CS639: Homework 2 (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	*(Makefile included in root directory of zip file)* *g++ *.cpp -Wall -O3 -o conjugate_gradients-fopenmp*
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

Changes to the base code:

- Removed image output
- Added timers for each invocation for every kernel
    - Kernel fusion experiments for the latter part of the assignment presented in section 3

```
54          // Algorithm : Line 6
55          timerLaplacian2.Restart(); ComputeLaplacian(p, z); timerLaplacian2.Pause();
56          timerIP2.Restart(); float sigma=InnerProduct(p, z); timerIP2.Pause();
57
58          // Algorithm : Line 7
59          float alpha=rho/sigma;
60
61          // Algorithm : Line 8
62          timerSaxpy2.Restart(); Saxpy(z, r, r, -alpha); timerSaxpy2.Pause();
63          timerNorm2.Restart(); nu=Norm(r); timerNorm2.Pause();
64
65          // Algorithm : Lines 9-12
66          if (nu < nuMax || k == kMax) {
67              timerSaxpy3.Restart(); Saxpy(p, x, x, alpha); timerSaxpy3.Pause();
68              std::cout << "Conjugate Gradients terminated after " << k << " iterations; residual
                norm (nu) = " << nu << std::endl;
69              if (writeIterations) WriteAsImage("x", x, k, 0, 127);
70              return;
71          }
72
73          // Algorithm : Line 13
74          timerCopy2.Restart(); Copy(r, z); timerCopy2.Pause();
75          timerIP3.Restart(); float rho_new = InnerProduct(z, r); timerIP3.Pause();
```

Excerpt to illustrate the instrumented timers for per-kernel measurements

2 Profiling kernel runtimes through fine-grained instrumentation

	A	B	C	D	E
1	**Invocation count**	**Kernel**	**Line# in algorithm**	**Time (ms) for 256 iter with OMP threads =1**	**Time (ms) for 256 iter with OMP threads =48**
2	1	ComputeLaplacian	2	32.2003	8.73162
3	2	ComputeLaplacian	6	3864.54	1932.02
4	1	Copy	4	25.9018	10.1811
5	2	Copy	13	2991.33	1530.49
6	1	InnerProduct	4	21.1788	4.35304
7	2	InnerProduct	6	5538.1	1412.89
8	3	InnerProduct	13	5627.88	1408.16
9	1	Norm	2	8.36105	3.72889
10	2	Norm	8	2301.29	1110.93
11	1	Saxpy	2	30.2982	29.5458
12	2	Saxpy	8	4618.46	6565.22
13	3	Saxpy	9-12	16.2547	20.3249
14	4	Saxpy	16	4335.75	5473.43
15	5	Saxpy	16	4800.6	7527.73
16	**Total runtime as sum of kernels:**			34212.14485	27037.73535
17	**Total runtime obtained from entire algorithm :**			32608.3	27841.4

Fig 1. Per-kernel runtimes for the kernels in Conjugate Gradient

1. I used LaplaceSolver_0_2 to measure the total runtime of the entire algorithm (row 17). As shown in Fig 1, we see that the sum of kernel runtimes (row 16) is nearly the same as the total algorithm runtime. The minor difference in the actual number can be attributed to two factors (i) Overheads in profiling each kernel i.e. the instrumentation calls themselves can affect the measurements (ii) Statistical variation give that this algorithm runs 256 iterations, it may be susceptible to variations induced by other processes running on system or measurement artifacts that can be overcome by averaging multiple runs.
2. In column D of Fig 1, the colors indicate the kernel runtime distribution in the Conjugate Gradient algorithm. One direct insight from these metrics is that Line 6 and Line 13 have the longest running kernels in the algorithm. This is useful in the kernel fusion in the latter part of the assignment since we will see most gain for the optimization effort when in context. This proves to be a useful tool in performance analysis for narrowing down in larger algorithm parts that can benefit from optimization.
3. Generally, increasing the threads improves the performance measured for all kernels except SAXPY. I suspect two reasons for the lack of scaling with SAXPY: (i) Parallelization via OMP for is not used in the SAXPY kernel (ii) $p = z + \beta p$ on Line 16 (indicated by row 15 in Fig 1) exhibits the worst case performance in the above measurements. This presents a potential bottleneck because multiple threads in flight where cache is the shared resource across all meaning there is more chances of contention through cache misses due to eviction of prior loads & more so because ***p*** is both read-from+written-to. Also, since pointer p is an aliased input to the SAXPY function, it limits compiler optimization.

3 Kernel fusion to improve data reuse in memory-bound applications

The two sets of kernels for fusion were chosen from Line 6 and Line 13 of the algorithm as the kernels associated with these lines seemed to be the most time consuming from the analysis in the previous section. The merged kernel is present in the *kernelFusion* directory of the source code. I ensured that the function remains the same due to these optimizations by verifying the residual norms before and after fusion.

Fig 2 shows the first kernel set of kernels (Laplacian+InnerProduct). Similarly, Fig 4 shows the second set of kernels that were fused (Copy+InnerProduct). The fusion primarily aids in data reuse by merging the parallel for-loop structures that share the same operands. The performance gain is due to the fact that we are amortizing the memory access cost by improving the operations carried out per byte. We can see the results of the kernel fusions in Fig 3 and Fig 5 to be around 1.41x and 1.68x respectively for the multi-threaded execution.

```
// Algorithm : Line 6
// timerLaplacian2.Restart(); ComputeLaplacian(p, z); timerLaplacian2.Pause();
// timerIP2.Restart(); float sigma=InnerProduct(p, z); timerIP2.Pause();
timerLaplacian2.Restart();float sigma= MergedLaplacian(p, z); timerLaplacian2.Pause();
```

Fig 2: Merged kernel 1

	A	B	C	D
1	**Kernel**	**Line# in algorithm**	**Time (ms) for 256 iter with OMP threads =1**	**Time (ms) for 256 iter with OMP threads =48**
2	ComputeLaplacian	6	3864.54	1932.02
3	InnerProduct	6	5538.1	1412.89
4		**Total runtime:**	9402.64	3344.91
5		**Merged kernel runtime:**	10872.7	2365.88
6		**Speed up with merged:**	0.86x	1.41x

Fig 3: Multi-threaded performance gain from kernel fusion presented in Fig 2

```
// Algorithm : Line 13
// timerCopy2.Restart(); Copy(r, z); timerCopy2.Pause();
// timerIP3.Restart(); float rho_new = InnerProduct(z, r); timerIP3.Pause();
timerCopy2.Restart(); float rho_new = MergedCopyIP(r, z); timerCopy2.Pause();
```

Fig 4: Merged kernel 2

	A	B	C	D
1	**Kernel**	**Line# in algorithm**	**Time (ms) for 256 iter with OMP threads =1**	**Time (ms) for 256 iter with OMP threads =48**
2	Copy	13	2991.33	1530.49
3	InnerProduct	13	5627.88	1408.16
4		**Total runtime of 2 kernels:**	8619.21	2938.65
5		**Merged kernel runtime:**	6779.76	1751.25
6		**Speed up with merged (C4/C5):**	1.27x	1.68x

Fig 5: Multi-threaded performance gain from kernel fusion presented in Fig 4

```
float MergedLaplacian(const float (&u)[XDIM][YDIM][ZDIM], float (&Lu)[XDIM][YDIM][ZDIM])
{

double result = 0.;
#pragma omp parallel for reduction(+:result)
    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
    for (int k = 1; k < ZDIM-1; k++){
        // p-u,z-Lu
        Lu[i][j][k] =
            -6 * u[i][j][k]
            + u[i+1][j][k]
            + u[i-1][j][k]
            + u[i][j+1][k]
            + u[i][j-1][k]
            + u[i][j][k+1]
            + u[i][j][k-1];

        // p-x, z-y
        result += (double) u[i][j][k] * (double) Lu[i][j][k];
    }

    return (float) result;
}
```

Fig 6: The merged kernel 1 corresponding to Fig 2/ Line 6 of algorithm

```
float MergedCopyIP(const float (&x)[XDIM][YDIM][ZDIM], float (&y)[XDIM][YDIM][ZDIM])
{
    double result = 0.;
#pragma omp parallel for reduction(+:result)
    for (int i = 1; i < XDIM-1; i++)
    for (int j = 1; j < YDIM-1; j++)
    for (int k = 1; k < ZDIM-1; k++){
        // r-x,z-y
        y[i][j][k] = x[i][j][k];
        // z-x,r-y
        result += (double) y[i][j][k] * (double) x[i][j][k];
    }
    return (float) result;
}
```

Fig 7: The merged kernel 2 corresponding to Fig 4/ Line 13 of algorithm