3

Structured Grids and Stencils

Structured grids are one of the original "Seven Dwarfs" of parallel computing [20]. In structured grids, the problem domain is represented by a grid of cells which are updated together in timesteps. These problems have high spatial locality, meaning that the cells depend only on other cells in some close neighbourhood. Grid resolutions can be different for different parts of the spacial domain (adaptive mesh refinement).

Iterative solvers for partial differential equations (PDEs), such as the finite difference methods common in scientific simulations, are typically implemented as structured grid applications. Filters for image processing are another common example.

3.1 Stencils

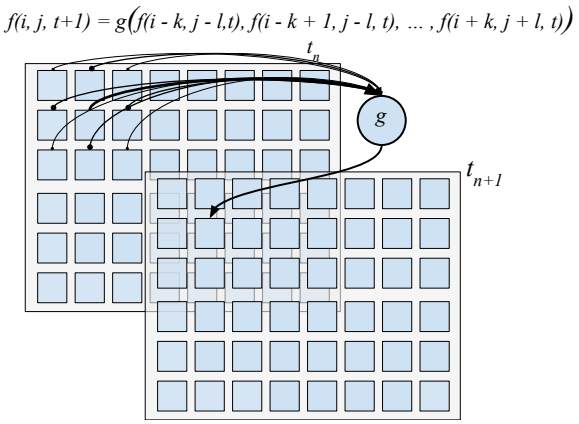

Figure 3.1: Illustration of a stencil function. $g(i, j)$ uses the neighbourhood of cells of $f(i, j, t)$ to compute the next cell value $f(i, j, t + 1)$

Updates to cells in the structured grid are normally described using the *stencil* pattern. McCool, Reinders and Robertson describe the pattern as a special case of the *map* pattern [21] in which inputs to a transformation include not only a cell, but also its immediate neighbours, but the transformation has only one output cell (see Figure 3.1). For iterative PDE solvers, the stencil operation can be repeated many (usually 10000s of) times, forming a *recurrence*.

The importance of stencils in HPC software has resulted in considerable research for their efficient implementation over a range of architectures.

Stencils tend to have a low ratio of work to the amount of data loaded for each stencil - that is, they have a low *operational intensity*. This means that on most architectures, getting the data to the processor is the bottleneck for performance, and stencils are well-suited to architectures with very high memory bandwidth, such as GPUs.

Exploiting parallelism for stencils requires dividing the input grid up into partitions to be computed by different processors. However, for any given partition, the neighbouring partitions' borders make up a "halo" of data-dependency cells. This halo is communicated either by passing messages or through shared memory (in a process called "halo exchange").

In addition to being necessary for splitting up work over cores, partitioning the grid space affects memory access patterns and data reuse in caches. Tiling patterns that account for both space and time re-use of data (i.e. within as well as between loop iterations) aim to increase the operational intensity of stencils by ensuring that caches are efficiently used and global memory accesses are minimised.

Below we give a brief overview of related work into optimising stencils on CPUs and GPUs. Optimisation of these communication patterns, partitioning approaches, and other appropriate optimisations are discussed for the IPU in Chapter 5.

3.2 Related work

Optimisation of communication patterns of stencils on multicore architectures is studied by Palmer and Nieplocha in [22]. They list strategies for reducing communication by sending more border data than needed, and redundantly recomputing "ghost zones" on different tiles. They conclude that appropriate communications patterns are specific to each architecture. Ding and He created a "two-wave" algorithm for reducing the number of messages needed for communicating partition borders in [6], which we will apply for the IPU.

Datta *et al* [8] consider a number of stencil optimisations for a range of multi-core architectures (both CPUs and GPUs), and develop and auto-tuning framework for these. Most of their optimisations are focused on improving data locality and hiding latency: breaking the domain into a hierarchy of (spatial) blocks, NUMA-aware memory allocation, array padding for filling cache lines, multilevel blocking (for cores, threads and registers), loop unrolling and reordering, as well as pre-fetching and double buffering. Of these, only loop optimisations and double buffering may be of use for the cacheless IPU. Notably, the architecture most similar to the IPU in their study, which had fast 256KiB local stores and no caches, showed no benefit from tiling optimisations.

Much of the recent work on stencil optimisation is focused on improving cache locality (and local memory locality for GPUs) through a variety of loop tiling mechanisms. Cache oblivious [23] as well as cache-aware tiling strategies (e.g. [7], [10], [9], [24], [25]) are reported to show great improvement for CPU and GPU implementations with shared caches, but are inapplicable to IPU implementations when the whole grid is in IPU memory.

In a recent study for the NVIDIA V100 GPU, Yang applied eight optimisations to a stencil

[26], guided by a Roofline performance model. We will follow a similar approach after presenting a Roofline model for the IPU in Chapter 4. However, we note that Yang's optimisations also focus on improving GPU local memory and cache locality, and so have limited application for the IPU.

The range of complex options for stencil optimisation, and their different applicability to different architectures, has resulted in the development of stencil auto-tuning frameworks. These take a high-level description of a stencil in a domain-specific language (DSL), and then test the effectiveness of a range of optimisations on a specific platform to produce customised code that runs fast. Examples are SkelCL [27], the Pochoir stencil compiler [28], PATUS [29] and pystencils [30]. Halide [31] allows stencil operations to be represented as a high-level pipeline, allowing for rapid manual exploration of the impact of loop transformations for various platforms. None of these frameworks currently target the IPU, but we suggest integrating with them as future work.

Rajopadhye *et al* argue in [32] that the inability of GPU synchronise between streaming multiprocessors results in unnecessary accesses to global memory during stencil computations, and describe what an idealised "stencil processing unit" might look like. This idealised architecture shares some of the characteristics of the IPU in allowing exchanges between on-chip memories for better energy efficiency, encouraging our study.

3.3 Why might the IPU be a viable platform for structured grids?

Computing each point on a stencil requires repeated accesses to surrounding data values in both space and time, making most stencils memory bandwidth-bound rather than compute bound. Accessing main main (off-chip) memory on a CPU can take 100s of instruction cycles. Values that are in faster caches can be retrieved in a few cycles, these operations require much less energy. As a result, most of the optimisations discussed above focus on making sure that the right values are in the fastest (lowest-latency, highest bandwidth) memory.

We have seen in Chapter 2 that the IPU devotes more of its transistors to fast on-chip SRAM than any other processor to date. In effect, this SRAM forms an extraordinarily large cache. These memories are distributed to be physically close to each core to reduce latency to 6 instruction cycles. Local memory is ultra high bandwidth (listed as theoretical peak of 45 TB/s in Graphcore marketing material).

Furthermore, points needs access to their neighbourhood in the grid, needing some form of communication when the grid is distributed between memories. The IPU's fast all-to-all exchange fills this role. Tthe communication patterns which arise from predictable, regular data dependencies in a small neighbourhood, mean that dependencies are static and can be scheduled via a dataflow graph, making them extremely suitable for Poplar's compute graph paradigm.

The work of Rajopadhye *et al* [32] suggests that an ideal energy-efficient stencil architecture will have the ability to cheaply synchronise shared data between processors without relying on round-trips to global memory. The above points show that the IPU satisfies this through its fast all-to-all exchange and distributed on-chip SRAM memory.

3.4 Summary

In this chapter we have described structured grid applications and the stencil pattern common in their implementation. We argued that the IPU's high memory bandwidth, massive parallelism and relatively large on-chip SRAM are well suited to stencil computation, and that stencil data dependencies result in communication patterns which can easily be expressed in Poplar's compute graph paradigm. Next, we consider how to implement the building blocks for stencils in Chapter 5.

5

Building blocks for implementing stencils on the IPU

This work presents the first known implementation of stencils on the IPU. During the course of this project, patterns for implementing stencils in Poplar had to be discovered, and implementation alternatives critically evaluated without recourse to existing examples. This was a time-consuming and iterative process, which often led to late discoveries that required re-evaluating earlier work. In this chapter, we present our original, re-usable, problem-independent building blocks for implementing stencil programs on the IPU for the first time, saving other from having to repeat this process. We use these patterns in Chapters 6 (Gaussian Blur) and 7 (Lattice Boltzmann).

The building blocks discussed are:

- Halo Exchange, discussed in Section 5.1, where we find an efficient implementation for communicating the state of shared cells between the IPU's processors.
- Grid partitioning, discussed in Section 5.2, where we develop an algorithm to split the input grid between the tile processors and their distributed memories in a way which supports halo exchange
- A discussion of out-out-place vs in-place update schemes, presented in Section 5.3, for using the IPU's relatively small memory effectively.
- Optimisations which apply to stencils on the IPU, which are discussed in Section 5.4.

Finally, we present a generic end-to-end stencil Poplar program skeleton that uses these concepts in Section 5.5.

5.1 Halo exchange

The IPU only has local memory for each tile, and has no global shared memory, so we are forced to partition the input grid between tiles' memories. But stencils need access to every cell's neighbourhood, which is a problem for cells at the boundaries of a tile's partition. For these neighbours, we will need to communicate with remote tiles. This communication of boundary regions is called halo exchange, and is common in distributed stencil implementa-

tions [6]. As shown in Figure 5.1, a tile's received halo cells are stored in extra cells, called ghost cells.

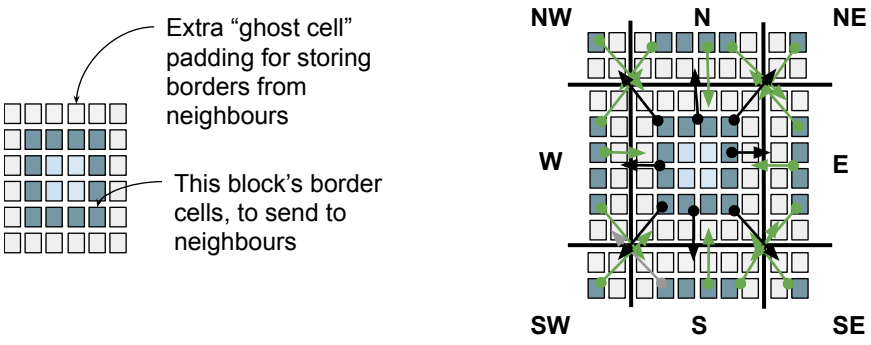

Figure 5.1: Al illustration of halos, ghost cells and halo exchange.

The behaviour of cells at the very edges of the grid (i.e where there are no obvious neighbours) is algorithm-dependent, but common choices are zero-padding, reflecting, or wrapping around to values at the far side of the grid.

The choices made during partitioning affect the amount of the data that needs to be communicated during halo exchange. If we partition the grid into strips (so-called "strip-mining" [21]), then only the top-bottom, or left-right neighbours need to exchange information, but the borders are large ($2n_{cols}$) relative to the size of the non-border region ($n_{cols} \times n_{rows}$ cells, with $n_{cols} \gg n_{rows}$). If we partition into blocks, then each processor needs to communicate with up to 8 other processors, but the border regions are smaller ($2n_{cols} + 2n_{rows} + 4$ cells, with $n_{cols} \approx n_{rows}$). Our choices for grid partitioning will be discussed extensively in Section 5.2, but for now we note that we will rely on blocks as our primitive, and have 8 halo regions to exchange.

When communication latency is expensive, neighbour communication can be optimised into two waves [6]. Another common optimisation is to make halos deeper (so-called "fat halos", described in [6]), so that ghost cells for several layers are exchanged and computation for several timesteps can proceed without requiring further halo exchange. Computations in the fat halo region are redundantly repeated on neighbouring tiles, so this only makes sense when communication latency is much more expensive than re-computation.

In asynchronous architectures, the communication latency can also be hidden to a certain extent by overlapping computation and communication of halos, especially with fat halos. Interior computations can proceed while waiting for halo exchange at the boundaries.

In the next section we describe the various approaches to implementing halo exchange on the IPU.

Halo exchange on the IPU

Our experiments in this section were inspired by work by Palmer and Nieplocha [22], who performed a similar analysis for distributed memory systems based on Global Arrays [17].

The IPU architecture constraints our implementation choices. Firstly, the BSP architecture means that communication is inherently synchronous, and exchanges between tiles are only

possible during the exchange part of the superstep[1]. This fundamentally affects the program's design, while also eliminating the possibility of explicitly overlapping communication and computation, and any advantages for fat halos.

Secondly, exchanges are fast (8TB/s bandwidth on the same IPU – in the same order of magnitude as local memory bandwidth) and low-latency ($\sim$ 133ns), whereas communication between remote nodes in distributed systems normally has orders of magnitude difference in latency from memory access. This reduces the pressure to communicate less frequently on the IPU.

Thirdly, a primary limiting factor in program design for the IPU is the relatively small local tile memory size (256KiB). Messages to be passed between tiles consume tile memory, encouraging a strategy of smaller, more frequent exchanges, and preferring strategies which uses less memory for ghost cells (e.g. blocks rather than strips).

For these reasons, we choose single-layer halos, and block-style communication when communicating halos between tiles.

Implicit vs Explicit halo exchange On the IPU, data exchange between tiles is implemented with special instructions at the hardware level [15]. In the Poplar programming model, these are only indirectly available to the programmer except through two mechanisms, both at the compute graph conceptual level:

1. Communication is *implicitly* scheduled when a vertex's `Inputs` or `Outputs` are wired up to tensor slices stored on remote tiles. Note that, although the ghost cells on one tile and the originating boundary region on its neighbour tile are physically stored in different memories, there is no logical tensor distinction between them. They refer to the same slice of the same tensor, and the compiler will enforce these semantics by not allowing the values to diverge. The compiler generates the exchange messages to be passed, and performs considerable optimisation in message packing and scheduling. This naturally places communication exchange part of the BSP superstep.

2. Communication can also be *explicitly* triggered by using the `Copy` program primitive to specify that a slice of a tensor should be copied to different location in the same tensor, or to a different tensor. Note that the source and destination are no longer the same tensor slice, and can change independently.

Both implicit and explicit communication naturally imply sharing data between `Tensors` in which all elements are of the same data type. Sending a heterogeneous structure to remote tiles is awkward, and requires using several tensors or manually recasting bytes in the vertex.

We ran experiments with variations of explicit halo exchange on the emulator device, including the two-wave optimisation from [6] and various schemes for partitioning and padding. The emulator allows us to accurately measure `Copys` while setting a dummy cycle count for other operations. While these scenarios worked on the emulator, unfortunately on the real

[1] although in reality, computation proceeds as soon as the local exchange is complete, and the IPU optimises overlap of phases to some extent

IPU device, `Copy` programs were scheduled by the compiler in an invalid order, resulting in a `TEXCH_RERR_CLASH_INSTR` device exception. This issue is still open with Graphcore support. Because the explicit schemes never worked on the real device, we do not show timings in this report. In any case, our experiments showed a clear speed advantage for implicit halo exchange, which we could run successfully on the IPU.

The implicit style has a nasty consequence for vertex design: because of the way halos are wired up, halo regions must be defined as separate `Inputs` in the vertex, resulting in extra `Input` variables for "northeast", "north", "south" etc. as well as the middle "data". When looping over the partition, accessing the neighbourhood of a cell uses different data structures depending on its location, resulting in many edge cases (see Figure 5.2). In the worst 2D case (a 9-point stencil), the vertex has to deal with 9 different data structures, and the compute graph has to wire all of these correctly. In a simple demonstration program, we found that our implicit implementation needed 2-3x as much code for the vertex implementation than the explicit one, and had more complex wiring. The situation is even worse for higher-dimensional grids.

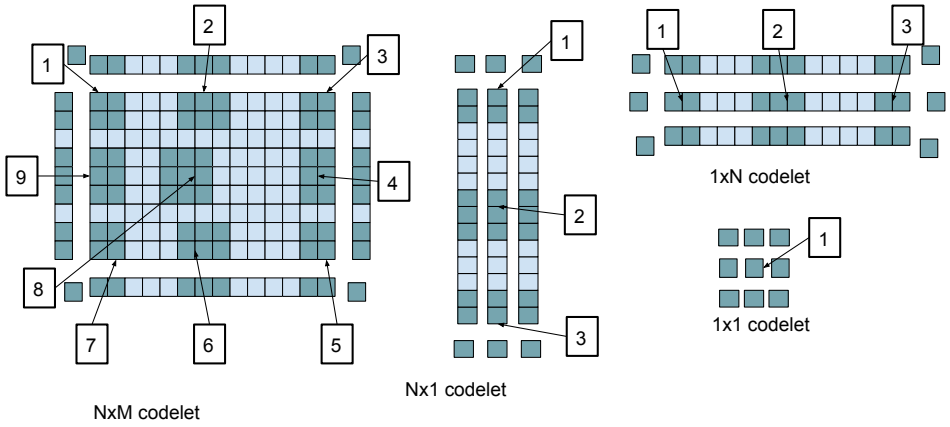

Figure 5.2: Loop regions when halo stitching is not used (trading code simplicity for performance). Different data structures overlap in the stencil, meaning having to implement 4 vertexes to deal with different partition shapes

This downside can be entirely avoided by using compute-graph level operations on the input tensors to concatenate them during wiring, resulting in only one `Input` (with halos) and one `Output` (without halos) in the vertex. This operation is implemented as `stitchHalos` in our library, and is shown in Listing 5.1.

```
auto stitchHalos(const Tensor &nw, const Tensor &n, const Tensor &ne,
                 const Tensor &w, const Tensor &m, const Tensor &e,
                 const Tensor &sw, const Tensor &s, const Tensor &se) -> Tensor {
    return concat({
                      concat({nw, w, sw}),
                      concat({n, m, s}),
                      concat({ne, e, se})
                  }, 1);
}
```

Listing 5.1: Concatenating the 9 data regions into one convenient input tensor with ghost cells

Halo stitching and data rearrangements

The convenience of halo stitching comes at a cost to performance. The graph compiler generates "pre-arrange" and ''post-arrange" operations in which the blocks of data are copied into the right form before the BSP compute phase begins. In some of our experiments with simple kernels (e.g. vectorised average), rearrangement cost took up to 25% of the execution time.

Rearrangements are also triggered when vertex inputs are specified to be aligned differently to how they are stored in tile memory, as is usual when we want to process data using vectorised data types like `float2`, or when a different memory bank is requested in a vertex. When rearrangements are occurring in any case, it may be worth opting for the convenience of stitched halos.

Whether or not to stitch halos should be decided on a case-by-case basis. We recommend starting with the simpler code (stitched halos), and opting for separate halo variables when needed. These optimisations are a good target for future code generators.

5.2 Grid partitioning

In parallel computing, domain decomposition, geometric composition, tiling, grid partitioning and grid segmentation are terms used to describe the process of splitting the input grid's cells into regions and allocating them to the parallel processors and distributed memories. To avoid confusion with the IPU meaning of "tile" (a core with its associated local memory), we refer to "partitioning" or "decomposition" in this work, and refer to "blocks" rather than "tiles" of the grid.

Each decomposition scheme will have different consequences for program design and execution efficiency, especially in the matters of data locality and load balance:

Data Locality A decomposition which achieves good data locality will avoid unnecessary communication between processors by storing the grid cells in the memories closest to the processors where they will be processed. On architectures with memory hierarchies and caches, this means optimising for memory bandwidth and latency by allowing cells which are processed together to be loaded efficiently into the fastest layers of memory (usually caches) and keeping them there as long as they are needed. The IPU is unusual in that it has a flat memory hierarchy with low memory latency and high bandwidth, and there are no caches or non-uniform memory access (NUMA) concerns. Furthermore, exchanges between tiles are relatively cheap. This means that assumptions about common implementation patterns developed for CPU and GPU memory hierarchies need to be re-visited, since most are designed to achieve good cache locality.

Load Balance Achieving good load balance means dividing the input grid so that processing time is roughly equal between the different processors. In the most general case, work can be in the form of an unstructured grid, or sparse graph, and must be allocated to processors in a heterogeneous system made up of devices and communication channels with differing performance characteristics. This problem is challenging (in fact, it can be expressed as the well-studied graph partitioning problem, which is NP-hard), and

the focus of scientific software libraries such as ParMETIS [45]. However, in designing a structured grid problem for the IPU, our constraints are significantly simpler: each processor is identical, the communication costs are easy to model, and the cost of work is the same for pieces of the grid that are the same size.

Good domain decomposition is the primary determiner of a program's execution time on the IPU. Even the best-crafted vertex implementations cannot compensate for bad choices at this stage, so it is worth ensuring that our grid partition algorithm performs well.

Several approaches to domain decomposition exist. In its simplest form – geometric decomposition – consideration is given to the spatial geometry of the grid, and "spatial blocking" divides the grid into parts such as strips or blocks in 2D, or planes or cubes in 3D. However, another less obvious dimension is available for consideration: time.

Since our stencil problems' outermost loop represents an iteration through time, time decomposition ("temporal blocking") can be applied to group work into regions that share common data dependencies, by considering which steps will depend on the output of which preceding ones. Polyhedral loop optimisation theory considers how loops can be skewed and spatio-temporally tiled to best reduce dependencies and maximise parallelisation. These lead to approaches such as pipelined wavefront parallelisation and various polytope decompositions by polyhedral compilers presented in [46], and are common in the state-of-the art stencil tools for CPUs and GPUs [47, 10]. However, these approaches come at the cost of greater code complexity, and since they specifically aim to increase the operational intensity of kernels by encouraging better cache (or local memory) locality, it is not obvious that they will offer any advantage for the IPU.

Considerations for the IPU

The IPU's architecture and the Poplar framework not only force us to consider our approach to geometric decomposition explicitly and early on in the program's design, but the architecture also makes some approaches better than others. Specifically:

1. How can we fit a given input grid into the 1216 small distributed memories? Each tile can only host as much of the grid as can fit in its memory (at most 256KiB), while also leaving enough space for communication buffers, stack space, executables (vertex code) etc. This places an upper limit on each partition's size.
2. Since we want to enable maximum parallelism, we want grid partitions to be granular enough so that we can use as many tiles as possible (i.e. we should aim for roughly 1216 even chunks of work per IPU). This encourages small partitions.
3. Each tile has 6 hardware worker threads which run in parallel, and to get the best utilisation of the hardware and hide instruction and memory latency, Graphcore encourages scheduling at least 6 vertexes per tile. Since we want the 6 workers to finish at the same time, we must further subdivide a tile's allocated piece of the grid into 6 roughly equal-sized parts. Needing to keep threads busy places a lower limit on a tile's partition size.
4. Our partitions' border regions should be rectangular, and cannot be jagged (e.g. covering the last third row n and the first third of $n+1$). Although slicing a tensor will always result

in a non-jagged shape, it is possible to flatten tensors and store jagged regions on tiles. But if we do this, halo exchange becomes much more difficult: we will have to keep track of the jagged boundary and schedule more complex halo communication. This constraint eliminates the use of `poputils` tensor partitioning utilities, which flattens tensors before partitioning.

5. Each tile can only reference its own local memory, so the obvious allocation is to have the vertexes on the tile process data which is stored in the local tile memory rather than on a remote tile (which would implicitly insert exchanges to bring a copy of the data to the tile). Hence domain decomposition overlaps with the concern of optimal `Vertex` placement and wiring.
6. The BSP architecture means that load imbalance has a huge effect: since each superstep will be determined by the busiest tile (slowest worker), all computation is blocked until the busiest worker finishes. Conversely, increasing parallelism by assigning smaller grid partitions to more tiles will have no effect unless doing so reduces the biggest grid partition assigned to any tile. This encourages an equal partition size.
7. Reading and writing local tile memory is cheap (one "thread cycle", i.e. latency <4ns), sending/receiving data on other tiles on the same IPU is slower (latency 133ns), and from other IPUs slightly slower again (<1ms), so on-chip communication is to be preferred when local memory access is not possible. Since tiles have no cache, schemes designed for good cache locality do not apply, since they cannot be implemented.
8. Memory latency does not depend on access pattern (stride, row-major vs column major etc.) Other than thinking about storing data to be able to exploit vector instructions (only 2 floats wide), we have considerably more freedom to choose data access patterns than on architectures where cache lines and wide vector widths encourage an obvious choice.
9. As will be discussed in Section 5.3, to avoid graph cycles, Poplar enforces a restriction that two vertexes cannot both be using a tensor slice as read-write in the same compute set. Grouping together things which are being read vs things that are being modified results in simpler code.

Grid Partitioning Strategy

We used a benchmark-driven approach to design a grid partitioning algorithm that attempts to satisfy the constraints and meet the optimisation objectives outlined in the previous section. Based on the discussion in the previous section, we do not apply temporal blocking in the decomposition (since the high number of processors means pipelined wavefront startup will be excessive, and the goal of better cache locality is not applicable for the IPU).

Our approach is hierarchical and applies spatial decomposition at three levels: the IPU, the tile and the worker levels.

IPU-level split

Inter-IPU exchange can be two orders of magnitude slower than intra-IPU exchange ([5]). In considering whether our partition model needs to account for this, we must determine whether the increase in parallelism from using more IPUs offsets the cost of communication, once we have managed to fit a grid in IPU memory. To test this, we construct a problem that fits on 1

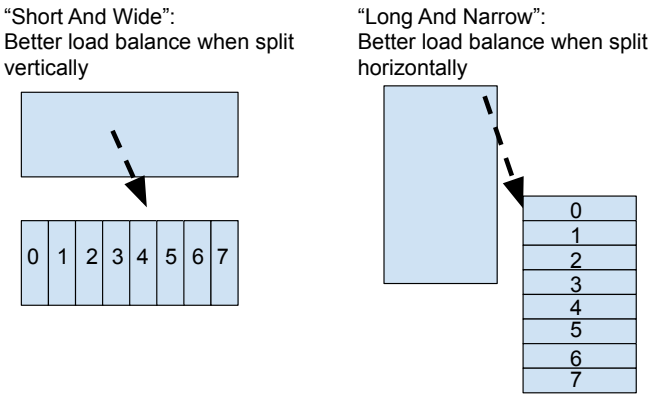

Figure 5.3: `LongAndNarrow` and `ShortAndWide` partitioning strategies. Applied at IPU and worker levels.

IPU, with 128x128 cells assigned to each tile, and run a simple stencil computation (averaging the Moore neighbourhood). We can then test the strong scaling by repeating the test on 2, 4, 8 and 16 IPUs, spreading the load evenly over IPUs every time (128x64 tile partitions for 2 IPUs, 64x64 for 4 IPUs, 64x32 on 8 IPUs, 32x32 on 16 IPUs). By measuring the strong scaling (Figure 5.4), we see that it is always worth using more IPUs.

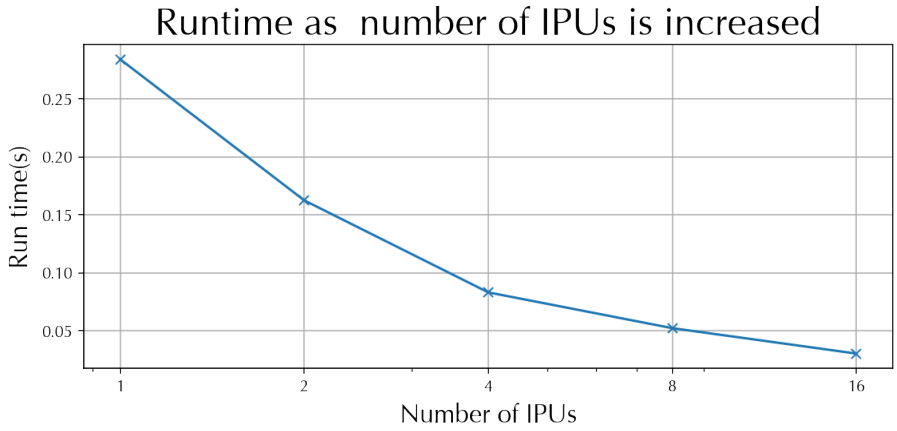

Figure 5.4: Strong scaling profile for a simple stencil demonstrating the preference for multi-IPUs. While the parallel efficiency decreases with multi-IPUs because of communication cost, the overall execution time is still lower, so in general we try to use the maximum number of IPUs.

In our implementation, for each input grid three IPU-level split outcomes are possible (see Figure 5.3:

1. we use the `LongAndNarrowIpuStrategy` when splitting up the grid horizontally would give a better balance between IPUs, or
2. we use the `ShortAndWideIpuStrategy` to split up the grid vertically, or
3. we cannot fit the data onto the available number of IPUs

We slice along this deciding dimension so that every IPU initially gets a slice of size $\lfloor \frac{n}{numIpus} \rfloor$. The remaining volume $mod(n, numIpu)$ is allocated round-robin over the IPUs. The maximum load imbalance is 1 row or column between the busiest and least busy IPU.

We saw no particular advantage in splitting work between IPUs as blocks rather than strips. Doing so would mean that some blocks have to exchange information between more than 2 IPUs, and since the number of possible configurations is limited to 1,2,4,8 or 16, the choice of efficient block arrangements is limited at this level.

Tile Level split

For each IPU, we further split its partition into one suitable for its tiles. We use blocks at this level because they allow us to better fill the tile memories [2] have fewer number of bytes to send as halos.

Firstly, we define parameters for the most granular block size allowed: $minColsPerTile \times maxColsPerTile$. In Section 5.2 we discuss good defaults for these.

Based on the partition assigned to the IPU, two outcomes are possible:

1. Use the `SingleTile` strategy when there is less data than the minimum block size (unlikely for real problem sizes).
2. Use the `GeneralBlockStrategy` for the general case.

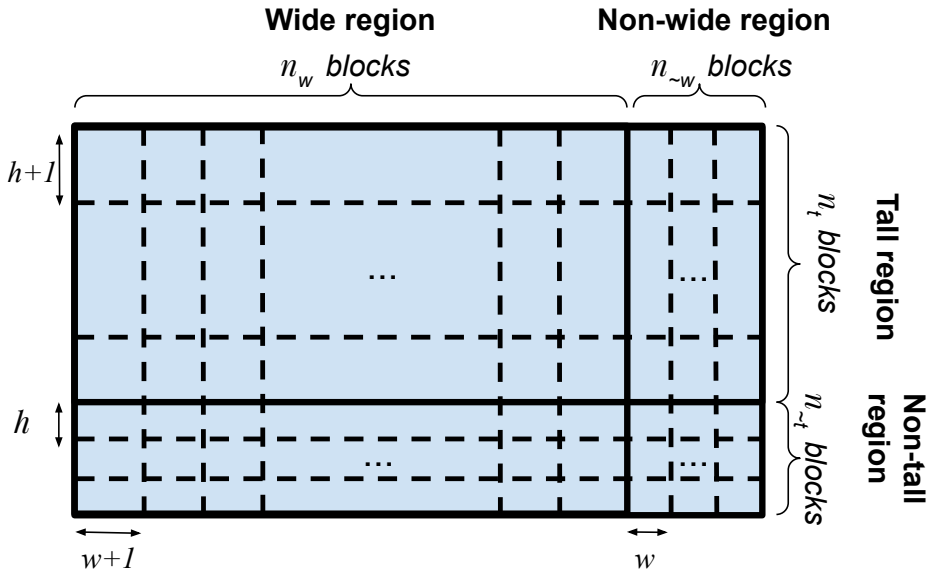

Figure 5.5: General block partitioning strategy. The IPU's 1216 tiles are arranged into a virtual grid with four possible regions of imbalance in the number of rows and columns. Our strategy tries to find an optimum tile size by minimising $(h + w)$ with the constraints: $h \geq minRowsPerTile$; $w \geq minColsPerTile$; $(n_t + n_{\tilde{t}})(n_w + n_{\tilde{w}}) \leq 1216$; $(h + 1)n_t + hn_{\tilde{t}} = n_{rows}$; $(w + 1)n_w + wn_{\tilde{w}} = n_{cols}$; $h, w, n_t, n_{\tilde{t}}, n_w, n_{\tilde{w}} \in \mathbb{N}$

In the `GeneralBlockStrategy`, in we want to divide the given grid of $n_{rows} \times n_{cols}$ cells using $numTiles$, so that $numTiles <= 1216$, as shown in Figure 5.5. To achieve this, we arrange as many as possible of the 1216 tiles into a 'virtual rectangle of tiles' determined by the aspect ratio $\frac{n_{cols}}{n_{rows}}$ of the input grid.

[2] Consider a grid with 8500 columns, where each cell is a float. Using blocks allows us to fit approximately 8500 rows of data in an IPU's aggregate memory. With a strip-based approach, we could only fit 3648 rows – less than half of the block approach.

The possible tile arrangements are determined by the factorisation of 1216: ($1 \times 1216, 2 \times 608, 4 \times 304, 8 \times 152, 16 \times 76, 19 \times 64, 32 \times 38$, and reflections of these). We find the arrangement that best matches the input grid size, while also taking into account the constraints of the $minRowsPerTile$ and $minColsPerTile$.

$$possibleRatios = \{\frac{1}{1216}, \frac{2}{608}, ..., \frac{608}{2}, \frac{1216}{1}\}$$

$$idealTileGridRatio = \frac{max(1, \lfloor \frac{n_r}{minRowsPerTile} \rfloor)}{max(1, \lfloor \frac{cols}{minColsPerTile} \rfloor)}$$

We pick the ratio $\frac{R}{C} \in possibleRatios$ so that it has the smallest distance $|\frac{R}{C} - idealTileGridRatio|$.

It is unlikely that our input grid will divide perfectly into this arrangement of tiles. Similar to the IPU case above, we use a round-robin allocation so that some tiles will have one more row than others (be "tall") and tiles can also have one more column than others (be "wide") (see Figure 5.5). The dimensions of a "tall and wide" block determines the minimum run time for the IPU program.

Worker level split

Each IPU tile has 6 hardware worker threads, which are run in a time-sliced way [15]. This allows hiding memory latency: it takes 6 cycles to execute a 32-bit load, but when 6 threads are running, on average, each thread looks like it took only 1 "thread cycle" to read, and each thread cannot issue instructions faster than this. While it is possible to schedule more vertexes than there are threads, we want to load balance work between the 6 worker threads so that all vertexes start and finish at the same time, since the busiest worker will determine the length of the compute cycle.

Workers can access all of a tile's memory, so at first it seems that different code is needed for workers with boundaries requiring inter-tile halo-exchange vs those that do not. But in practice, this is not that case. We simply schedule 6 of the same kind of vertex, but with different tensor slices corresponding to worker sub-partitions, and the graph compiler schedules inter-tile exchanges for those areas of the input tensor that are remote. Note that, had the "explicit" style of halo communication been used, we would have had to solve this problem differently. Our halo exchange is designed for non-jagged tensor slices, so worker partitions cannot be jagged.

Because there is no communication cost involved in intra-tile partition boundaries, we can use strips instead of blocks to achieve the most even split with no penalty. To decide whether to split the tile partition vertically or horizontally, we test to see which direction results in the most balanced split and choose either the `LongAndNarrowWorker` strategy or the `ShortAndWideWorker` strategy.

Our final partitioning is represented as a map with keys being the tuples $(ipu, tile, worker)$ and values being slice definitions in the original input grid. Our library includes functions to easily apply a partitioning to an input tensor during vertex placement as well as tools for visualising the partition, and exporting it to JSON format for further analysis.

Good defaults for partition granularity

To determine good values for $minColsPerTile$ and $maxColsPerTile$ we ran a 2200x1100 stencil on 1,2,4,8 and 16 IPUs with a range of minimum block size values (see Figure 5.6) and found that the increase in parallelism from allowing smaller grid partitions outweighed communication costs in almost all cases (but at the price of graph compile time - see Section 8.2). Based on this observation, we recommend choosing the smallest possible block size that makes sense based on other constraints. For example, we show in Section 5.3 that choosing minimum block size of 6x6 results in a simpler vertex implementation and can be balanced between workers, so we use that default for further experiments.

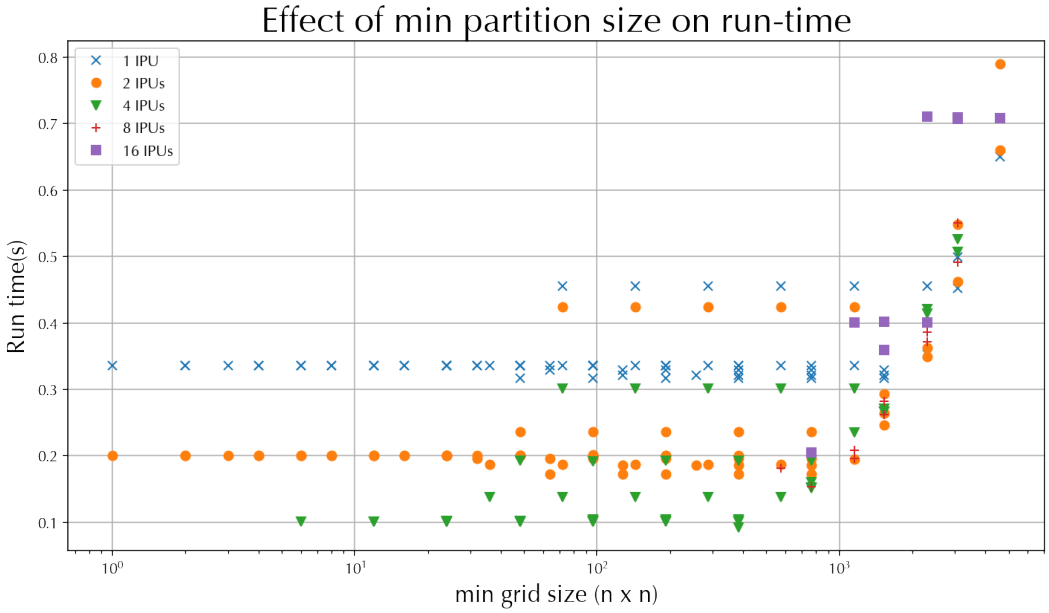

Figure 5.6: Effect of minimum tile partition sizes on execution time for a simple stencil. Smaller minimum tile partition sizes resulted in better run times

Evaluating Load Balance

We define the load balance as

$$LB = \frac{\frac{1}{n}\sum_{i=1}^{n}|P_i|}{\max_i\{|P_i|\}} \tag{5.1}$$

where n is the number of active workers, $|P_i|$ is the size of the ith partition and n_{ipus} is the number of IPUs available.

Figure 5.7 shows the distribution of achievable load balance for 10000 samples of different grid sizes and shapes, taken for 1,2,4,8, and 16 IPUs. We see that almost all configurations can result in high load balance (the median of the samples tested achieved 96.5%).

Where it is possible to choose the input grid shape, choosing an aspect ratio closest to that which can be represented as ratio of two factors of 1216 results in the best tile distribution, and if the tile partition is also divisible by 6, perfect balance can be achieved (e.g. a 2048x1900 grid can be perfectly partitioned into 64x50 blocks on 1216 tiles of 1 IPU, with each worker processing 8 rows).

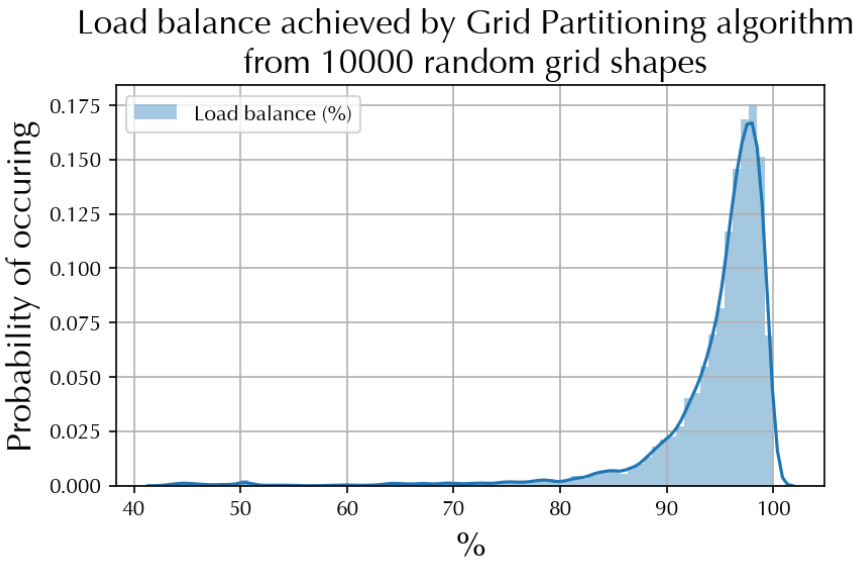

Figure 5.7: Distribution of achievable load balance and parallel efficiency using default minimum tile size (6x6). Measured by partitioning 10000 random grid shapes on 1-16 IPUs. Most shapes should achieve good results using our hierarchical partitioning strategy. Median 96.5%. Actual load balance and parallel efficiency may be be influence by the application.

Extending this implementation to 3D is easy because of first-class support for multidimensional `Tensor` slices in Poplar.

5.3 In-place vs out-of-place update schemes

When the same tensor element is wired up as writeable or read-writeable to two different vertexes in the same compute set, the compiler detects a graph cycle and raises an error. This presents a problem for halo exchange: vertex A cannot read its right halo region if it is the vertex B's left boundary and B is writing to it in the same compute set.

Double-buffered solution

In the simplest solution (double-buffering), we use two copies of the grid data, called A and B, and schedule two compute sets of computation. During an AB phase, we read from A (with halos from A) and write to B. In the subsequent BA phase, we read from B (with halos from B) and write to A.

In many distributed systems (such as implementations based on MPI), this approach allows overlapping the communication of sending halos while starting the next round of computation once halos are received, but with the BSP supersteps, that does not apply for the IPU.

This approach solves the graph cycle (memory clash) elegantly, but doubles the memory requirements. Given the relatively low amount of SRAM available, doubling the memory requirements is a very high price to pay for elegance.

In-place solution

A simple AA scheme is presented in Figure 5.8. The scheme relies on two phases:

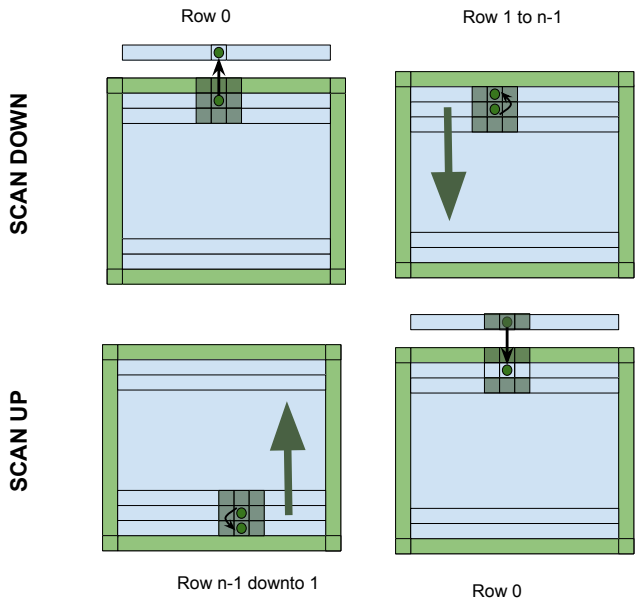

Figure 5.8: Simple two-phase in-place stencil computation scheme

- In the `SCAN_DOWN` phase, we begin from a state where rows are stored in-order, with the top row in index 0. We iterate over the rows assigned to the vertex from top to bottom, applying the stencil, but storing the output in the cell immediately above (the north cell). For the very first row, store the result in a special one-row wide buffer called `tmp`. Note that the last row has no new results.
- In the `SCAN_UP` phase, we iterate over the rows assigned to the vertex from second-to-bottom to top, storing the stencil's result in the row below (the south neighbour cell). For the top row, we apply a special case where we use the north halo, the tmp buffer and the row index 0, and write to row index 0. At the end of this phase, all the results are in their expected place again.

Such direction-changing in-place solutions for stencils are well represented in the literature, especially for the Lattice Boltzmann method [48, 49]. However, we have created a graph cycle because boundary regions are declared as `InOut`. Only the middle region of each partition is safe to use as `InOut`.

Proposed solution for IPU In our extended solution for the IPU, we also store a redundant copy of the boundary region in separate `Tensors` called `borderBottom`, `borderTop`, `borderLeft` and `borderRight`. We also subdivide the data tensor into regions `dataTop`, `dataLeft`, `dataRight`, `dataMiddle` and `dataBottom`. In each phase, regions alternate between being `Inputs` and `Outputs`, while `dataMiddle` is always `InOut` . This arrangement allows us to convey to the compiler which parts of the data tensor are actually changing in the vertex .

At the compute graph conceptual level, we store the grid in a single tensor called `data`, and the borders in 4 additional tensors. During the `SCAN_DOWN` phase, reads only occur from `data`, with writes spread over the safe parts of `data` (i.e. those slices mapped to `dataMiddle` in

Vertexes) and to the border tensors. During the `SCAN_UP` phase, we read from the border tensors and from `dataMiddle`, and can safely write results to `data` tensor (all regions).

This complex dance requires two compute sets, with vertexes wired up to 5 tensors that are mapped to 17 regions. When we use separate boundary regions as in Figure 5.2, there are more than 30 corner cases, making it difficult to optimise vertexes. Even more corner cases emerge if we consider that a partition might be narrow (so that the left border case must also include the right border), or short (top and bottom borders are used at the same time). To avoid these, we impose the limitation that grid partitions are a minimum of 6x6 cells, which also allows use to give one row or column to a worker.

For larger problem sizes, such as a 10000x10000 grid, this AA solution almost halves the memory requirements of the AB case, allowing us to fit significantly larger problems on the IPU. However, the code complexity means that it is best used in automatic code generation, rather than during manual stencil implementation. We did not pursue this AA approach in this work, and stick to the AB scheme, despite the memory usage.

5.4 Optimisations

In this section, we discuss various optimisations to improve the performance of IPU code once a correct stencil program has been implemented. Most stencil optimisations described in the literature try to improve the reuse of the values in a cache and avoid unnecessary loads from memory, but this is wholly inapplicable to the IPU. As we shall see, compared to other platforms, opportunities for "easy wins" are limited.

Vectorisation

Vectorisation is a SIMD technique where processors implement instructions that perform the same operation on contiguous multiples of a data type (vectors) simultaneously. Where it is possible to structure programs to exploit such data parallelism, vectorisation can improve serial program performance considerably (by a constant amount, limited by the vector width). The IPU's ISA describes several vector instructions which operate on 2 `floats`, or 4 `halfs` and (sometimes) 4 `floats` or up to 8 `halfs` simultaneously [15, 5]. These widths are low in comparison with the wide vectors in contemporary CPUs (e.g. 512bits in Fujitsu's A64FX cores).

Rather than accessing these instructions directly by assembly programming, programmers rely on the compiler to generate auto-vectorised code from a standard high-level language such as C++, ensuring portable code. One can hint to the compiler that auto-vectorisation is possible through techniques such as declaring buffers to be memory aligned and non-overlapping, and using pragmas (e.g. `#omp simd`) or intrinsics.

Our experiments showed that `popc` could auto-vectorise loops over standard C++ arrays, but did not auto-vectorise at all for loops over `Input<Vector<...>>` variables, mirroring the findings in [5].

Without writing assembly, accessing the IPU's 64-bit vector instructions is awkward but possible from C++using the `float2` vector type from `<ipudef.h>`. Pointers from the vertex's inputs and outputs must be cast from their original types, and the programmer has to ensure correct memory alignment and handle unaligned loop boundaries.

The IPU supports even wider 128-bit vector widths for some load and store instructions (4 `floats`, or 8 `halfs`), which can only be accessed using assembly.

Despite the awkwardness and invasive code changes, using `float2`s is a worthwhile exercise for some code regions, since we could double the performance of some loops. It is certainly worth pursuing for the "middle" (non-halo) block of the vertex loop, which can be explicitly aligned to support this.

Assembly

Full access to the IPU's optimised instructions is available from assembly, if the programmer is willing to forgo portability completely. Using assembly, we have access to zero overhead loops, vectorised instructions, the optimised convolution and matrix multiplication instructions, and many techniques such as simultaneous load-stores from interleaved memory banks, or executing concurrent memory and arithmetic instructions for doubling loop performance. Jia *et al* [5] point out that using assembly is the only way to get performance close to the theoretical peak.

The IPU documentation for writing Vertexes in assembly [50, 15] is excellent, including a discussion many helpful patterns such as pipelining. But given the learning curve, the difficulty of maintaining and reasoning about the code, and the drop in programmer productivity, this level of optimisation is only for sections of mature kernels where performance is critical.

A reasonable compromise for getting at specialised hardware operations is to instead use Graphcore's library implementations of optimised vertex primitives for operations such as convolution and matrix multiplication. We show one example of this in Chapter 6. There are also some intrinsic wrapper functions available in the SDK, although these are not documented.

Chunked I/O

For many ML problems, multiple small batches of inputs flow through the compute graph. But for many stencil problems, such as those used in PDE solvers, the output of an iteration is fed back as the data for the next iteration, often tens of thousands of times. The input grid's initial state is large relative to the inputs typically encountered in machine learning problems.

The only viable mechanism to load initial conditions into IPU memory is `DataStreams`. However, this allocates buffer memory on the IPU that is the same size of the grid (allowing streaming to happen while computation is in progress), doubling the memory requirements. For an output buffer, the same applies again, so instead of an $n \times n$ array, now we can only fit $\frac{n}{\sqrt{3}} \times \frac{n}{\sqrt{3}}$.

When memory is critical, one pattern we used is to have a small buffer, stream chunks of the initial grid data to it, and `Copy` from there to a slice of the larger input grid. This required more synchronisation, so is slower. But since we typically only perform this operation at the beginning and end of the solver, it is a viable approach when memory is low.

3-phase row scan

The IPU has two instruction pipelines (memory and arithmetic), and arithmetic instructions operands must be registers. We can save some load instructions by structuring code to reuse

values previously loaded into registers.

While performing a 9-point stencil stencil operation, and scanning a row of cells left-to-right, we see that each loop iteration re-uses 6 of the 9 values needed from the previous step. It is relatively trivial to decompose the loop into three sub-phases so that only the new values are loaded, and the old values are kept in the registers where they were.

Although we did not explicitly implement this in assembly, we implemented a C++version which encourages the same use of registers. We found that the performance benefit was minimal. Because the IPU tile has a small number of registers (16 for arithmetic operands), registers are likely to have been needed for other operands before the reuse in the loop is possible. Futhermore, grid cells are also often larger than one register width (e.g. in the Gaussian Blur case, we will use 4 `floats` per cell). We did not pursue this approach beyond basic tests.

Data type

Using a data type which can be represented in fewer bits is an easy way to overcome memory bandwidth limitations, since more grid cells can be loaded and operated on in the same amount of bytes. The IPU's largest data type is a 32-bit float, but a 16-bit `half` is available – which is uncommon in CPU implementations – as well as various integer types. For many algorithms (e.g. image blurring) 16-bit precision may suffice. The IPU also supports hardware stochastic rounding, which gives better results than a strict result for many inherently stochastic use cases (common in machine learning). Since Vertexes are C++, specifying vertexes as template classes allows for easy code re-use.

Unfortunately, the benefits of using lower-precision data types are not automatically unlocked simply by switching `floats` to `halfs` – we also had to rewrite code to use the vectorised data types (`float4`) to see a performance benefit (up to 8x over using 32-bit `floats`).

Data Layout

When cells are multi-valued (e.g. the 4 dimensions of the RGBA data in an image, or the 9 dimensions of the Lattice Boltzmann distributions in Chapter 7), the developer must choose whether to store this dimension first or last in the tensor. For the RGBA image pixel case we use a three dimensional tensor for rows, columns and channel. Storing the channel first corresponds to "SoA" (structure of arrays) storage, so that all the red pixels are stored contiguously, then all the green pixels etc. Storing the channel in the last dimension corresponds to "AoS" (array of structures) storage, so that all the channels for a single pixel are stored contiguously. This decision has a large performance impact on CPU implementations (where we typically want all the values for a cell available together in the cache and choose AoS) and GPU implementations (where coalesced memory access encourages SoA). Refactoring SoA code to AoS code is typically non-trivial.

On the IPU, the choice is less pronounced. Because memory is not loaded in cache lines, access patterns such as striding and contiguous storage have no impact on access latency. The only advantage for choosing AoS over SoA is if the layout would encourage easier vectorisation using the vector data types (e.g. `float2` or `half4`) and allow better use of SIMD instructions. This will vary from case to case.

In our Gaussian Blur implementation (Chapter 6), AoS allows us to load the RGBA `half4` values in one instruction. In our Lattice-Boltzmann implementation (Chapter 7), SoA allows us to very efficiently perform a memory-intensive step, but the data rearrangement costs turn out to negate the benefit and we also use AoS.

5.5 A generic end-to-end stencil program template

In this section, we describe a typical stencil program both at the vertex (codelet) level, and at the compute graph conceptual level, showing how the vertexes are wired for halo exchange in two compute sets, wrapped in a `Repeat` program.

Codelets

With halo stitching. Halo stitching results in the simpler vertex shown in in Listing 5.2. For optimal performance, our inputs are single `Vectors` rather than `VectorLists`, and we use the aligned single-pointer form, allowing elements to be accessed as one contiguous block, or be recast into vector data types like `float2`.

```
template<typename T>
class VertexWithStitching : public Vertex {

#define INIDX(ROW, COL) ((width+2)*((y+1)+ROW)+((x+1)+COL))
#define OUTIDX(ROW, COL) (width*(y+ROW)+(x+COL))+c

public:
 Input <Vector<T, VectorLayout::ONE_PTR>> in;
 Output <Vector<T, VectorLayout::ONE_PTR>> out;
 unsigned width;
 unsigned height;

 bool compute() {
  const T *inPtr = reinterpret_cast<T *>(&in[0]);
  T *outPtr = reinterpret_cast<T *>(&out[0]);
  for (auto y = 0u; y < height ; y++) {
   for (auto x = 0u; x < width ; x++) {
     outPtr[OUTIDX(0, 0)] = stencil(
       inPtr[INIDX(-1,-1)], inPtr[INIDX(-1,0)], inPtr[INIDX(-1,1)],
       inPtr[INIDX(0,-1)], inPtr[INIDX(0,0)], inPtr[INIDX(0,1)],
       inPtr[INIDX(1,-1)], inPtr[INIDX(1,0)], inPtr[INIDX(1,1)]
     );
   }
  }
  return true;
 }
};
```

Listing 5.2: Outline of a stencil vertex's codelet

Without halo stitching. Codelets for stencil compute vertexes that do not stitch halos follow the pattern shown Listing 5.3. Each vertex declares separate `Inputs` for its partition, and its each of its halos. We define an `Output`, which is the same size are the partition (double-buffered approach).

```
class StencilVertexNxM : public Vertex {
public:
    Input <Vector<float>> n,s,e,w,nw,ne,sw,se; // halos (sent from other tiles)
    Input <Vector<float>> in; // partition data (stored on this tile)
    Output<Vector<float>> out; // double-buffer data (stored on this tile)
    unsigned width, height;
```

```
7
8     bool compute() {
9         ...
10        #define OFFSET(r,c) (y + r) * width + (x + c)
11        for (auto y = 1u; y < height - 1; y++) {
12            for (auto x = 0u; x < width - 1; x++) { // Region 8
13                const auto idx = x + width * y;
14                out[idx] = stencil(in[OFFSET(-1,-1)], in(OFFSET(-1,0), ...);
15            }
16        }
17        ... // and similar for each loop region, but gathering
18        // neighbours from correct data structures
19        for (auto y = 1u; y < height - 1; y++) {
20           constexpr auto x = 0u; // Region 9
21        ...
22
23    }
```

Listing 5.3: Outline of a stencil vertex's codelet

Because halos are in different data structures to the main partition data, looping over all the different pixels is awkward and requires either inefficient indirect addressing, or defining separate loops for each of the regions in Figure 5.2. In each loop region, we gather the neighbours of the current cell and call the `stencil` function, an example of which is shown in Listing 6.2, If partitions can be restricted to being at least 4x4, only the $n \times m$ codelet in Figure 5.2 needs to be implemented.

Compute Graph

Tensors We take a double-buffered approach and define `Tensors` for the `data` and the temporary scratch `data_tmp`.

Compute sets We define two compute sets, `in2out` and `out2in`, which use the same vertexes and tensors, but with `data` and `data_tmp` swapped as `Inputs` and `Outputs` in the vertex wiring. This means that memory clashes are avoided and execution is naturally split into two phases.

Vertexes and wiring For every $(ipu, tile, worker)$ in the partitioning, we instantiate 2 `Vertexes` (or the correct variant from Figure 5.2 depending on the partition size). We wire one vertex in compute set `in2out` with the halo regions and the partition being mapped to slices of `data`, and the output being mapped to the corresponding slice of `data_tmp`. We wire up the other vertex in compute set `out2in` with the halo regions and partition mapped to slices of `data_tmp` and the output to the corresponding slice of `data`. The partitions and halos are calculated using our libraries that implement the algorithms discussed in 5.2.

Iteration We implement iteration by wrapping the execution of the vertexes in a `Repeat` block, as shown in Listing 5.4.

```
auto stencilProgram = Repeat{numIters, Sequence{Execute{in2out}, Execute{out2in}}};
```

Listing 5.4: Wrapping the compute sets in a Repeat block

Host program

Our generic host program parses input arguments, requests an IPU device, creates a `Graph` object, registering the codelets from Section 5.5 with the graph, builds the compute graph as discussed above, and creates `DataStreams` for transferring initial data and reading the output. It then creates an `Engine`, loads the compiled `Graph` to the `Engine`, and run the programs for host-to-IPU data transfer, stencil iteration, and IPU-to-host transfer, and processes the program output.

5.6 Summary

In this chapter, we presented ways to implement prerequisites for stencils on the IPU: halo exchange and grid partitioning.

After implementing a correct stencil, the programmer is left with few easy options for further increasing speed, compared to other platforms where memory access costs are paramount. Without writing assembly, the only obvious optimisation comes from exploiting the vector data types, but these offer fewer benefits compared to the wide vector instructions available on contemporary CPU and GPUs.

Communication costs must be measured and optimised per application. Halo exchange can be improved by specifying separate tensors for each border region rather than concatenating ("stitching") regions during wiring. This results in faster execution, but both the vertex and wiring code become much more complex. Vectorisation and memory alignment may introduce rearrangement costs that offset any gains from border separation. Finally, we presented the outline of a generic end-to-end stencil application involving double buffering, which serves as the starting point for our next applications.

Bibliography

[1] jarro_2783. *cxxopts (C++ argument parsing library)*. 2020. URL: https://github.com/jarro2783/cxxopts.

[2] L. Vandevenne. *LodePNG (PNG image processing library)*. 2019. URL: https://lodev.org/lodepng/.

[3] C. Rau. *half (C++ IEEE-754 16 bit precision library)*. 2019. URL: https://sourceforge.net/projects/half.

[4] D. Amodei and D. Hernandez. *AI and Compute*. OpenAI Blog Post. June 2018. URL: https://openai.com/blog/ai-and-compute/.

[5] Z. Jia et al. 'Dissecting the Graphcore IPU Architecture via Microbenchmarking'. In: *arXiv preprint: 1912.03413* (2019). URL: http://arxiv.org/abs/1912.03413.

[6] C. Ding and Y. He. 'A Ghost Cell Expansion Method for Reducing Communications in Solving PDE Problems'. In: *PROCEEDINGS OF SC2001*. ACM Press, 2001.

[7] V. Bandishti, I. Pananilath, and U. Bondhugula. 'Tiling Stencil Computations to Maximize Parallelism'. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '12. Salt Lake City, Utah: IEEE Computer Society Press, 2012. ISBN: 9781467308045.

[8] K. Datta et al. 'Stencil Computation Optimization and Auto-Tuning on State-of-the-Art Multicore Architectures'. In: *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. SC '08. Austin, Texas: IEEE Press, 2008. ISBN: 9781424428359.

[9] S. Kamil et al. 'Implicit and explicit optimizations for stencil computations'. In: *Proceedings of the 2006 workshop on Memory system performance and correctness*. 2006, pp. 51–60.

[10] T. Muranushi and J. Makino. 'Optimal Temporal Blocking for Stencil Computation'. In: *Procedia Computer Science* 51 (2015). International Conference On Computational Science, ICCS 2015, pp. 1303–1312. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2015.05.315.

[11] J. Ortiz et al. 'Bundle Adjustment on a Graph Processor'. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2020)*. 2020. URL: http://arxiv.org/abs/2003.03134.

[12] I. Kacher et al. *Graphcore C2 Card performance for image-based deep learning application: A Report*. 2020. arXiv: 2002.11670 [cs.CV].

[13] L. R. M. Mohan et al. 'Studying the potential of Graphcore IPUs for applications in Particle Physics'. In: *arXiv preprint arXiv:2008.09210* (2020).

[14] L. G. Valiant. 'A Bridging Model for Parallel Computation'. In: *Communications of the ACM* 33.8 (Jan. 1990), pp. 103–111. ISSN: 15577317. DOI: 10.1145/79173.79181.

[15] A. Alexander and J. Mangnall. *Colossus GC2 Tile Worker ISA 1.1.3*. Tech. rep. Graphcore, Oct. 19, 2019.

[16] M. Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*. 2015. URL: www.tensorflow.org.

[17] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. 'Global Arrays: A Portable "Shared-Memory" Programming Model for Distributed Memory Computers'. In: *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*. Supercomputing '94. Washington, D.C.: IEEE Computer Society Press, 1994, pp. 340–349. ISBN: 0818666056.

[18] D. Lacey. *Intelligent Memory for intelligent computing*. Graphcore Blog post. 2020. URL: https://www.graphcore.ai/posts/intelligent-memory-for-intelligent-computing.

[19] S. W. Chien et al. 'TensorFlow Doing HPC: An Evaluation of TensorFlow Performance in HPC applications'. In: *Proceedings - 2019 IEEE 33rd International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2019*. Institute of Electrical and Electronics Engineers Inc., May 2019, pp. 509–518. ISBN: 9781728135106. DOI: 10.1109/IPDPSW.2019.00092.

[20] K. Asanović et al. *The Landscape of Parallel Computing Research: A View from Berkeley*. Tech. rep. UCB/EECS-2006-183. EECS Department, University of California, Berkeley, Dec. 2006. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html.

[21] M. McCool, A. D. Robison, and J. Reinders. 'Chapter 7 - Stencil and Recurrence'. In: *Structured Parallel Programming*. Ed. by M. McCool, A. D. Robison, and J. Reinders. Boston: Morgan Kaufmann, 2012, pp. 199–207. ISBN: 978-0-12-415993-8. DOI: https://doi.org/10.1016/B978-0-12-415993-8.00007-4.

[22] B. J. Palmer and J. Nieplocha. 'Efficient Algorithms for Ghost Cell Updates on Two Classes of MPP Architectures.' In: *IASTED PDCS*. 2002, pp. 192–197.

[23] M. Frigo and V. Strumpen. 'Cache Oblivious Stencil Computations'. In: *Proceedings of the 19th Annual International Conference on Supercomputing*. ICS '05. Cambridge, Massachusetts: Association for Computing Machinery, 2005, pp. 361–366. ISBN: 1595931678. DOI: 10.1145/1088149.1088197.

[24] A. Nguyen et al. '3.5d blocking optimization for stencil computations on modern CPUs and GPUs'. In: *in Proc. of the 2010 ACM/IEEE Int'l Conf. for High Performance Computing, Networking, Storage and Analysis, 2010*, pp. 1–13.

[25] J. Treibig, G. Wellein, and G. Hager. 'Efficient multicore-aware parallelization strategies for iterative stencil computations'. In: *Journal of Computational Science* 2.2 (2011). Simulation Software for Supercomputers, pp. 130–137. ISSN: 1877-7503. DOI: https://doi.org/10.1016/j.jocs.2011.01.010.

[26] C. Yang. *8 Steps to 3.7 TFLOP/s on NVIDIA V100 GPU: Roofline Analysis and Other Tricks*. 2020. arXiv: 2008.11326 [cs.DC].

[27] M. Steuwer et al. 'High-Level Programming of Stencil Computations on Multi-GPU Systems Using the SkelCL Library'. In: *Parallel Processing Letters* 24 (Sept. 2014). DOI: 10.1142/S0129626414410059.

[28] Y. Tang et al. 'The Pochoir Stencil Compiler'. In: SPAA '11. San Jose, California, USA: Association for Computing Machinery, 2011, pp. 117–128. ISBN: 9781450307437. DOI: 10.1145/1989493.1989508.

[29] M. Christen, O. Schenk, and H. Burkhart. 'Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures'. In: *2011 IEEE International Parallel & Distributed Processing Symposium*. IEEE. 2011, pp. 676–687.

[30] M. Bauer. *pystencils*. 2020. URL: https://i10git.cs.fau.de/pycodegen/pystencils.

[31] J. Ragan-Kelley et al. 'Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines'. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. Seattle, Washington, USA: Association for Computing Machinery, 2013, pp. 519–530. ISBN: 9781450320146. DOI: 10.1145/2491956.2462176.

[32] S. Rajopadhye et al. *The Stencil Processing Unit: GPGPU Done Right*. Tech. rep. CS-13-103. Computer Science Department, Colorado State University, Fort Collins, CO 80523-1873., 2013.

[33] S. Williams, A. Waterman, and D. Patterson. *Roofline: An Insightful Visual Performance Model for Multicore Architectures*. New York, NY, USA, Apr. 2009. DOI: 10.1145/1498765.1498785.

[34] J. D. McCalpin. 'Memory bandwidth and machine balance in current high performance computers'. In: *IEEE computer society technical committee on computer architecture (TCCA) newsletter* 2 (19-25 1995).

[35] T. Deakin et al. 'GPU-STREAM v2.0: Benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models'. In: *Paper presented at P^3MA Workshop at ISC High Performance* (2016).

[36] T. Deakin, J. Price, and S. McIntosh-Smith. 'Portable methods for measuring cache hierarchy performance'. In: *IEEE/ACM Super Computing* (2017).

[37] Graphcore. *Graphcore 740 IPU Server Product Brief*. 2020. URL: https://www.graphcore.ai/hubfs/assets/pdf/740%20Product%20Brief.pdf.

[38] A. Ilic, F. Pratas, and L. Sousa. 'Cache-Aware Roofline Model: Upgrading the Loft'. In: *IEEE Comput. Archit. Lett.* 13.1 (Jan. 2014), pp. 21–24. ISSN: 1556-6056. DOI: 10.1109/L-CA.2013.6. URL: https://doi.org/10.1109/L-CA.2013.6.

[39] O. G. Lorenzo et al. '3DyRM: a dynamic roofline model including memory latency information'. In: *The Journal of Supercomputing* 70.2 (2014), pp. 696–708.

[40] V. C. Cabezas and M. Püschel. 'Extending the roofline model: Bottleneck analysis with microarchitectural constraints'. In: *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2014, pp. 222–231.

[41] J. Choi et al. 'A Roofline Model of Energy'. In: *Proceedings - IEEE 27th International Parallel and Distributed Processing Symposium, IPDPS 2013*. IEEE. May 2013, pp. 661–672. DOI: 10.1109/IPDPS.2013.77.

[42] D. Cardwell and F. Song. 'An Extended Roofline Model with Communication-Awareness for Distributed-Memory HPC Systems'. In: *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*. Jan. 2019, pp. 26–35. ISBN: 9781450366328. DOI: 10.1145/3293320.3293321.

[43] C. Yang, T. Kurth, and S. Williams. 'Hierarchical Roofline analysis for GPUs: Accelerating performance optimization for the NERSC-9 Perlmutter system'. In: *Concurrency and Computation: Practice and Experience* (Nov. 2019). DOI: 10.1002/cpe.5547.

[44] A. Lopes et al. 'Exploring GPU performance, power and energy-efficiency bounds with Cache-aware Roofline Modeling'. In: *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2017, pp. 259–268.

[45] G. Karypis, K. Schloegel, and V. Kumar. 'Parmetis: Parallel graph partitioning and sparse matrix ordering library'. In: *Version 1.0, Dept. of Computer Science, University of Minnesota* 22 (1997).

[46] D. G. Wonnacott and M. M. Strout. 'On the scalability of loop tiling techniques'. In: *IMPACT 2013* 3 (2013).

[47] V. Bandishti, I. Pananilath, and U. Bondhugula. 'Tiling stencil computations to maximize parallelism'. In: *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE. 2012, pp. 1–11.

[48] J. Latt. 'Technical report: How to implement your DdQq dynamics with only q variables per node (instead of 2q)'. In: *Tufts University* (2007), pp. 1–8.

[49] M. Geier and M. Schoenherr. 'Esoteric twist: an efficient in-place streaming algorithmus for the lattice Boltzmann method on massively parallel hardware'. In: *Computation* 5.2 (2017), p. 19.

[50] Graphcore. *Vertex Assembly Programming Guide*. Tech. rep. 2020. URL: https://docs.graphcore.ai/projects/assembly-programming/en/latest/.

[51] beedieu. *Bricks*. Image licensed under CC BY-NC-SA 2.0. URL: https://search.creativecommons.org/photos/b5b0bad3-b1fa-472c-acec-297a6826a2f4.

[52] yelsnia. *Leaf*. Image licensed under CC BY-SA 2.0. URL: https://search.creativecommons.org/photos/c1d7580b-8388-442b-a0ff-f365887ee8ce.

[53] R. Dimov. *Say Cheese Tromp Cheese deli in Amsterdam*. Image licensed under CC BY-SA 3.0. URL: https://search.creativecommons.org/photos/3a95a82e-542e-471b-b36b-9a9302f9865e.

[54] X. He and L.-S. Luo. 'A priori derivation of the lattice Boltzmann equation'. In: *PHYSICAL REVIEW E* 55 (June 1997), R6333–R6336. DOI: 10.1103/PhysRevE.55.R6333.

[55] G. Wellein et al. 'On the single processor performance of simple lattice Boltzmann kernels'. In: *Computers & Fluids* 35.8 (2006). Proceedings of the First International Conference for Mesoscopic Methods in Engineering and Science, pp. 910–919. ISSN: 0045-7930. DOI: https://doi.org/10.1016/j.compfluid.2005.02.008.

[56] M. Bauer et al. 'waLBerla: A block-structured high-performance framework for multiphysics simulations'. In: *Computers & Mathematics with Applications* (2020). ISSN: 0898-1221. DOI: https://doi.org/10.1016/j.camwa.2020.01.007.

[57] M. Wittmann et al. 'Comparison of different propagation steps for lattice Boltzmann methods'. In: *Computers & Mathematics with Applications* 65.6 (2013). Mesoscopic Methods in Engineering and Science, pp. 924–935. ISSN: 0898-1221. DOI: https://doi.org/10.1016/j.camwa.2012.05.002.

[58] C. Körner et al. 'Parallel Lattice Boltzmann Methods for CFD Applications'. In: *Lecture Notes in Computational Science and Engineering*. Vol. 51. Jan. 2006, pp. 439–466. DOI: 10.1007/3-540-31619-1_13.

[59] S. Williams et al. 'Lattice Boltzmann Simulation Optimization on Leading Multicore Platforms'. In: *International Parallel and Distributed Processing Symposium (IPDPS)*. Apr. 2008. DOI: 10.1109/IPDPS.2008.4536295.

[60] C. Feichtinger et al. 'Performance modeling and analysis of heterogeneous lattice Boltzmann simulations on CPU–GPU clusters'. In: *Parallel Computing* 46 (2015), pp. 1–13. ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2014.12.003.

[61] J. Latt et al. 'Palabos: Parallel Lattice Boltzmann Solver'. In: *Computers & Mathematics with Applications* (2020). ISSN: 0898-1221. DOI: https://doi.org/10.1016/j.camwa.2020.03.022.

[62] M. Martineau et al. 'An Evaluation of Emerging Many-Core Parallel Programming Models'. In: (2016), pp. 1–10. DOI: 10.1145/2883404.2883420. URL: http://www.bristol.ac.uk/pure/about/ebr-terms.

[63] C. Lattner and V. Adve. 'LLVM: A compilation framework for lifelong program analysis & transformation'. In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE. 2004, pp. 75–86.

[64] G. Guennebaud, B. Jacob, et al. 'Eigen: A C++ template library for linear algebra'. In: *http:/eigen. tuxfamily. org* (2014).

[65] K. Devine et al. 'Getting Started with Zoltan: A Short Tutorial'. In: *Proc. of 2009 Dagstuhl Seminar on Combinatorial Scientific Computing*. Also available as Sandia National Labs Tech Report SAND2009-0578C. 2009.