High-level, Mid-level, and Low-level GPU Programming Comparison

Marco Julian Solanki, Thibault Meier, Sebastian Heckers

August 31, 2024

Abstract

This report aims to investigate the performance characteristics of different programming models targeting Nvidia GPUs. The considered programming models encompass high-level (CuPy & GT4Py), mid-level (OpenACC), and low-level (CUDA) approaches. The performance of the different models is evaluated by considering the performance of various incarnations of the stencil2d miniapp, which solves a fourth-order diffusion partial differential equation under periodic boundary conditions. Benchmarking is performed on the Tesla P100-based GPU nodes of the Piz Daint supercomputer, as well as a GeForce GTX 1070-based local machine with a more up-to-date software stack.

1 Introduction

The recent slowing of Moore's law has motivated the widespread adoption of specialised hardware, in particular GPUs, in scientific computing. In contrast to CPUs, which sport core counts up to the low 100s, GPUs typically exhibit core counts from the 1,000s well into the 10,000s. This, in combination with the higher bandwidth memory GPUs tend to be equipped with, tends to give them an edge in highly parallel applications, such as the modelling of partial differential equations. At the same time, the highly parallel and generally complex hardware design of GPUs has posed a challenge for scientists seeking to leverage them. To tackle this challenge, several programming models of various degrees of complexity have arisen over the last two decades. This report aims to investigate the performance characteristics a selection of these exhibits.

1.1 Code overview

As representatives for high-level approaches, the CuPy Python library [OUN+17] and the GT4Py Python framework [PGU+23] are considered. The corresponding mid- and low-level programming models that are explored are OpenACC directives [The22] and Nvidia's CUDA runtime API [Nvi24]. Their performance is compared through a series of benchmarks centred around the sencil2d miniapp, which has been ported to each of these programming models. Details concerning the physics solved by the stencil2d miniapp are provided in section 1.3.

1.2 A brief literature review

A similar comparison between CUDA and OpenACC, also covering stencil applications, is provided by [HMMT13]. This paper concludes that while OpenACC may indeed offer ease of use and portability, it falls short in terms of performance compared to CUDA, especially for applications requiring fine-grained optimisations and efficient memory access (most notably, applications that benefit from the usage of shared memory). It must be noted, however, that this paper is already over a decade old, with presumably many advancements both in hardware and software having occurred in the interim. Instead of Fermi-generation Tesla M2050s and a software stack representative of around that era, this report will rely upon two Pascal-generation GPUs, a Tesla P100 and a GeForce GTX 1070, as well as software stacks that should be a lot more representative of the present-day. A comparison with the results from [HMMT13] will nevertheless be made in section 5.3.

1.3 Target problem

The goal of the stencil2d miniapp is to solve the fourth-order diffusion equation, as is presented by [Xue00], on various (x, y)-planes equispaced in z-direction within a cuboid domain:

$$u_t + \alpha_4 \nabla^4_{(x,y)} u = 0 \tag{1}$$

The existence and uniqueness of the solution $u := u(x, y, z, t)$ is ensured through the inclusion of initial as well as periodic boundary conditions in x- and

y-directions. Defining the finite-difference solution $u_h^{(n)}$ at time $t := n\Delta t$ and applying first-order forward finite differences in time (explicit Euler) yields:

$$u_h^{(n+1)} = u_h^{(n)} - \alpha_4 \Delta t \cdot \nabla^4_{(x,y)} u_h^{(n)} \tag{2}$$

In the following, $\Delta x := \Delta y := 1$ and $z := k\Delta z$, where k is chosen arbitrarily but fixed (and is therefore omitted from the following localised representations for $u_h^{(n)}$). $\nabla^4_{(x,y)} u_h^{(n)}$ can be evaluated either by sequentially applying two second-order 5-point Laplacian stencils (herein abbreviated as `laplap`):

$$\begin{aligned}\nabla^2_{(x,y)} u_{i,j}^{(n)} = -4u_{i,j}^{(n)} + u_{i+1,j}^{(n)} + u_{i,j+1}^{(n)} \\ + u_{i-1,j}^{(n)} + u_{i,j-1}^{(n)}\end{aligned} \tag{3}$$

...or, equivalently, by applying a single second-order 13-point biharmonic stencil (herein abbreviated as `biharm`):

$$\begin{aligned}\nabla^4_{(x,y)} u_{i,j}^{(n)} = u_{i+2,j}^{(n)} - 8u_{i+1,j}^{(n)} + 2u_{i+1,j+1}^{(n)} \\ + u_{i,j+2}^{(n)} - 8u_{i,j+1}^{(n)} + 2u_{i-1,j+1}^{(n)} \\ + u_{i-2,j}^{(n)} - 8u_{i-1,j}^{(n)} + 2u_{i-1,j-1}^{(n)} \\ + u_{i,j-2}^{(n)} - 8u_{i,j-1}^{(n)} + 2u_{i+1,j-1}^{(n)} \\ + 20u_{i,j}^{(n)}\end{aligned} \tag{4}$$

The periodic boundary conditions are enforced by adding dedicated boundary points (a.k.a., halo points/ghost cells) at the edges of the domain facing in x- and y-directions. These are synchronised every timestep to match the points located on the opposite end of the interior of the domain.

2 Implementation details

The CUDA and OpenACC implementations referred to in the following were written from scratch around the C++11 standard. CuPy and GT4Py codes were provided but ended up being modified to match the conventions introduced in the CUDA and OpenACC codes (and, in the case of the latter, to be compatible with modern GT4Py releases). In the following, some optimisations and (non-cosmetic) changes applied to these codes are discussed.

2.1 Model-agnostic optimisations

The following optimisations were, at the very least, attempted for all programming models:

- Computations were generally treated as fully parallel in z-direction. The aim of this was to keep the GPU well-utilised even at small domain sizes. This has the side-effect of reducing the total number of required (CUDA) kernel launches by a factor `zsize` but also increasing the required amount of intermediate memory by a factor `zsize`.
- It was generally attempted to keep the total device memory usage limited to approximately `2 * xsize * ysize * zsize * sizeof(realtype)` by fusing the computations in a way as to only require two real arrays housing `xsize * ysize * zsize` elements to be stored at any given step within the computations.

2.2 CUDA

The following potential optimisations were explored for the CUDA version of the miniapp:

- Using page-locked (a.k.a. "pinned") host memory via `cudaMallocHost`. Using such memory allows the CUDA runtime to skip checking whether the array referenced by a given host pointer is indeed located in physical main memory or whether it has been paged out to swap storage.
- Using asynchronous memory management routines such as `cudaMemcpyAsync`, `cudaMallocAsync` and `cudaFreeAsync` (the latter two requiring at least CUDA 11.2). These allow the host thread to continue enqueuing additional work items into the utilised CUDA stream, even while these routines are still being executed (i.e., they are non-blocking from the host's point of view).
- Using shared memory to speed up Laplacian calculations. Shared memory offers high-speed on-chip intermediate storage that is shared within the threads of a thread block. This storage permits reducing the number of global memory accesses each thread has to perform, e.g., while evaluating a Laplacian stencil from 5 to 1 – 2.
- Using the 13-point biharmonic stencil instead of a sequence of two 5-point Laplacian stencils. While there seems to be no strong reason to a priori believe that using a single 13-point stencil is faster than using two 5-point stencils, this still seems worth trying out (both with and without shared memory).
- Using optimised thread block dimensions. The only thread block sizes that seem to

have a measurable impact on overall performance (at least when using thin enough boundaries) are those used for the interior biharmonic/Laplacian evaluations. When tested, the best-performing thread blocks for these interior evaluations are those with dimensions 16x16 ($\rightarrow$ 256 threads per block). Therefore, the benchmarking results presented in the following are all obtained using these thread block dimensions.

- Using `nvc++` instead of `nvcc` as the CUDA compiler. `nvcc` is Nvidia's LLVM-based CUDA compiler that, by default, ships with Nvidia's CUDA toolkit. `nvc++` is based on the old PGI `pgc++` compiler and ships with Nvidia's HPC SDK. Considering their distinctive histories (and, presumably, distinct code bases), it seems worth testing whether one of them yields better-performing binaries than the other.

In terms of error handling, every `cudaError_t` that is returned by some API function is captured and checked. If its value is found to be not equal to `cudaSuccess`, a (descriptive) error message is printed to `stderr`, and the program exits with an `EXIT_FAILURE` return value. Also, for launching kernels, the somewhat lower-level `cudaLaunchKernel` syntax is used, which comes with the benefit of, in contrast to the triple-chevron (<<<>>>) syntax, always returning a `cudaError_t` that can be checked.

2.3 OpenACC

Naively using OpenACC `kernels` pragmas (see `kernels.hpp`) leads to extremely slow code, with simulations typically taking more than 100 times longer than the equivalent CUDA code. An explanation for this behaviour becomes apparent when using `nvc++`'s `-Minfo` flag: `nvc++` claims that there are "complex loop carried dependences" between the iterations of the (nested) interior and boundary loops — even though, in reality, all of these iterations are completely independent and trivially parallelisable.

Thankfully, explicitly annotating the individual (nested) loops with `parallel loop collapse` pragmas (see `parallel.hpp`) yields no friction with `nvc++`, as it simply parallelises the nested loop iterations without complaining.

A direct 13-point biharmonic stencil approach was not explored, considering such an approach performed universally worse when implemented in CUDA (see the results provided in section 5).

For error handling, the somewhat poorly documented `acc_set_error_routine` interface was used. This interface allows for the definition of a callback routine that is called when the OpenACC runtime encounters an error. Once invoked, this callback routine prints a (descriptive) error message and terminates the program with an `EXIT_FAILURE` return value.

2.4 CuPy

Apart from trying to incorporate the optimisations detailed in section 2.1 and adjusting its naming conventions to match those of the CUDA and OpenACC codes, not much was changed in the CuPy version of the stencil2d miniapp relative to the version that is provided on the HPC4WC GitHub page.

2.5 GT4Py

The GT4Py code, as it is provided on the HPC4WC GitHub page, is incompatible with the most recent releases of GT4Py. This seems to be the effect of breaking changes in GT4Py's API between the version of GT4Py installed on Piz Daint (0.1.0) and the latest one available from PyPI (1.0.3). Furthermore, even GT4Py 1.0.3 seems to be incompatible with the latest release of Python (3.12). Therefore, in order to run GT4Py locally, the code had to be adapted, and a previous version of Python (for the purposes of this report: 3.10) had to be installed.

In addition, as an early attempt to rectify the extremely poor performance of GT4Py on Piz Daint (visible in figure 1), it was attempted to rewrite the GT4Py-based variant of the miniapp to use `zyx`-instead of `xyz`-indexing. This, however, led to the problem that GT4Py seemingly would not accept differentiating the intermediate `lap` field ($= \nabla^2_{(x,y)} u_h^{(n)}$) in x-direction, as this would, according to GT4Py, lead to an "invalid access with offset in `k` to temporary field `lap`". This problem could, however, be bypassed by simply applying a single 13-point biharmonic stencil instead of two 5-point Laplacian ones. The resulting implementation is stored in `zyx-biharm.py` while its `xyz`-indexing-based counterpart is provided in `xyz-laplap.py`.

3 Code verification

The different stencil2d miniapp versions and code paths were verified firstly by visual inspection but

also by computing the L^1, L^2 and L^∞ distances between the results produced by them. The resulting point-wise errors/deviations were generally on the order of machine precision. The worst-case L^1/L^2/L^∞ errors/deviations measured in a simple benchmarking scenario are given in table 1. Notably, even the there-listed worst-measured point-wise maximal error/deviation (L^∞) barely exceeds 10^{-14}. This strongly suggests that the outputs by the different backends are indeed consistent and, therefore, probably correct.

Norm	Error/Deviat.	When comparing...
L^1	$4.840 \cdot 10^{-10}$	biharm-$*$ & zyx-biharm
L^2	$1.155 \cdot 10^{-12}$	biharm-$*$ & zyx-biharm
L^∞	$1.021 \cdot 10^{-14}$	*various*

Table 1: Maximum L^1, L^2 and L^∞ errors/deviations observed when making a pairwise comparison of the outputs of all of the different code paths of the stencil2d miniapp for `nx` $=$ `ny` $=$ 128, `nz` $=$ 64, boundary width $=$ 2 and #timesteps $=$ 1024. "biharm-$*$" refers to the CUDA version's `biharm-global` and `biharm-shared` stencil evaluation modes while "zyx-biharm" refers to the `zyx-biharm.py` version of the GT4Py code.

4 Benchmarking methodology

For the purposes of the following benchmarks, the "runtime" of the code is defined to encompass the following:

- Memory transfers from the host to the device.
- Allocations of intermediate device memory.
- On-device computations.
- Memory transfers from the device to the host.

The rationale for defining the runtime in such a fashion is that:

- Including all device-specific operations makes the resulting timings more comparable with CPU-only computations (as the latter obviously do not have to perform, for example, host-to-device memory copies).
- Not all programming models allow for the same level of control over memory, e.g., CUDA offers page-locked host memory allocations via `cudaMallocHost`. Meanwhile, OpenACC, for example, allows for no such fine-grained control. Should such features exclusive to lower-level programming models indeed have a sizable impact on overall performance, this surely ought to be reflected in the benchmarking results presented in this report.

Benchmarking was performed on two different systems, the first one being the GPU nodes of Piz Daint and the second one being Strix, a privately owned machine with an up-to-date software stack. Their respective system specifications are provided in tables 2 and 3.

Name:	**Piz Daint**
CPU:	Intel Xeon E5-2690 v3
GPU:	NVIDIA Tesla P100-PCIE-16GB
Linux distribution:	SLES 15 SP2
Kernel version:	5.3.18-24.102-default
Driver version:	470.57.02
CUDA version:	11.2 (nvcc), 10.2 (nvc++), 11.4 (runtime)
HPC SDK version:	21.3
Python version:	3.9.4
CuPy version:	10.1.0
GT4Py version:	0.1.0

Table 2: Software and hardware configuration of the GPU nodes of Piz Daint.

Name:	**Strix**
CPU:	Intel Core i7-6700HQ
GPU:	NVIDIA GeForce GTX 1070
Linux distribution:	Arch Linux (rolling)
Kernel version:	6.9.7-zen1-1-zen
Driver version:	555.58
CUDA version:	12.5 (nvcc), 12.4 (nvc++), 12.5 (runtime)
HPC SDK version:	24.5
Python version:	3.12.4 (CuPy), 3.10.14 (GT4Py)
CuPy version:	13.2.0
GT4Py version:	1.0.3

Table 3: Software and hardware configuration of Strix.

Each of the benchmarked code paths was run for `nz` $=$ 64, `bdry` $=$ 2 (boundary width) and `itrs` $=$ 1024 (#timestepping iterations). `nx` and `ny` were jointly varied from 2^7 to 2^{11} in power of 2 increments (2^7, 2^8, …, 2^{11}). Each of the benchmarked configurations was run a total of five times. The final runtime for a given configuration was arrived at by averaging over its five runs.

5 Benchmarking results

Figures 1 and 2 show the runtimes achieved by the different programming models at various domain sizes on Piz Daint and Strix, respectively.

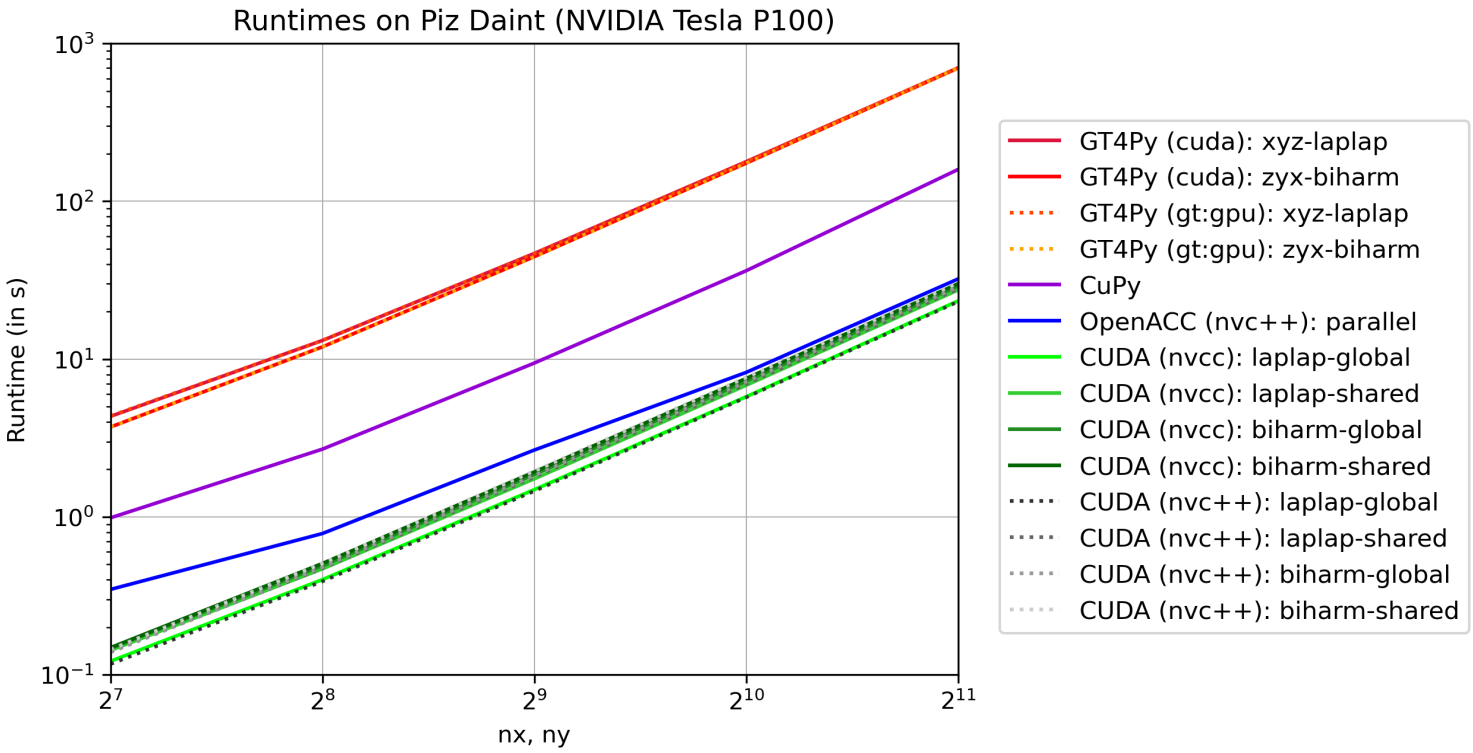

Figure 1: Runtimes at various domain sizes on Piz Daint.

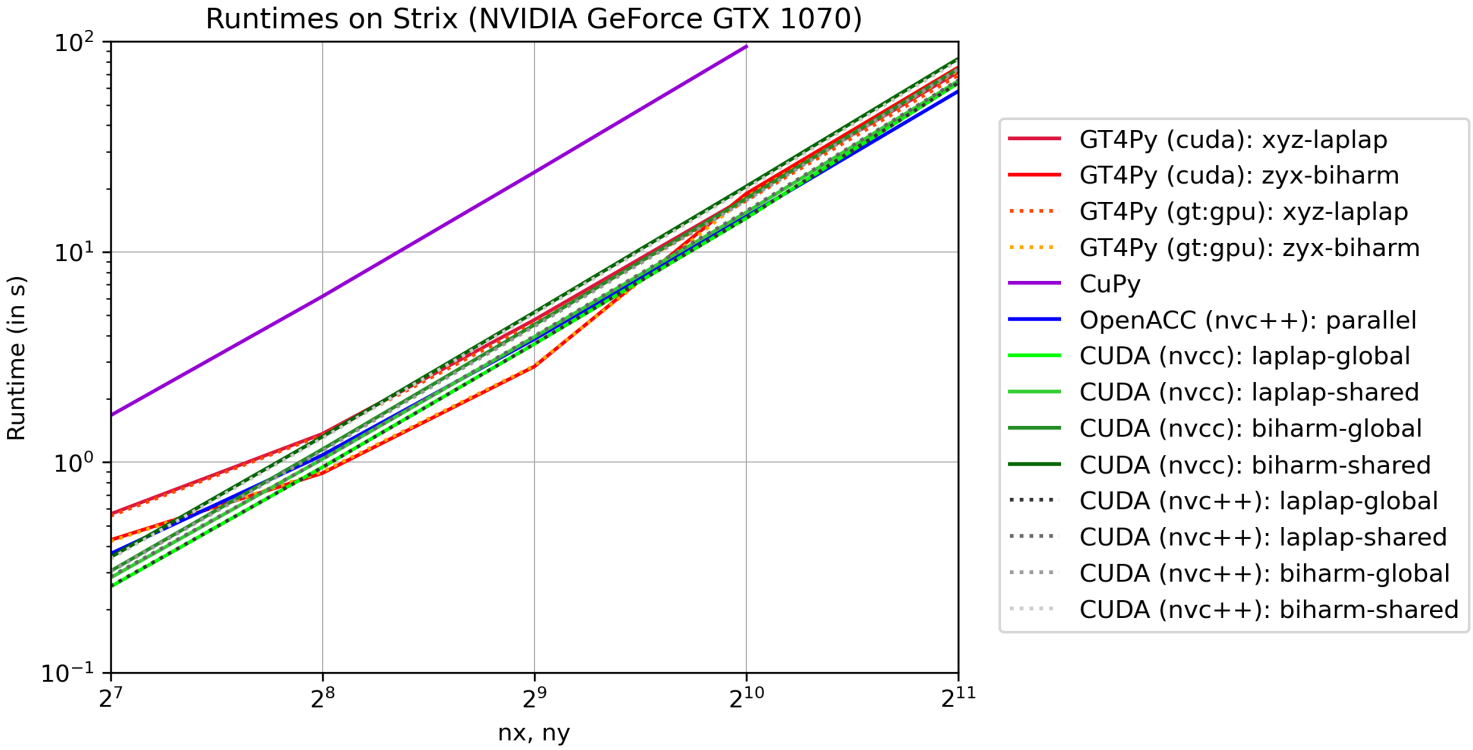

Figure 2: Runtimes at various domain sizes on Strix. No runtime for CuPy at $\texttt{nx} = \texttt{ny} = 2^{11}$ is given due to an on-device out-of-memory (OOM) error at this domain size.

5.1 CUDA

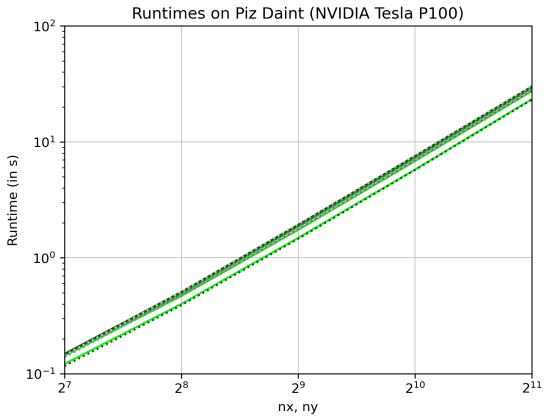

Figure 3: CUDA runtimes at various domain sizes on Piz Daint. Legend provided in figure 1.

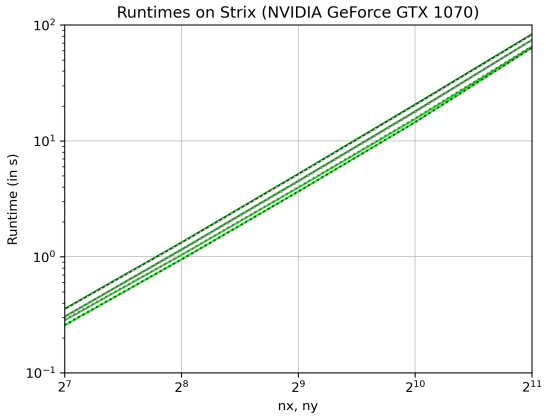

Figure 4: CUDA runtimes at various domain sizes on Strix. Legend provided in figure 2.

A clear hierarchy seems to arise within the different CUDA versions. The `laplap-global` version stands out as the top performer by a significant margin despite merely applying two consecutive 5-point Laplacian stencils without utilising shared memory.

The runtime delta the `laplap-global` version achieves over its next best-performing competitor (the `laplap-shared` version) seems to vary significantly between GPUs. On Piz Daint, this delta remains consistently large, regardless of the domain size. A contributing factor to this behaviour might be that the P100 accelerators used by Piz Daint use HBM2 (High Bandwidth Memory Gen. 2) as their device memory. The high bandwidths HBM2 can achieve, which come about also as a consequence of its very wide 4096-bit bus, may help alleviate the overhead caused by the additional global memory accesses the `laplap-global` version has to perform over its shared memory utilising counterpart.

The GTX 1070 used by Strix, in contrast to the P100, only uses regular GDDR5 as its device memory, which only comes with a bus width of 256 bits, thereby leading to the GTX 1070 presumably being more sensitive to a suboptimal amount of global memory accesses. It is, therefore, unsurprising that on Strix, at least for the larger domain sizes, the runtimes of the `laplap-shared` version get fairly close to those of the `laplap-global` version.

The code paths relying on a single 13-point biharmonic stencil continue the trend of the global memory-only versions outperforming their shared memory-utilising counterparts. They also perform universally worse than both the `laplap-global` and `laplap-shared` versions.

In general, it has to be said that the fact that the global memory-only versions of the code can, in any case, perform so well relative to their shared memory-utilising counterparts is probably also in part due to how well the caching strategy employed by Pascal-generation GPUs works. It can be expected that the global memory-only versions would be far less competitive on earlier generations of Nvidia GPUs (Maxwell, Kepler, Fermi).

Yet, while the above arguments may help explain why the global memory-only and shared memory-utilising code paths of the miniapp might perform similarly to each other, they do not explain why the former outperform the latter. The key to this probably lies in the additional overheads that may be encountered when using shared memory, which are not seen when solely relying on global memory. These overheads include the extra branches necessary to load field entries located on the thread block boundaries and the thread block-synchronising `__syncthreads()` calls required after the threads have finished loading their blocks' respective input field slices into shared memory.

In terms of compilers, there only appear to be minor benefits in using `nvc++` over `nvcc`. The setting in which the most notable differences in terms of performance seem to arise is on Piz Daint for small domain sizes, where the `nvc++`-compiled binaries seem to slightly outperform their `nvcc`-compiled counterparts.

5.2 OpenACC

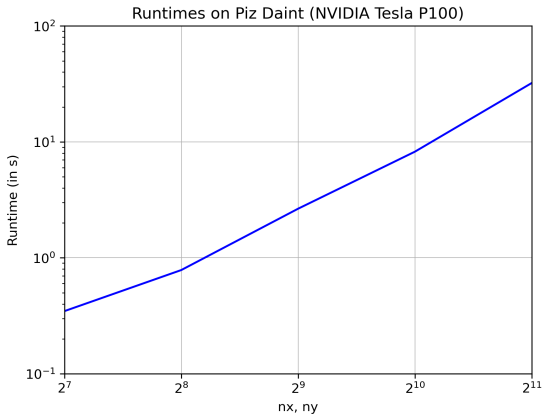

Figure 5: OpenACC runtimes at various domain sizes on Piz Daint. Legend provided in figure 1.

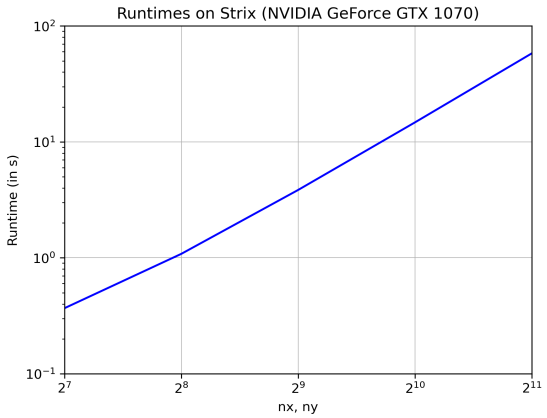

Figure 6: OpenACC runtimes at various domain sizes on Strix. Legend provided in figure 2.

The `kernels` version of the code is not plotted. This is because, already on a 128x128x64 domain, it takes this version of the code over 500 seconds to run on Piz Daint and over 600 seconds to run on Strix.

The `parallel` version of the code, in contrast, performs fairly well, particularly at larger domain sizes. While it never quite reaches the performance of the CUDA codes on Piz Daint, it comes remarkably close as the tested domain sizes increase. This is not hugely surprising, considering that the problem — only involving some stencil computations and copying some boundary data — is extremely simple.

What is, however, surprising are some of the measured runtimes on Strix. Here, the `parallel` version outperforms all of the CUDA code paths by a reasonably sized margin on the largest benchmarked domain size ($\texttt{nx} = \texttt{ny} = 2^{11}$). Notably, this is readily reproducible across multiple benchmarking runs. This feat by OpenACC is as impressive as it is confusing, if for no other reason than the above-expressed notion that the problem being solved is very simple.

The thesis that the better performance of the OpenACC code is due to some inherent advantage of using `nvc++` over `nvcc`, which was considered during the writing of this report, is refuted by the very similar performance that is yielded by benchmarking the CUDA version of the miniapp with both `nvcc` and `nvc++`.

Ultimately, the most likely explanation is that OpenACC simply happens to set some underlying parameter(s), e.g., the thread block sizes it uses, to some values that have it outperform the CUDA versions of the code in this particular scenario ($\texttt{nx} = \texttt{ny} = 2^{11}$ on a GTX 1070) but nevertheless see it be outperformed in the majority of other settings.

5.3 Revisiting the literature

The results presented in sections 5.1 and 5.2 are plainly inconsistent with the findings made by [HMMT13]. In particular, the claim that without careful tuning, OpenACC codes may struggle to keep up with their CUDA-based counterparts does not seem applicable here, as the benchmarked OpenACC (`parallel`) version of the miniapp, for large enough domain sizes, gets fairly close to or, depending on the system, even outperforms the CUDA-backed miniapp. It does this while solely relying on `parallel loop collapse` pragmas, which, notably, are completely device-agnostic.

Furthermore, the importance of using shared memory, which was also stressed by [HMMT13], seemingly also does not apply to the provided benchmarks. While the benefits of using shared memory are almost certainly problem-dependent, the only very limited to non-existent usefulness of using shared memory in the provided benchmarks in light of its claimed importance by [HMMT13] still seems to support the thesis from section 5.1 that Pascal-era GPUs perform better in terms of caching strategy than earlier, in particular, Fermi-era GPUs (like the Tesla M2050 used by [HMMT13]).

5.4 CuPy

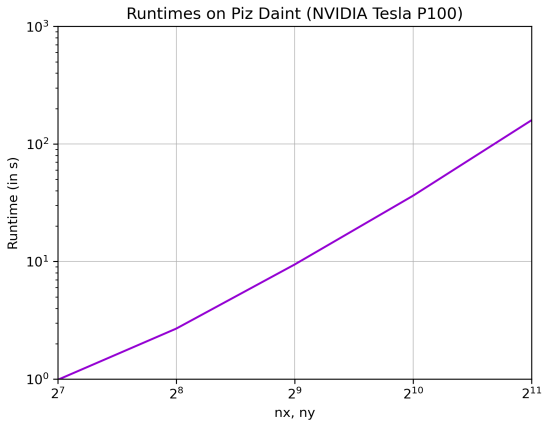

Figure 7: CuPy runtimes at various domain sizes on Piz Daint. Legend provided in figure 1.

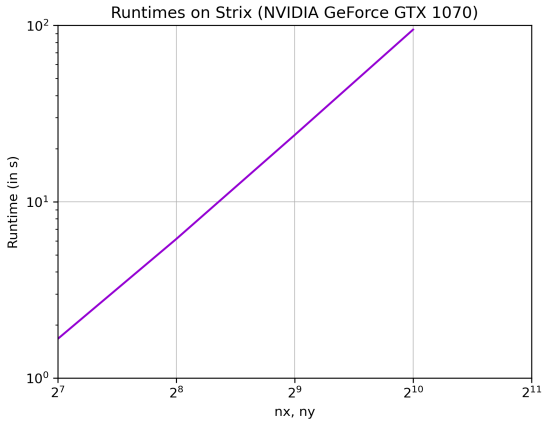

Figure 8: CuPy runtimes at various domain sizes on Strix. Legend provided in figure 2.

It is remarkable that the CuPy code is the only one that runs out of device memory on Strix for `nx` $=$ `ny` $= 2^{11}$. The corresponding error message suggests that this is because CuPy employs additional device memory to save intermediate results (such as `-4 * u[:, jmin:jmax, imin:imax]` during the Laplacian stencil computation). It is probably possible to work around this limitation, i.e., rewrite the code in a way that it runs successfully even for `nx` $=$ `ny` $= 2^{11}$. Yet, the whole point of using CuPy is that it is high-level and easy to use. Having to look for workarounds seems very much contrary to this design goal — especially considering the apparently rather lacklustre performance one receives in return.

5.5 GT4Py

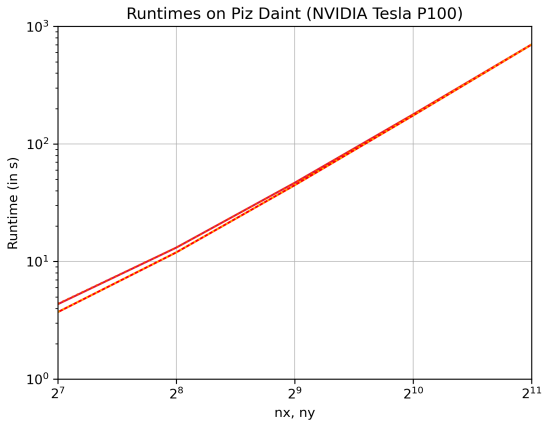

Figure 9: GT4Py runtimes at various domain sizes on Piz Daint. Legend provided in figure 1.

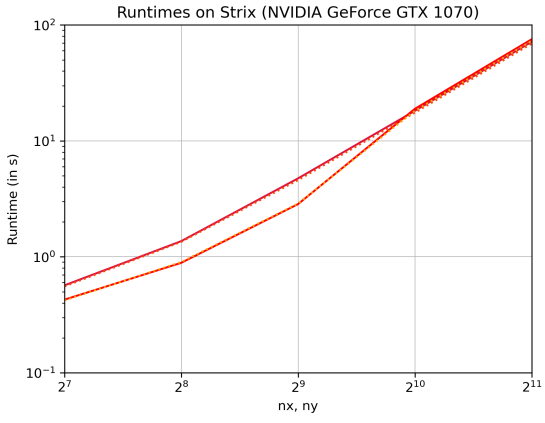

Figure 10: GT4Py runtimes at various domain sizes on Strix. Legend provided in figure 2.

The most notable feature of the Piz Daint results is the very poor performance of the GT4Py codes. This poor performance is what motivated running the benchmarks on Strix in the first place. On Strix, GT4Py performs approximately as well as CUDA and OpenACC. This suggests that the poor performance of GT4Py on Piz Daint is either due to being hopelessly outdated or simply misconfigured.

In any case, it does not seem to matter whether one uses the `gt:gpu` or `cuda` backends, as they seem to result in exactly the same runtimes independently of whether GT4Py is run on Piz Daint or Strix.

Furthermore, it is remarkable that on both Piz Daint and Strix `zyx-biharm.py` outperforms `xyz-laplap.py` for all tested domain sizes, where `nx`, `ny` $< 2^{10}$. It is not immediately apparent why this is the case ($\rightarrow$ possible caching effects).

5.6 General comments

Overall, it seems that the CUDA, OpenACC, and (on Strix) GT4Py codes perform fairly similarly. The CuPy code, however, generally seems to run significantly slower. Notably, the CUDA, OpenACC, and GT4Py codes are all compiled (or in the case of GT4Py, at least its performance-critical portions are), while CuPy (without using, e.g., `cupyx.jit`) is effectively interpreted (technically, it uses what its documentation refers to as "on-the-fly kernel synthesis" [Pre24]; as this "synthesis" seems to fail however in optimising away redundant intermediate memory allocations, I would still classify this approach as overall interpreted). This suggests a certain advantageousness of using a compiled approach, at least for the performance-critical portions of the miniapp.

6 Further discussion

The previous sections have detailed the implementation process and benchmarking results for the various here-considered programming models. Beyond just raw performance, however, additional factors such as portability and the time required for development are also important when deciding on which programming model to use. This section seeks to contextualise the benchmarking results from the previous section by considering these additional factors.

CUDA is a very consistent performer: It achieves universally low runtimes while also exhibiting remarkably linear strong scaling plots.

The more pressing question with CUDA, however, is whether it really is worth dedicating hours upon hours to, for the most part, rewriting existing loop-based computational code to CUDA-friendly computational kernels, adding the appropriate kernel launch infrastructure, initialising a CUDA context, etc., when the resulting binaries seem to be (almost) matched and in some cases even exceeded by far less elaborate programming models (OpenACC & GT4Py). While it is to be presumed that when tackling more challenging problems, the additional control one receives through CUDA would be more useful, it does not necessarily seem worth the additional hassle in the here-considered setting.

OpenACC is quite a strong contender: While it is soundly outperformed by CUDA on smaller-sized domains, it becomes increasingly more competitive on larger-sized domains (where performance arguably matters more anyway). It does this while only taking a fraction of the development time required by the CUDA version of the code.

The main concern with OpenACC is not necessarily its performance but its compiler support. For instance, OpenACC support was dropped from Cray's C and C++ compilers when they were migrated to using LLVM as their backend. OpenACC is also not natively supported by LLVM/Clang, and while there is an ongoing effort to implement OpenACC support on top of LLVM's existing OpenMP offloading support, this effort still seems ongoing at this time. This makes it seem quite a lot more practical and overall desirable to simply use OpenMP offloading in place of OpenACC, at least when accelerating a C or C++ code, as OpenMP offloading is natively supported by a wide variety of compilers such as: `gcc/g++`, `clang/clang++`, `nvc/nvc++`, `craycc/crayCC`, etc.

CuPy appears a bit lost in this comparison. Its performance and efficiency suffer greatly, presumably due to its interpreted design. It would be interesting to see how CuPy would perform in conjunction with a just-in-time compiler (such as the one provided by Numba through its llvmlite backend). The resulting execution times and memory utilisations would presumably be significantly lower than what was observed in the here-considered benchmarks.

It might also be worth exploring the performance of, e.g., PyTorch tensors, as with the recent surge in interest with regards to everything machine learning, their efficient implementation has presumably received quite a bit of attention.

GT4Py is a bit of a special case, as it is the only programming model investigated that specifically targets stencil applications. Its performance is quite good (at least when using recent versions of it).

Yet, GT4Py, too, comes with a set of quirks: Its toolchain seems to be somewhat fragile; it breaks when used with the latest Python version and generates numerous compiler warnings while compiling the `laplap` and `biharm` stencils. Additionally, its error messages are not always particularly helpful, with another point of concern being that among all of the programming models surveyed in this report, GT4Py has by far the smallest user community.

7 Conclusion

This report has provided a comprehensive analysis of the performance characteristics of different GPU programming models covering high-level (CuPy & GT4Py), mid-level (OpenACC), and low-level (CUDA) approaches. By benchmarking different incarnations of the stencil2d miniapp across various domain sizes on two different GPUs — those being Nvidia's Tesla P100 and Nvidia's GeForce GTX 1070 — significant insights into the strengths and weaknesses of each of these programming models have been gathered.

Ultimately, the correct choice of programming model really depends on what one wishes to achieve. If one wants very good, consistent performance with essentially guaranteed long-term support from Nvidia and one does not mind having to put in quite a lot of effort for potentially little additional benefit in terms of performance, one should choose CUDA. If one is prepared to put faith in the compiler's auto-vectoriser doing a decent job (as it, on the whole, did here), one should choose OpenACC (or, in order to address potential longevity concerns, OpenMP offloading). If one is mostly interested in accelerating stencil computations and one can live with its idiosyncracies, GT4Py might be a solid choice. CuPy (at least when used in the fashion benchmarked here) does not seem recommendable unless one has an existing NumPy-based code and one wants to get a taste of the benefits GPU acceleration may deliver.

References

[HMMT13] Tetsuya Hoshino, Naoya Maruyama, Satoshi Matsuoka, and Ryoji Takaki. CUDA vs OpenACC: Performance Case Studies with Kernel Benchmarks and a Memory-Bound CFD Application. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 136–143, 2013.

[Nvi24] Nvidia Corporation. CUDA C++ Programming Guide: Release 12.5, Jun 2024.

[OUN+17] Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido, and Crissman Loomis. CuPy: A NumPy-Compatible Library for NVIDIA GPU Calculations. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017.

[PGU+23] Enrique G. Paredes, Linus Groner, Stefano Ubbiali, Hannes Vogt, Alberto Madonna, Kean Mariotti, Felipe Cruz, Lucas Benedicic, Mauro Bianco, Joost VandeVondele, and Thomas C. Schulthess. GT4Py: High Performance Stencils for Weather and Climate Applications using Python, 2023.

[Pre24] Preferred Networks, inc. and Preferred Infrastructure, inc. CuPy Documentation: Release 13.2.0, Jun 2024.

[The22] The OpenACC Organization. The OpenACC Application Programming Interface: Version 3.3, Nov 2022.

[Xue00] Ming Xue. High-Order Monotonic Numerical Diffusion and Smoothing. *Monthly Weather Review*, 128(8):2853–2864, 2000.