ASPP Coursework 1 Report

Exam number: `B281684`

1. A100 performance estimate (site updates/s)

A *site update* is one time-step update for one grid point (i, j, k) in `step()` (`src/wave_cpu.cpp`). The scheme uses a 7-point Laplacian and a damping term in the x/y boundary layers:

$$\Delta u \approx u_{i-1,j,k} + u_{i+1,j,k} + u_{i,j-1,k} + u_{i,j+1,k} + u_{i,j,k-1} + u_{i,j,k+1} - 6u_{i,j,k}, \tag{1}$$

$$u^{n+1} = 2u^n - u^{n-1} + \left(\frac{\Delta t^2}{\Delta x^2}\right) c^2 \Delta u \quad (d = 0), \tag{2}$$

$$u^{n+1} = \frac{2u^n - (1 - d\Delta t)u^{n-1} + \left(\frac{\Delta t^2}{\Delta x^2}\right) c^2 \Delta u}{1 + d\Delta t} \quad (d > 0). \tag{3}$$

Most grid points are in the bulk where $d = 0$ (damping is only a thin layer in x and y).

Per-site cost. For the bulk update ($d = 0$), the kernel costs about 12 FLOPs/site. Damped points add a few extra FLOPs and a divide, but for $n_{\text{bl}} = 4$ the undamped fraction is $\frac{(n_x-8)(n_y-8)}{n_x n_y}$ (about 0.56 for 32^3 and 0.94 for 256^3).

Arithmetic intensity. $I = \frac{\text{FLOPs}}{\text{Bytes}}$. A pessimistic DRAM model is that each site reads 10 doubles and writes 1 double:

$$B_{\text{naive}} \approx (10 \text{ loads} + 1 \text{ store}) \cdot 8 \approx 88 \text{ B/site}, \qquad I_{\text{naive}} \approx \frac{12}{88} \approx 0.14 \text{ FLOP/B}. \tag{4}$$

For A100, the ridge point is $I_{\text{ridge}} = \frac{P_{\text{FP64}}}{B_{\text{HBM}}} \approx \frac{9.7\times10^{12}}{1.555\times10^{12}} \approx 6.2$ FLOP/B [1]. Since $I_{\text{naive}} \ll I_{\text{ridge}}$, performance is limited by HBM bandwidth.

The 88 B/site model is pessimistic because the stencil has strong spatial reuse and the coefficient arrays are small (c^2 depends only on k; d depends only on (i, j)). With cache reuse and coefficient compression, the effective DRAM traffic can approach $B_{\text{opt}} \approx 24$ to 32 B/site, giving $I_{\text{opt}} \approx 0.38$–$0.50$ FLOP/B (still $\ll I_{\text{ridge}}$).

Roofline limits. Using $P_{\text{FP64}} \approx 9.7 \times 10^{12}$ FLOP/s and $B_{\text{HBM}} \approx 1.555 \times 10^{12}$ B/s [1], the roofline ceilings are:

$$\text{SU/s}_{\text{compute}} \approx \frac{P_{\text{FP64}}}{12} \approx \frac{9.7 \times 10^{12}}{12} \approx 8.1 \times 10^{11}, \tag{5}$$

$$\text{SU/s}_{\text{BW,naive}} \approx \frac{B_{\text{HBM}}}{B_{\text{naive}}} \approx \frac{1.555 \times 10^{12}}{88} \approx 1.8 \times 10^{10}, \tag{6}$$

$$\text{SU/s}_{\text{BW,opt}} \approx \left[\frac{B_{\text{HBM}}}{32}, \frac{B_{\text{HBM}}}{24}\right] \approx (4.9\text{–}6.5) \times 10^{10}. \tag{7}$$

This work therefore use $\text{SU/s}_{\text{max}} \approx 6 \times 10^{10}$ as the practical theoretical upper limit on one A100 for large problems; small problems will fall below this due to launch/offload overhead.

2. Implementation choices and performance evidence (32–1000)

2.1 Setup and metric

The program runs CPU reference, then CUDA, then OpenMP offload, and checks both GPU results against the CPU with tolerance $\epsilon = 10^{-8}$ (`src/main.cpp`). All results reported here had `Number of differences detected = 0`. Performance is reported as mean site updates per second (SU/s) over timed chunks of `run()`. File output is excluded from timing because `append_u_fields()` runs after each timed chunk.

$$\text{SU/s} = \frac{n_x n_y n_z \cdot n_{\text{steps}}}{t_{\text{chunk}}}. \tag{8}$$

GPU tests used one full `NVIDIA-A100-SXM4-40GB`. For OpenMP, `OMP_TARGET_OFFLOAD=MANDATORY` was set to avoid silent fallback. Unless stated otherwise, runs used the default `-dx 10` and `-dt 0.002` and $n_{\mathrm{bl}} = 4$ (fixed in `src/main.cpp`). Table 1 uses `AWAVE_KERNEL_MODE=1`. To reduce noise while keeping output volume small, I used `-nsteps 200 -out_period 100` for $L \leq 96$, and `-nsteps 20 -out_period 10` otherwise (two timed chunks in both cases). The code was built with CMake `Release` on EIDF (NVHPC toolchain).

2.2 Key choices and reasons

- **Device-resident fields + pointer rotation:** `u_prev/u_now/u_next` stay on the device for the whole run (`cudaMalloc` or `omp_target_alloc`). Each step writes `u_next` and then swaps pointers, so there is no per-step device memcpy. For OpenMP, `is_device_ptr` is used so the offload regions use the existing device pointers rather than remapping arrays each step. **This concentrate the overhead of each step on the stencil kernel to avoid turning PCIe/D2H/H2D into a bottleneck.**
- **Compressed coefficients:** in the initial conditions, $c^2(i, j, k)$ depends only on k, and $d(i, j, k)$ depends only on (i, j) (`src/wave_cpu.cpp`). Both GPU versions store `cs2_k[k]` and `damp_xy[i, j]` on device. For $1000 \times 64 \times 1000$, `cs2_k` is 1000 doubles ($\approx 8\,\mathrm{kB}$) and `damp_xy` is $1000 \cdot 64$ doubles ($\approx 0.5\,\mathrm{MB}$). **This makes the coefficient arrays small and reduce the DRAM access pressure. And It is easy to reside on L2, thus allocating more bandwidth budget to the three *u* fields.**
- **Launch mapping:** CUDA uses a 3D grid with `k` mapped to `threadIdx.x` (block $(32, 4, 2)$), which matches the contiguous `k` dimension in memory and gives coalesced loads for the ± 1 neighbors. OpenMP uses `teams distribute parallel for collapse(3)` with the same linear indexing. **If copyback falls within the timing zone, it will seriously "mask" the actual computing performance.**
- **Timing fairness:** both implementations run one warm-up kernel/offload before the first timed chunk. Device-to-host copies are done in `append_u_fields()` for output and checking, so they stay outside the timed `run()` region (matching the driver which excludes I/O from timing). **Reduce the mean/std deviation caused by the slow timing of the first section and improve the credibility and reproducibility of the data in the report.**
- **Kernel decomposition sweep:** both implementations support `AWAVE_KERNEL_MODE` $\in \{1, 2, 3\}$ to split interior/boundary work. On the A100 size sweep (32 to 1000), mode 1 (single kernel/offload with a damping branch) was fastest in 14/15 shapes for CUDA and 15/15 for OpenMP. Mode 3 only won for 384^3 on CUDA by about 2% (see detail in Appendix B). **This work used mode 1 as the default because the branch only affects a small boundary region, while extra launches increase overhead, especially for small and mid-size problems.** In mode 1, the damping branch depends only on (i, j) while threads in a warp vary in k (CUDA block $(32, 4, 2)$), so the branch is warp-uniform; only boundary warps do the extra damping arithmetic, which is cheaper than launching additional kernels/offloads.

2.3 Results

To cover sizes up to 1000 without very large outputs, this work used cubes up to 384^3, and then rectangular shapes $(L, 64, L)$ up to $L = 1000$. Kept $n_x = n_z$ because the checker in `src/main.cpp` assumes this for the k loop bound.

Shape	CPU SU/s	CUDA SU/s	OpenMP SU/s	OpenMP/CUDA
32^3	3.45×10^8	6.54×10^9	2.31×10^9	0.35
64^3	3.37×10^8	3.04×10^{10}	1.42×10^{10}	0.47
128^3	3.03×10^8	4.64×10^{10}	3.52×10^{10}	0.76
256^3	2.90×10^8	5.21×10^{10}	4.76×10^{10}	0.91
384^3	2.68×10^8	5.26×10^{10}	4.95×10^{10}	0.94
$512 \times 64 \times 512$	3.00×10^8	5.10×10^{10}	4.66×10^{10}	0.91
$1000 \times 64 \times 1000$	2.72×10^8	5.32×10^{10}	4.91×10^{10}	0.92

Table 1: Mean SU/s on one A100 (mode 1).

Trends:

- **Small sizes** (32^3 to 96^3) are limited by kernel/offload overhead. CUDA is faster because the launch path is lighter than OpenMP target offload.
- **Large sizes** ($\geq 128^3$) show bandwidth saturation. CUDA stays around 4.6 to 5.3×10^{10} SU/s and OpenMP around 3.5 to 4.9×10^{10} SU/s across the tested range.
- **Speedup vs CPU:** for the large cases in Table 1, CUDA is about $1.8\text{–}2.0 \times 10^2\times$ faster than the serial CPU, and OpenMP offload is about $1.6\text{–}1.8 \times 10^2\times$ faster.
- **Comparison to Section 1:** the best CUDA result is 5.32×10^{10} SU/s, about 0.89 of the estimated 6×10^{10} SU/s limit. **Nsight Compute** roofline for CUDA mode 1 `step_kernel` at shape 256^3 reports achieved DRAM bandwidth $\approx 1.37\,\text{TB/s}$ and achieved FP64 throughput $\sim 9\%$ of peak. Using the measured CUDA SU/s 5.21×10^{10}, the effective DRAM traffic is $\frac{1.37 \times 10^{12}}{5.21 \times 10^{10}} \approx 26\,\text{B/site}$, which matches the 24 to 32 B/site model in Section 1.

2.4 Extra validation and profiling notes

Extra validation (all with `diff` $= 0$)(see detail in Appendix A):

- Non-cubic shapes to check there is no dependence on a special geometry: $256 \times 96 \times 256$ gave 4.91×10^{10} (CUDA) and 4.22×10^{10} (OpenMP) SU/s; 333^3 gave 5.25×10^{10} (CUDA) and 4.87×10^{10} (OpenMP) SU/s.
- Changed resolution while keeping dt/dx constant (to match the expected CFL scaling): at 256^3, $(dx, dt) = (5, 0.001)$ and $(20, 0.004)$ both matched the CPU reference and kept SU/s within the same bandwidth-saturated range.

Profiling observations (Nsight Systems/Compute): for 32^3, the CUDA step kernel was about $3.8\,\mu$s per step and the OpenMP offload kernel about $4.4\,\mu$s, so extra launches in modes 2/3 reduce performance. For a large case ($1000 \times 64 \times 1000$), the `run` NVTX range was only $\sim 2.6\text{–}3.1$ ms per chunk (two steps), but the `copyback` range was $\sim 133\text{–}149$ ms per chunk. This is why copyback is kept out of `run()` timings, consistent with the coursework driver.

3. CUDA vs OpenMP offload: performance, effort, recommendation

Performance:

- CUDA was faster for every tested shape. The gap was largest for small problems: at 32^3, OpenMP reached $0.35\times$ the CUDA SU/s. This matches the profiling results in Section 2.4 (the kernel body is only a few microseconds, so launch/offload overhead is significant).
- For larger problems ($\geq 128^3$), both implementations saturate memory bandwidth. OpenMP typically reached 0.88 to $0.94\times$ the CUDA SU/s (Table 1), so the remaining gap is mainly runtime and code-generation details.

Ease of implementation and control:

- OpenMP target offload kept the loop structure close to the CPU code. However, to get stable performance we still needed explicit device allocation (`omp_target_alloc`), explicit copies for initialisation/output, and `is_device_ptr` to avoid implicit remapping overhead.
- CUDA required more boilerplate but gave direct control of key performance choices: explicit stream usage, asynchronous copies for initialisation/copyback, and an explicit thread-block shape $((32, 4, 2))$ matched to the contiguous `k` dimension. This helped reduce overhead at small sizes while keeping bandwidth-saturated performance at large sizes [2].

If I could only pick one model for this task, I would choose **CUDA**. It gives higher performance across the full size range, and it is easier to control overhead for small problems while still approaching the bandwidth limit for large problems. OpenMP offload is still competitive for large stencil problems and would be a good choice if portability was the main goal.

References

[1] NVIDIA. *NVIDIA A100 Tensor Core GPU Datasheet* (SXM4 40GB). 2020. URL: `https://www.nvidia.com/en-us/data-center/a100/`. Accessed: 2026-02-07.

[2] NVIDIA. *CUDA C++ Programming Guide* (CUDA Toolkit Documentation). URL: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`. Accessed: 2026-02-07.

Appendix A: Full 32–1000 performance results

Table 2 reports the full sweep results discussed in Sections 2–3 (A100 full GPU, mean SU/s, mode 1).

Shape	CPU SU/s	CUDA SU/s	OpenMP SU/s	OpenMP/CUDA
32^3	3.45×10^8	6.54×10^9	2.31×10^9	0.35
64^3	3.37×10^8	3.04×10^{10}	1.42×10^{10}	0.47
96^3	3.36×10^8	5.19×10^{10}	2.85×10^{10}	0.55
128^3	3.03×10^8	4.64×10^{10}	3.52×10^{10}	0.76
160^3	2.84×10^8	4.80×10^{10}	4.06×10^{10}	0.85
192^3	2.87×10^8	4.94×10^{10}	4.34×10^{10}	0.88
224^3	2.83×10^8	5.13×10^{10}	4.62×10^{10}	0.90
256^3	2.90×10^8	5.21×10^{10}	4.76×10^{10}	0.91
$256 \times 96 \times 256$	2.97×10^8	4.91×10^{10}	4.22×10^{10}	0.86
333^3	2.96×10^8	5.25×10^{10}	4.87×10^{10}	0.93
384^3	2.68×10^8	5.26×10^{10}	4.95×10^{10}	0.94
$512 \times 64 \times 512$	3.00×10^8	5.10×10^{10}	4.66×10^{10}	0.91
$512 \times 128 \times 512$	2.93×10^8	5.26×10^{10}	4.84×10^{10}	0.92
$640 \times 64 \times 640$	2.74×10^8	5.20×10^{10}	4.77×10^{10}	0.92
$768 \times 64 \times 768$	2.74×10^8	5.27×10^{10}	4.70×10^{10}	0.89
$896 \times 64 \times 896$	2.72×10^8	5.30×10^{10}	4.89×10^{10}	0.92
$1000 \times 64 \times 1000$	2.72×10^8	5.32×10^{10}	4.91×10^{10}	0.92

Table 2: Full 32–1000 sweep results (mean SU/s, mode 1).

Appendix B: Full results by kernel mode

Table 3 reports the full sweep results across all tested shapes and kernel modes (A100 full GPU, mean SU/s).

Shape	Mode	CPU SU/s	CUDA SU/s	OpenMP SU/s	OpenMP/CUDA
32^3	1	3.45×10^8	6.54×10^9	2.31×10^9	0.35
32^3	2	3.29×10^8	3.63×10^9	1.18×10^9	0.32
32^3	3	2.98×10^8	2.60×10^9	7.96×10^8	0.31
64^3	1	3.37×10^8	3.04×10^{10}	1.42×10^{10}	0.47
64^3	2	3.40×10^8	2.08×10^{10}	8.02×10^9	0.39
64^3	3	3.43×10^8	1.63×10^{10}	5.73×10^9	0.35
96^3	1	3.36×10^8	5.19×10^{10}	2.85×10^{10}	0.55
96^3	2	3.60×10^8	4.09×10^{10}	1.97×10^{10}	0.48
96^3	3	3.78×10^8	3.53×10^{10}	1.54×10^{10}	0.44
128^3	1	3.03×10^8	4.64×10^{10}	3.52×10^{10}	0.76
128^3	2	3.05×10^8	4.26×10^{10}	2.67×10^{10}	0.63
128^3	3	3.17×10^8	3.96×10^{10}	2.45×10^{10}	0.62
160^3	1	2.84×10^8	4.80×10^{10}	4.06×10^{10}	0.85
160^3	2	3.03×10^8	4.60×10^{10}	3.24×10^{10}	0.70
160^3	3	2.86×10^8	4.43×10^{10}	3.17×10^{10}	0.72
192^3	1	2.87×10^8	4.94×10^{10}	4.34×10^{10}	0.88
192^3	2	2.81×10^8	4.81×10^{10}	3.61×10^{10}	0.75
192^3	3	2.96×10^8	4.71×10^{10}	3.76×10^{10}	0.80
224^3	1	2.83×10^8	5.13×10^{10}	4.62×10^{10}	0.90
224^3	2	2.94×10^8	5.02×10^{10}	3.90×10^{10}	0.78
224^3	3	2.90×10^8	4.98×10^{10}	4.18×10^{10}	0.84
256^3	1	2.90×10^8	5.21×10^{10}	4.76×10^{10}	0.91
256^3	2	2.99×10^8	5.15×10^{10}	4.08×10^{10}	0.79
256^3	3	2.73×10^8	5.15×10^{10}	4.51×10^{10}	0.88
320^3	1	1.30×10^8	5.27×10^{10}	4.86×10^{10}	0.92
320^3	2	1.29×10^8	5.22×10^{10}	3.74×10^{10}	0.72
320^3	3	1.25×10^8	5.26×10^{10}	4.76×10^{10}	0.90
384^3	1	2.68×10^8	5.26×10^{10}	4.95×10^{10}	0.94
384^3	2	2.69×10^8	5.30×10^{10}	4.34×10^{10}	0.82
384^3	3	2.96×10^8	5.37×10^{10}	4.95×10^{10}	0.92
$512 \times 64 \times 512$	1	3.00×10^8	5.10×10^{10}	4.66×10^{10}	0.91
$512 \times 64 \times 512$	2	2.96×10^8	5.02×10^{10}	3.95×10^{10}	0.79
$512 \times 64 \times 512$	3	3.04×10^8	4.87×10^{10}	4.27×10^{10}	0.88
$640 \times 64 \times 640$	1	2.74×10^8	5.20×10^{10}	4.77×10^{10}	0.92
$640 \times 64 \times 640$	2	2.56×10^8	5.14×10^{10}	4.06×10^{10}	0.79
$640 \times 64 \times 640$	3	2.82×10^8	5.02×10^{10}	4.49×10^{10}	0.89
$768 \times 64 \times 768$	1	2.74×10^8	5.27×10^{10}	4.70×10^{10}	0.89
$768 \times 64 \times 768$	2	2.65×10^8	5.22×10^{10}	4.15×10^{10}	0.79
$768 \times 64 \times 768$	3	2.58×10^8	5.14×10^{10}	4.68×10^{10}	0.91
$896 \times 64 \times 896$	1	2.72×10^8	5.30×10^{10}	4.89×10^{10}	0.92
$896 \times 64 \times 896$	2	2.71×10^8	5.28×10^{10}	4.21×10^{10}	0.80
$896 \times 64 \times 896$	3	2.72×10^8	5.10×10^{10}	4.69×10^{10}	0.92
$1000 \times 64 \times 1000$	1	2.72×10^8	5.32×10^{10}	4.91×10^{10}	0.92
$1000 \times 64 \times 1000$	2	2.74×10^8	5.28×10^{10}	3.84×10^{10}	0.73
$1000 \times 64 \times 1000$	3	2.77×10^8	5.25×10^{10}	4.83×10^{10}	0.92

Table 3: Full sweep results by kernel mode (mean SU/s). Mode 1: Global single kernel/offload (Each point branch determines whether it is damped); Mode 2: Two segments (X-boundary alone; the rest merged); Mode 3: Three segments (inner domain x boundary and y boundary).