CS639: Homework 1 (git) Rajesh Shashi Kumar

1 Machine Configuration

Attribute	**Value**
Hostname	barolo.cs.wisc.edu
OS	Ubuntu 20.04.5 LTS
Compiler	g++ (gcc version 9.4.0), OpenMP version 4.5
Compile command	*g++ main.cpp Laplacian.cpp -Wall -O3 -o laplacian3D -fopenmp* *(Makefile included in root directory)*
CPU	AMD EPYC 7451 24-Core Processor
Cache configuration	L1-3MiB, L2-24MiB, L3-128MiB
Memory bandwidth	~150 GiB/s (Source,2). The machine has 256GB of memory spread across (8 out of 16) slots each housing 32GB stick. This information was retrieved using *sudo lshw -class memory*
Number of threads	24 cores * 2 threads/core = 48 threads
Dependencies	GCC, OpenMP

2 Theoretical estimate of performance for 3D Laplacian

Assuming perfect caching and reuse per iteration, every element of u[][][] is read at least once from memory even if accessed several times from the cache (without eviction). Similarly, across iterations, every element of Lu[][][] is written to. This is a reasonable estimate for the upper bound on performance. By this metric, we have

- Two arrays (u,Lu) of size 512x512x512 elements each, for a total 2×512^3 = **268,435,456**
- Each element is a float, that consumes **4Bytes**
- Approximate memory footprint = 1,073,741,824 bytes = 1024 MiB/s = **1GiB**
- At ~150GiB/s of theoretical memory bandwidth, I estimate a **theoretical runtime of ~6.6ms.**

3 Performance measurements

We use the instrumented timers in the code to measure the runtime that can be used to calculate the effective memory bandwidth utilization by using the above upper bound memory footprint value. The measure values are captured in the table below. As expected, we see that the algorithm reached near peak values for memory bandwidth when utilizing all the threads available on the CPU. The single threaded memory utilization is low since in the absence of parallelization, there aren't concurrent and independent memory requests to the main memory.

Configuration	Best runtime observed	Estimated effective memory bandwidth
OMP_THREADS = 48	**7.98ms** (From Fig 1)	**125.3GiB/s** (83.5% of peak)
OMP_THREADS = 1 (one)	**94.19ms** (From Fig 2)	**10.6 GiB/s**

Caveats to above measurement. I ensured that the compile optimization flags and OMP threads are set. However, since this is a lab machine, there maybe other concurrent users on the machine which may hinder the application from reaching peak performance. But given that the laplacian stencil is a short running application, I tried to avoid the effects of background processes by repeating the measurements across iterations and averaging them out for further analysis. Furthermore, in Fig3. I captured measurements with varying levels of parallelization and confirmed the expected linear growth in memory bandwidth since the used memory footprint falls within the peak bandwidth supported by the device.

```
[rajesh@barolo] (39)$ ./laplacian3D_fast
Number of OpenMP threads = 48
Running test iteration  1 [Elapsed time : 86.9332ms]
Running test iteration  2 [Elapsed time : 8.56576ms]
Running test iteration  3 [Elapsed time : 8.10684ms]
Running test iteration  4 [Elapsed time : 7.98034ms]
Running test iteration  5 [Elapsed time : 8.03633ms]
Running test iteration  6 [Elapsed time : 8.05257ms]
Running test iteration  7 [Elapsed time : 8.07493ms]
Running test iteration  8 [Elapsed time : 8.05625ms]
Running test iteration  9 [Elapsed time : 7.98926ms]
Running test iteration 10 [Elapsed time : 9.18732ms]
```

Fig 1. Execution times observed using 48 OMP threads

```
[rajesh@barolo] (116)$ ./laplacian3D
Number of OpenMP threads = 1
Running test iteration  1 [Elapsed time : 293.26ms]
Running test iteration  2 [Elapsed time : 94.4182ms]
Running test iteration  3 [Elapsed time : 94.1945ms]
Running test iteration  4 [Elapsed time : 94.3466ms]
Running test iteration  5 [Elapsed time : 94.5538ms]
Running test iteration  6 [Elapsed time : 94.4614ms]
Running test iteration  7 [Elapsed time : 94.4491ms]
Running test iteration  8 [Elapsed time : 94.4475ms]
Running test iteration  9 [Elapsed time : 94.431ms]
Running test iteration 10 [Elapsed time : 94.4271ms]
```

Fig 1. Execution times observed for sequential execution (single OMP thread)

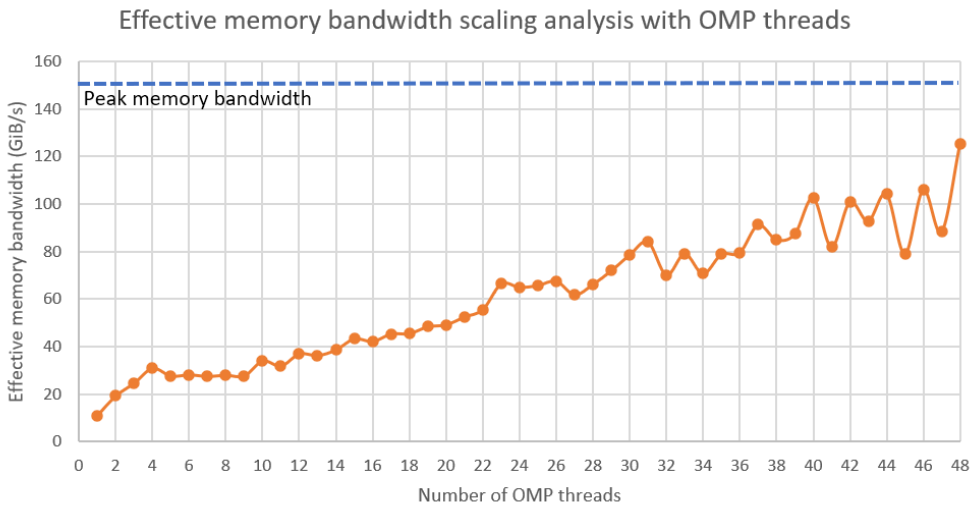

Fig 3. Scaling pattern in memory bandwidth with number of OMP threads

4 Stride access patterns

Re-ordering the loop affects the memory access pattern, cache reuse and locality. To observe this, I tried the following six permutations presented in the table below:

Loop ordering	**Measured best runtime across 10 iterations (with 48 threads)**	**Effective mem. bandwidth**
Baseline (XYZ)	**7.98ms**	**125.3 GiB/s**
ZYX	**584.425ms**	**0.171 GiB/s**
XZY	**66.4745ms**	**1.5 GiB/s**

Assuming row major ordering, the 3D stencil operation operates on nearby cachelines to perform computation that presents an opportunity for data reuse. In the baseline case, there is both temporal and spatial locality of the innermost [k] index, since accesses from subsequent iterations or [k+/-1] within the same iteration are located on a line that has been cached maximizing reuse. Reversing the loop order in case (2) of ZYX above completely breaks this potential reuse in the stencil algorithm making it the worst performing combination.

Furthermore, in the baseline, if we assume each cache line has 16 elements, we can see that the u[k]th index presents an opportunity to have hits to this single cache line at least for 14 iterations with 7 hits per iteration for a total of 98 avoided memory accesses. On the other hand, XZY preserves locality for the [j]th index accesses (innermost loop) but suffers from losing out on the huge locality presented by the last k-index when using row-major ordering for the matrix. This presents an intermediate level of performance compared to the other two orderings.