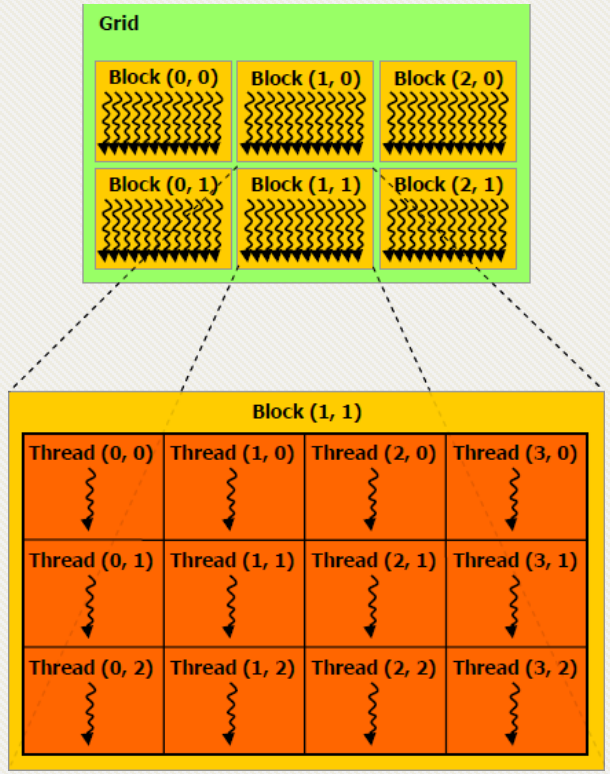
- The parallel efficiency of a program is the ratio of the speedup factor S(n) and the number of processors. **Efficiency = S(n) / n**

#	Version	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?
- There are very Mature environments & Platforms:
  - CUDA, OpenCL, …
- Keywords to remember:
  - SM (Streaming Multiprocessor)
  - ThreadBlock & GridBlock
  - Warp (SIMD execution)
  - Kernel



LeGendre Pairs - CUDA v1

- Let's see the code first.
- Need to measure data transfers, too.
- Profile and monitor GPU codes:
  - Nvtop
  - NVIDIA Visual Profiler (nvvp)
  - NVIDIA Nsight Systems (nsys)
  - NVIDIA Nsight Compute (ncu)
- Results are for running on NVIDIA GeForce GTX 1060 3GB

#	Version	Duration of part A (ms)	Speedup vs. #1	Duration of Part B	Speedup vs. #1
1	Single CPU core – cleaned up	1188047.15	1x	578017.02	1x
10	CUDA v1 (126 x 84) = 10584 threads	55190.91	21.5x	48097.78	11.8x

LeGendre Pairs - CUDA v2 – Constant Memory

- GPUs have a dedicated memory section for constant memory.
- This decreases memory access latency significantly.

#	Version	Duration of part A (ms)	Speedup vs. #1	Duration of Part B	Speedup vs. #1
1	Single CPU core – cleaned up	1188047.15	1x	578017.02	1x
2	CUDA v1 (126 x 84) = 10584 threads	55190.91	21.5x	48097.78	11.8x
3	CUDA v2 (126 x 84) – Constant Memory	29913.48	39.71x	23396.84	24.7x

LeGendre Pairs - CUDA v3 – Fill Up Scheduler Queue

- Utilize 3D grids of blocks and 2D blocks of threads
  - Basic strategy: each dimension covers one of the loops
  - (126 x 84 x 84) x (32 x 16) = 455196672 threads!
  - Let's see the code.

#	Version	Duration of part A (ms)	Speedup vs. #1	Duration of Part B	Speedup vs. #1
1	Single CPU core – cleaned up	1188047.15	1x	578017.02	1x
2	CUDA v1 (126 x 84) = 10584 threads	55190.91	21.5x	48097.78	11.8x
3	CUDA v2 (126 x 84) – Constant Memory	29913.48	39.71x	23396.84	24.7x
4	CUDA v3 – 3D grids	18837.89	63.05x	16291.29	35.4x

LeGendre Pairs - CUDA v4 – Tuned Kernel Parameters

- This optimization is almost hardware specific. It is based on:
  - Number of threads per warp (group of threads that run together) - 32
  - Number of registers available for each block, and *register spilling* problem
  - Amount of Constant Memory accessible on each SM
  - Hardware architecture and compute capabilities
  - So, the best parameters will vary from device to device. Let's check Nvvp and Nvtop!
- The best parameters on my machine is: (126 x 84 x 84) x (8 x 32)

#	Version	Duration of part A (ms)	Speedup vs. #1	Duration of Part B	Speedup vs. #1
1	Single CPU core – cleaned up	1188047.15	1x	578017.02	1x
2	CUDA v1 (126 x 84) = 10584 threads	55190.91	21.5x	48097.78	11.8x
3	CUDA v2 (126 x 84) – Constant Memory	29913.48	39.71x	23396.84	24.7x
4	CUDA v3 – 3D grids	18837.89	63.05x	16291.29	35.4x
5	CUDA v4 – with tuned kernel parameters	15920.46	74.62x	13680.04	42.25x

LeGendre Pairs – CUDA v5 – Utilizing Shared Memory

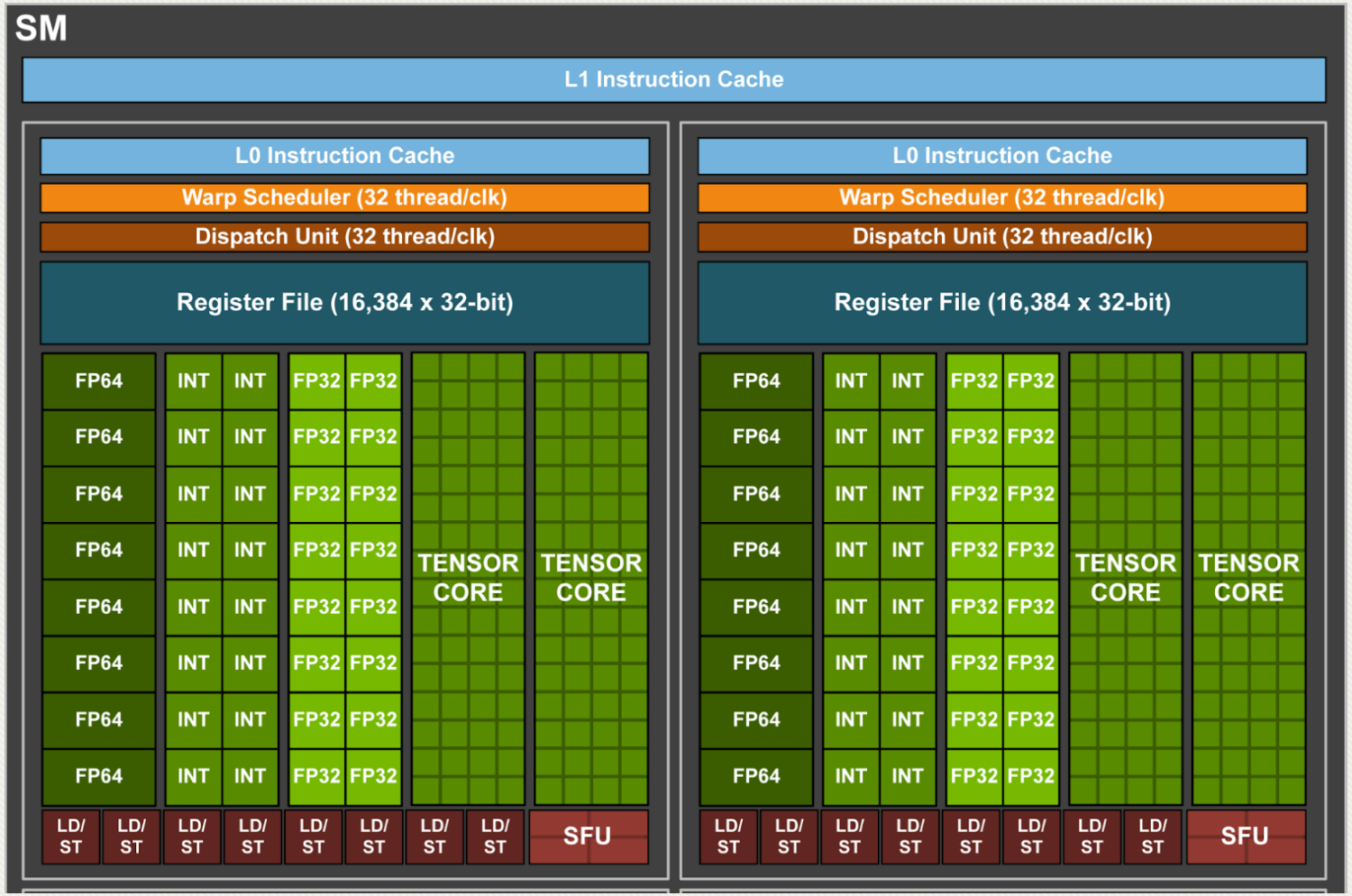
- Problem with local variables and register spilling
  - Compiler utilizes Global Memory to store local variables
  - Global memory is cached, but if we want to specifically cache some variable, we should use Shared Memory
- Let's see the code.

#	Version	Duration of part A (ms)	Speedup vs. #1	Duration of Part B	Speedup vs. #1
1	Single CPU core – cleaned up	1188047.15	1x	578017.02	1x
2	CUDA v1 (126 x 84) = 10584 threads	55190.91	21.5x	48097.78	11.8x
3	CUDA v2 (126 x 84) – Constant Memory	29913.48	39.71x	23396.84	24.7x
4	CUDA v3 – 3D grids	18837.89	63.05x	16291.29	35.4x
5	CUDA v4 – with tuned kernel parameters	15920.46	74.62x	13680.04	42.25x
6	CUDA v5 – Store A in Shared Memory	15032.12	79.03x	13022.03	44.38x

- It seems that we are hitting the limit. **Or are we?!**

Branch Divergence

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

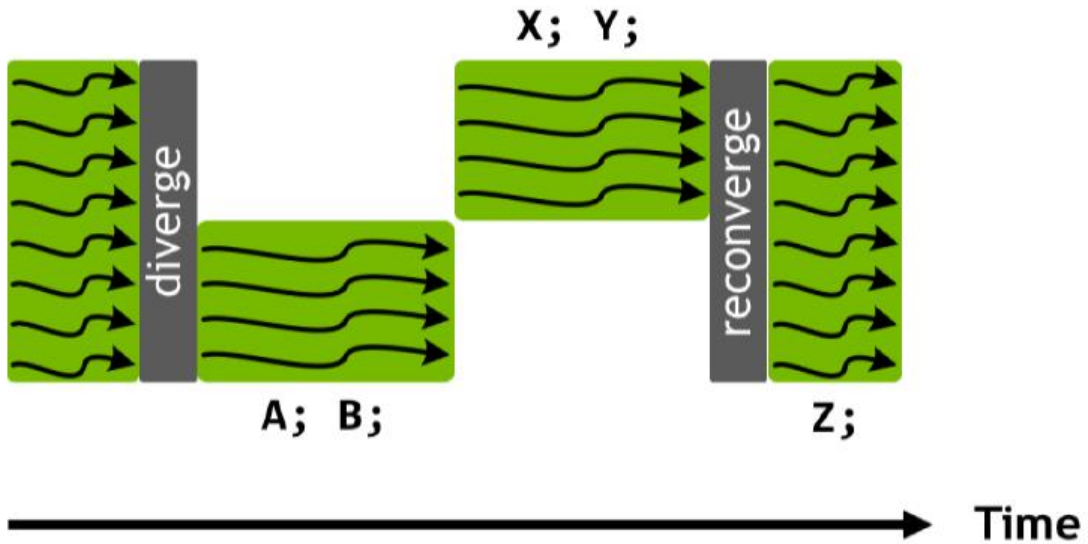


Images' source: NVIDIA and Illawarra Dragon Boat Club

Branch Divergence

- SIMT architecture and warp execution model

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



- Why is branch divergence important?!
- Let's check the profiler.

Image source: NVIDIA

LeGendre Pairs – CUDA v6 – Remove Branch Divergence

- Figure out permutations:
  - Dynamically – by each thread
  - Statically and inform all threads through (constant) memory
    - But some parts of constant memory is already used. Solution?
  - Let's see the code

#	Version	Duration of part A (ms)	Speedup vs. #1	Duration of Part B	Speedup vs. #1
1	Single CPU core – cleaned up	1188047.15	1x	578017.02	1x
2	CUDA v1 (126 x 84) = 10584 threads	55190.91	21.5x	48097.78	11.8x
3	CUDA v2 (126 x 84) – Constant Memory	29913.48	39.71x	23396.84	24.7x
4	CUDA v3 – 3D grids	18837.89	63.05x	16291.29	35.4x
5	CUDA v4 – with tuned kernel parameters	15920.46	74.62x	13680.04	42.25x
6	CUDA v5 – Store A in Shared Memory	15032.12	79.03x	13022.03	44.38x
7	CUDA v6 – Remove Branch Divergence	7357.17	**161.48x**	5011.70	115.33x

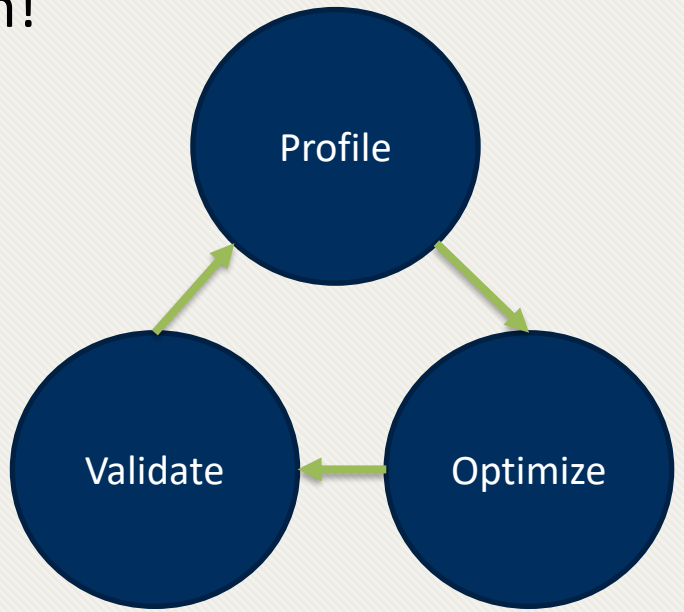
LeGendre Pairs – CUDA v7 – Tuned Kernel Parameters

- You'd think we are done! But no! Let's see one last profile.
- The best parameters was: **(126 x 84 x 84) x (8 x 32)**
- Now let's fill up the warp with new configuration: **(126 x 84 x 84) x (32 x 4)**

#	Version	Duration of part A (ms)	Speedup vs. #1	Duration of Part B	Speedup vs. #1
1	Single CPU core – cleaned up	1188047.15	1x	578017.02	1x
2	CUDA v1 (126 x 84) = 10584 threads	55190.91	21.5x	48097.78	11.8x
3	CUDA v2 (126 x 84) – Constant Memory	29913.48	39.71x	23396.84	24.7x
4	CUDA v3 – 3D grids	18837.89	63.05x	16291.29	35.4x
5	CUDA v4 – with tuned kernel parameters	15920.46	74.62x	13680.04	42.25x
6	CUDA v5 – Store A in Shared Memory	15032.12	79.03x	13022.03	44.38x
7	CUDA v6 – Remove Branch Divergence	7357.17	161.48x	5011.70	115.33x
8	CUDA v7 – with tuned kernel Parameters	5863.91	**202.6x**	3594	160.8x

Conclusions

- 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!

Questions, Comments, and Ideas are Welcome!