Analyzing Performance in Stencil Operations using the Sobel Filter

Ben Klein
Computer Science Department
San Francisco State University
San Francisco, CA, USA
Email: bklein@sfsu.edu

***Abstract*—This study looks at three implementations of the Sobel edge detection algorithm to analyze GPU versus CPU parallel and serial performance. The codes analyzed are a OpenMP CPU implementation, a CUDA GPU implementation, and a OpenMP CPU version that offloads computation onto a GPU. All implementations are tested on Perlmutters A100 GPU nodes, using an image of a zebra. The results show that higher thread counts and higher block counts result in better performance in Sobel edge detection, although increasing the block counts tends to result in better performance and utilization of a GPU quicker than increasing a thread count.**

I. Introduction

The Sobel edge detection algorithm is a 3×3 stencil operation that finds edges. The algorithm is simple, but represents a common problem in image processing: stencil operations. Comparing its performance on different implementations shows how parallelism and GPU usage affect runtimes.

Three versions of the Sobel filter are studied in this paper: an OpenMP CPU version, a CUDA GPU version, and an OpenMP offload version that runs on the GPU. Each version implements the Sobel filter logic but uses a different method of parallel computing. The experiments were ran on Perlmutter's A100 GPU nodes on an image of a zebra, with dimensions 7112×5146.

The CUDA GPU implementation resulted in the best performance, given the correct block and thread count. Overall the results show that GPU devices can result in very good performance in parallel code.

II. Implementation

The Sobel operation is a 3×3 stencil algorithm that detects edges. It multiplies each pixel area elementwise with a 3×3 filter to detect both vertical and horizontal edges. The results can be seen in Fig. 1.

Fig. 1. The Sobel filter applied to a zebra.

The specifics of the implementations is detailed in the following subsections. This study shows three implementations: the Sobel filter on a CPU node with OpenMP parallelism, Sobel filter on a GPU using CUDA, and Sobel filter with OpenMP offloading to a GPU.

A. Sobel Filter on CPU with OpenMP Parallelism

This version runs the 3×3 Sobel stencil on the CPU using OpenMP. The nested loops are collapsed using OpenMP so that all pixels are processed concurrently in separate threads. Each pixel computes its nearby edges by multiplying its neighbors by a Sobel filter, summing the results and combining them to get an 'edge magnitude'. The implementation can be seen in Listing 1.

B. Sobel Filter on the GPU using CUDA

This implementation is a GPU version of the Sobel filter. Each GPU thread computes one or more output pixels using the same 3×3 Sobel stencil. The thread

```
#pragma omp parallel for collapse(2)
for each row i in image:
    for each column j in image:
        sum_x = 0
        sum_y = 0
        for each neighbor (x, y) in 3x3 window around (i,j):
            pix = image[i+y, j+x] // clamp at edges
            sum_x += Sobel_x[y,x] * pix
            sum_y += Sobel_y[y,x] * pix
        out[i,j] = sqrt(sum_x^2 + sum_y^2)
```

Listing 1. CPU OpenMP Sobel Filter

indices and stride are computed using the CUDA built in variables. The implementation can be seen in Listing 2.

```
for each thread in GPU grid:
    for each pixel (i,j):
        sum_x = 0
        sum_y = 0
        for each neighbor (x, y) in 3x3 window around (i,j):
            pix = image[i+y, j+x] // clamp at edges
            sum_x += Sobel_x[y,x] * pix
            sum_y += Sobel_y[y,x] * pix
        out[i,j] = sqrt(sum_x^2 + sum_y^2)
```

Listing 2. CUDA Sobel Filter

C. Sobel Filter with OpenMP offload to the GPU

This version is also on the GPU but uses OpenMP's parallelization tools to offload the computation onto the GPU. The details can be seen in Listing 3.

III. Results

A. Computational Platform and Software Environment

The performance studies were conducted on a CPU node of the Perlmutter system at NERSC. [1] Each node has two AMD EPYC 7763 processors running at 2.45 GHz. Each core has a 32 KB L1 cache (with a 64-byte cache line size) and a 512 KB L2 cache, while all eight cores in a CCD share a 32 MB L3 cache. The node has 512 GB of memory, with a measured peak memory bandwidth of 204.8 GB/s.

The software environment consisted of the GNU C++ compiler (GCC 13.2.1) on SUSE Linux Enterprise Server 15 SP5. The build system used `cmake` version 3.20.4 and `make` version 4.2.1. The OpenMP code used static scheduling. Due to compiler challenges, the cmake build type used was "None" rather than "Release".

```
#pragma omp target data map(to: in[0:nvals]) map(to: width) map(to: height) map(to: Gx[0:9]) map(to: Gy[0:9]) map(from: out[0:nvals])
{
#pragma omp target teams distribute parallel for
  for (int index = 0; index < nvals; index++)
    {
      int i = index / width
      int j = index % width
      sum_x = 0
      sum_y = 0
      for each neighbor (x, y) in 3x3 window around (i,j):
          pix = in[i+y, j+x]  // clamp at edges
          sum_x += Sobel_x[y,x] * pix
          sum_y += Sobel_y[y,x] * pix
      out[i,j] = sqrt(sum_x^2 + sum_y^2)
  }
}
```

Listing 3. OpenMP Offload Sobel

B. Methodology

This study uses three metrics to evaluate the performance of the implementations: runtime, occupancy, and percentage of peak memory bandwidth.

CPU runtimes were measured with a C++ standard library `std::chrono` timer around the Sobel loop.

GPU kernel runtimes and hardware metrics were collected with NVIDIA Nsight Compute (`ncu`) [2]. Occupancy represents what percentage of the GPU's resources were utilized. Percentage of peak memory bandwidth represents how efficiently the algorithm utilized GPU memory relative to the potential bandwidth of the device.

All tests were ran on an image of a zebra, with dimensions: 7112×5146 pixels.

C. Scaling study of CPU/OpenMP code

The data shown in Table I shows the scaling of the CPU Sobel implementation in $1, 2, 4, 8$, and 16 threads. The data show the runtime decreasing dramatically as threads increase, reaching almost peak theoretical speedup on most thread counts. The speedup isn't quite as good at higher thread counts, likely due to inefficiencies in cache usage or memory utilization.

D. GPU-CUDA Performance Study

This study shows data gathered over multiple runs in Figures 2, 3, and 4. Unsurprisingly, the runs with low thread counts and low block counts show high runtime, low occupancy, and low DRAM throughput. One notable

TABLE I
CPU OPENMP SCALING FOR THE SOBEL FILTER

Threads	Runtime (s)	Speedup
1	0.2845	1.00
2	0.1435	1.98
4	0.0733	3.88
8	0.0378	7.53
16	0.0226	12.59

finding is that both occupancy and DRAM throughput tend to peak faster with higher block counts rather than higher thread counts.

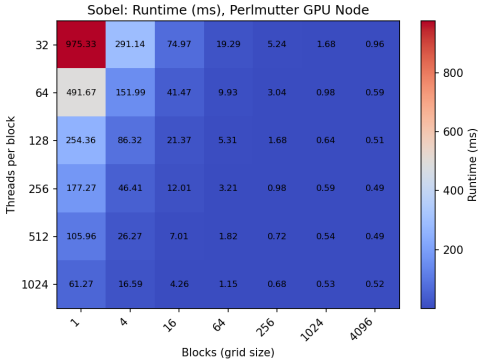

Fig. 2. CUDA Sobel runtime (ms)

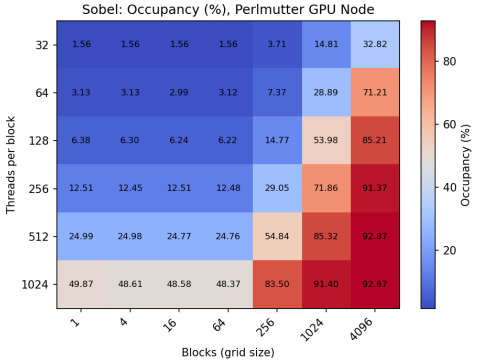

Fig. 3. Achieved Occupancy (%) over all runs.

E. OpenMP-Offload Study

Table II compares the OpenMP offload implementation with the best performing CUDA run (4096 blocks and 256 threads). Both runs have similar times, but CUDA squeezes a little more out of the GPU, achieving higher occupancy and memory throughput.

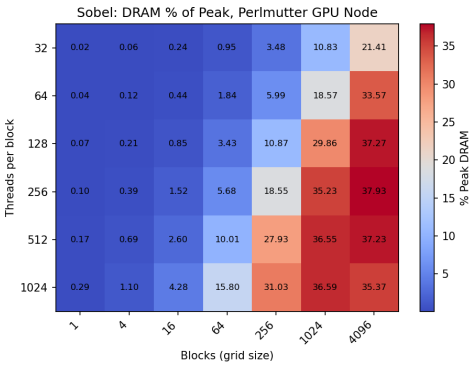

Fig. 4. DRAM throughput as a percentage of peak sustained bandwidth.

TABLE II
COMPARISON OF OPENMP GPU OFFLOAD VS. BEST CUDA CONFIGURATION

Method	Runtime (ms)	Occupancy (%)	DRAM % Peak
CUDA (4096×256)	0.49	91.4	37.9
OpenMP GPU Offload	0.62	66.8	29.5

ACKNOWLEDGEMENT

The author acknowledges the use of OpenAI's ChatGPT for data management, graphing and LaTeX formatting assistance.

REFERENCES

[1] "Architecture - NERSC Documentation." [Online]. Available: https://docs.nersc.gov/systems/perlmutter/architecture/

[2] "2. Profiling Guide — NsightCompute 13.0 documentation." [Online]. Available: https://docs.nvidia.com/nsight-compute/ProfilingGuide/index.html#sections-and-rules