# Spatial computation on a homogeneous, many-core architecture

Daniel Bates Alex Bradbury Andreas Koltes Robert Mullins
University of Cambridge
Firstname.Lastname@cl.cam.ac.uk

## 1. Introduction

With abundant transistors but limited energy budgets, chip designs have trended towards multiple cores and specialised logic which is used infrequently. This approach allows computer architects to sidestep the utilisation wall: the idea that we can place more transistors on a chip than we can use simultaneously. This suggests that transistors' functions should be specialised, so only a small fraction of the chip need be active at a time.

However, as trade-offs continue to change, this approach will become less effective. Increasing heterogeneity increases complexity, and this makes it harder to validate the chip's design; harder to generate optimised code; and harder to protect against hardware faults. Furthermore, beyond 28nm, we can no longer assume that smaller transistors will always be cheaper, so we cannot continue to provide dedicated logic which will be used infrequently. Instead, we propose switching to a homogeneous approach, and implementing the necessary specialisation in software. Having a single computation unit which is repeated many times reduces complexity and so makes the problems of validation, compilation and fault tolerance easier to solve. Homogeneous systems have the additional advantage that they are general-purpose, so a wider range of applications can be usefully accelerated.

The challenge then becomes: how do we make use of all the available processors? A thread-based approach will only get us so far. Thread-level parallelism (TLP) is only abundant in a small fraction of code, and TLP in general applications has remained stubbornly low [3]. Instead, we show that if communication between cores is low-latency and low-energy, large numbers of them can be grouped together at run-time to implement a *virtual architecture* optimised for a particular application. This virtual architecture can be given the ideal cache capacity, communication structure and number of functional units to execute a task efficiently. Since the underlying architecture is homogeneous, there is also scope for dynamically varying the resources allocated, depending on circumstances such as contention, priority and power budget.

## 2. Loki architecture

We use the Loki architecture [2] as a test platform and explore different ways in which a selection of applications can be mapped to the architecture. Loki has a hierarchical homogeneous structure

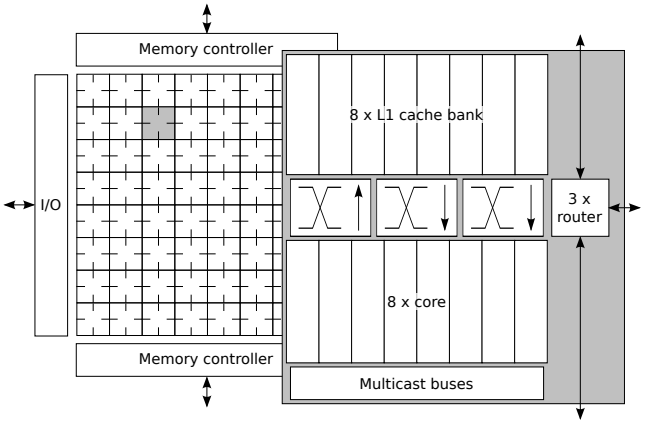


**Figure 1.** Loki's tiled architecture. Left: chip with one tile highlighted. Right: tile block diagram.

(Figure 1). At the top level, the chip is composed of a number of tiles. Each tile is identical, and is connected to its four nearest neighbours with three separate mesh networks. Each tile contains a number of identical cores, cache banks and multiple networks allowing them to communicate. Cores are designed to be very simple. This allows them to be smaller and more energy efficient, which in turn makes communication between cores cheaper in both time and energy.

Cores within a tile can communicate with each other in a single clock cycle over a multicast-capable crossbar network. They can also communicate with any cache bank on the tile, with a round-trip latency of two clock cycles. Tiles are sized to maximise the number of cores and cache banks reachable within a single cycle, while keeping within a two-cycle cache latency. Each tile also contains three single-cycle routers which can communicate with neighbouring tiles' routers with a latency of one cycle. Using a commercial 40nm low-power process, each tile can contain eight cores and eight 8kB cache banks, with a total size of 1mm$^2$. With conservative timing margins at the worst-case corner, a frequency of 435MHz is achieved. This corresponds to roughly 42 FO4 delays: within the typical range of 40-60 used by modern system-on-chip designs.

When cores are abundant, they can be used in unconventional ways. In previous work [2], we showed how cores within a tile could be configured to flexibly exploit any combination of instruction-level, data-level and thread-level parallelism. They can be used to provide services to other nearby cores, such as executing common functions or prefetching data into the local cache. They can also be used to increase instruction cache capacity or the number of available registers, for example. In this work, we extend this investigation across multiple tiles. It may be reasonable, for example, to reduce the density of computation by leaving some cores


*PRISM-2*, June 14, 2014, Minneapolis, MN, USA.

http://dx.doi.org/10.1145/

idle, and allowing their neighbours to use a larger fraction of the local cache capacity, network bandwidth and energy budget. We take the RISC philosophy of optimising for the common case, but retain the flexibility necessary to support applications which aren't common, and need a different balance of resources.

For various mappings of applications to cores, we model performance, energy and area of our architecture using our previous methodology [2]. Different application mappings allow us to target different performance, energy or area requirements. For example, matrix multiplication can be distributed across cores in a vector-like fashion to allow each instruction to be fetched by one core but executed by many, or using a systolic array structure to reduce the number of memory accesses required and increase specialisation of each core's task. We demonstrate the advantages of using a homogeneous architecture for computation, and identify bottlenecks which limit the efficiency of some implementations.

When communicating between tiles, we make use of end-to-end credit-based flow control to avoid the possibility of deadlock. We are able to guarantee that if data is put onto the network, it will always be removed again at its destination, and so will not hold up other messages indefinitely. End-to-end flow control was not a limiting factor for any of the benchmarks used in this paper, but is often dismissed as being too restrictive [4]. We plan to test this claim and explore possible ways of relaxing the flow control requirements in future work.

## 3. Case studies

We perform case studies on three simple benchmarks to help explore the features required to use large numbers of cores efficiently. For each application, we present the performance and energy consumption for a range of different mappings to the Loki architecture, and provide figures for the same application on a 700MHz ARM1176JZF-S processor implemented on the same 40nm process. Comparable results from other architectures are presented where available.

### 3.1 Sort

One possible way of parallelising a sort is to have each core sort a separate block of the input data, and then perform a final merge. With an architecture capable of message-passing, the final merge looks a lot like a dataflow graph (Figure 2). On Loki, in order to keep as much communication on the cheaper local networks, we use seven of the eight cores on each tile to form the first three levels of the merge tree, and use the final core from each tile to create the rest of the tree. We use the quicksort implementation from the newlib C standard library [6] to sort each individual block.

Figure 3 shows how energy and performance change as the number of cores change. The parallel quicksort phase of the algorithm demonstrates an almost perfect speedup as more cores are introduced, and energy slowly decreases as more of the sorting is done using the efficient merge network. The final merge phase, however, has a serialisation point at the root of the tree, which forms a bottleneck when there are four or more cores in the tree. The only way to improve the performance of this merge phase is to reduce the length of the critical path of the merge code, which currently stands at nine instructions.

The triggered instructions paradigm gives each instruction a trigger consisting of a number of different predicates [7]. Instructions are executed as soon as their predicates are satisfied. Predicates can be complex: an example trigger for a mergesort application is, "both lists have data remaining, and the first element of the first list is smaller than that of the second". Using such a paradigm, it is possible to have a critical path of two instructions per iteration.

While moving to this paradigm is beyond the scope of this work, we have identified a number of useful features of triggered

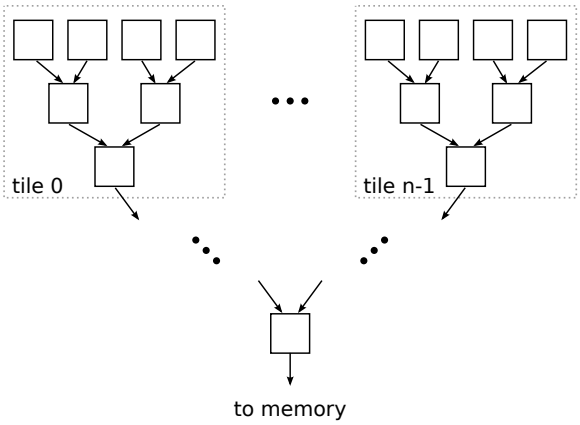


**Figure 2.** Spatial layout of the final merge phase of a parallel sorting algorithm.

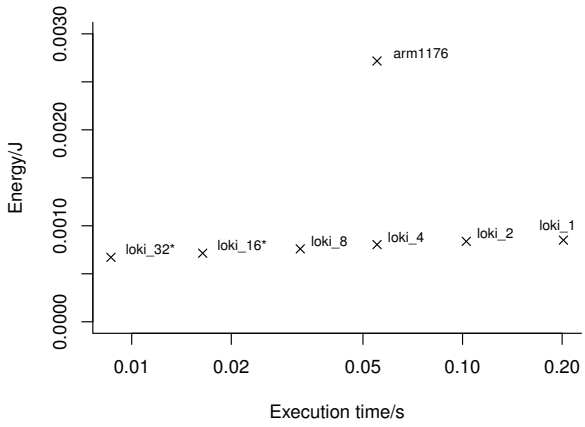


**Figure 3.** Performance and energy consumption when sorting $2^{16}$ integers. Starred points are projections based on trends seen for other input sizes.

instructions which would be possible to transfer to communication-centric architectures such as Loki, and would help reduce code size:

- Ensure that network communication is blocking, removing the need to explicitly check whether data has arrived or whether data can be sent.
- Provide a *peek* operation for the cores' input network buffers, removing the need to copy data into the register file.
- Provide data streams which expose an end-of-stream marker to software. In some cases, this removes the need to maintain an iteration counter.
- Add a simplified form of triggers. An example is a branch instruction which specifies two simple triggers. The trigger which fires first determines the branch to be taken.

Of these, Loki currently only supports blocking network communication. We are looking into supporting the other features. With all additional features implemented, it would be possible to reduce the critical path on Loki to four instructions, effectively separating the two triggered instructions into two triggers and two actions.

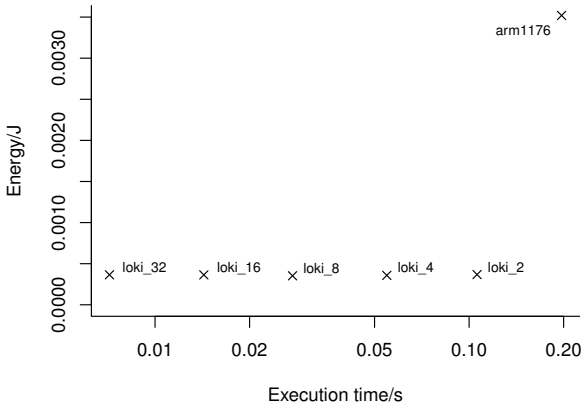


**Figure 4.** Performance and energy consumption when computing the FFT of $2^{16}$ complex fixed-point values. Additional comparison: 40nm Broadcom VideoCore IV GPU at 250MHz: 6.7ms, unknown energy (single-precision floating point) [9].

### 3.2 Fast Fourier Transform

The FFT has a predictable but cache-unfriendly memory access pattern. Each core must be able to read and write to almost any part of the given data, and so coherence is required.

The Loki architecture does not currently support hardware cache coherence due to its overhead in terms of energy and area. This approach agrees with the general consensus among many-core architectures: full coherence is expensive, and is only necessary for a small percentage of memory accesses [5, 10].

The SWEL protocol addresses this problem by keeping shared data in the lowest-level cache which is accessible by all sharers [8]. The results of using this approach on Loki are presented in Figure 4, assuming a fixed 20-cycle latency from the level 1 cache on each tile to the shared L2 cache.

Performance increases linearly as the number of cores increases, and energy remains mostly unchanged. This is because the same work is being done, only distributed over many cores.

In this particular benchmark, memory access patterns are predictable, so prefetching could be used to hide latency and improve performance further. This won't be possible in all benchmarks, however. Instead, we intend to provide a flexible memory system which maintains a single chip-wide copy of any data which must remain coherent, and make extensive use of message passing to allow cores to communicate data directly between each other.

### 3.3 Matrix multiplication

Matrix multiplication is an embarrassingly parallel problem: each element of the output matrix can be computed independently.

The most basic mapping of the algorithm is to treat the many cores like a single vector processor, and have them all execute the same code simultaneously to compute different parts of the output.

Since each Loki core can act independently, it is possible to scalarise this code, and extract common code such as iteration counting to a helper core [1]. This reduces the amount of code executed by the remaining cores, at the expense of having fewer cores directly computing the output. On Loki, we use one helper core per tile, as this is the range over which multicast communication is possible.

A systolic array is a grid of processing units which streams different data sets in different directions. Figure 5 shows how two $4 \times 4$ matrices can be multiplied using such a structure. Each core receives a different combination of data from the two input matrices, and so produces a different section of the output. The resultant matrix is drained out when computation has finished. In practice, multiple rows or columns of data are typically passed along each row or column of cores to reduce the number of cores required.

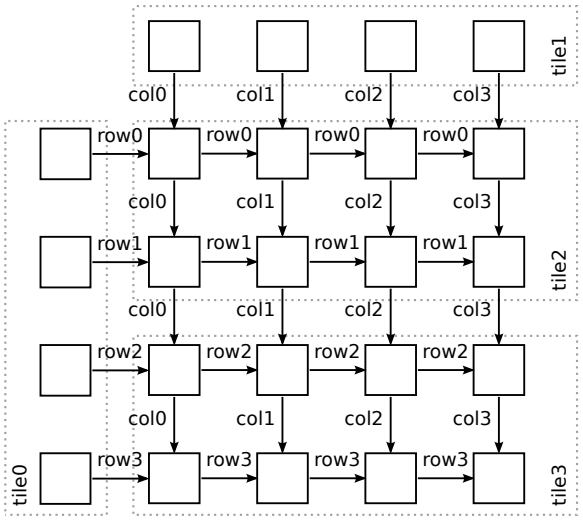


**Figure 5.** Spatial layout of a systolic array for matrix multiplication.

When implementing a systolic array on Loki, the cores are again laid out to minimise the requirement to transfer information between tiles. With the intensity of communication in this benchmark, however, this still leaves up to six incoming and six outgoing data streams per compute tile, and eight outgoing streams for the tiles providing the data. In this case, network bandwidth becomes the bottleneck.

One solution to this problem would be to provide a network with a higher bandwidth. This is likely to be an unnecessary overhead for the benchmarks which do not require it, however. Another more-general solution is to change the way the benchmark is mapped to the hardware. Since cores are abundant on the Loki architecture, we explore using a sparse mapping which only uses half the cores on each tile. This reduces the maximum number of incoming or outgoing data streams to four for all tiles, and effectively creates a virtual architecture which has an increased global network bandwidth. The unused cores on each tile could potentially be used for other computation which does not require data from the network, but this possibility is not explored here.

Figure 6 compares the different mappings. It can be seen that the default vector mapping scales linearly, as expected, and energy remains roughly constant. The energy-efficiency of mappings which use a helper core increases up to eight cores, as the helper is helping more cores simultaneously. Beyond this point, additional helper cores are needed, and no further efficiency gains are realised. The systolic array structure requires a large number of cores to multiply matrices of this size, but does not appear to give a proportional benefit. The sparser implementation helps a lot, but for this benchmark, a larger vector or scalarised implementation would probably be a better option.

## 4. Conclusion

We have shown that it is possible to make use of large numbers of cores when executing embedded application kernels. The use of a general-purpose fabric allows applications to be mapped to cores in a range of different ways which may not be possible on more-specialised architectures. This gives the programmer or compiler

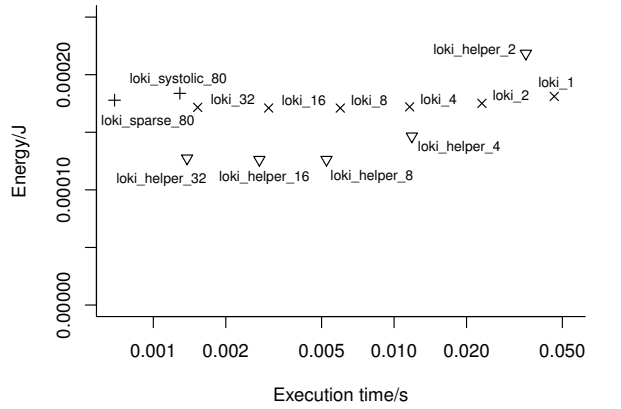


**Figure 6.** Performance and energy consumption when multiplying two $128 \times 128$ matrices. ARM1176 result omitted for clarity: 52ms, 2.4mJ.

more freedom to choose the energy-performance tradeoff which is best, given the current circumstances.

Loki is a broad project: current plans include investigating simple, configurable accelerators within each core; new compilation techniques and further optimisations; operating system support; and fabrication and distribution of Loki test chips and development boards.

## Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful comments and guidance. This work was supported by EPSRC grant EP/G033110/1 and ERC grant 306386.

## References

[1] K. Asanovic, S. W. Keckler, Y. Lee, R. Krashinsky, and V. Grover. Convergence and scalarization for data-parallel architectures. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–11, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-1-4673-5524-7. . URL `http://dx.doi.org/10.1109/CGO.2013.6494995`.

[2] D. Bates, A. Bradbury, A. Koltes, and R. Mullins. Exploiting tightly-coupled cores. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 2013 International Conference on*, pages 296–305, 2013. .

[3] G. Blake, R. G. Dreslinski, T. Mudge, and K. Flautner. Evolution of thread-level parallelism in desktop applications. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 302–313, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0053-7. . URL `http://doi.acm.org/10.1145/1815961.1816000`.

[4] A. Hansson, K. Goossens, and A. Rădulescu. Avoiding message-dependent deadlock in network-based systems on chip. *VLSI design*, 2007.

[5] H. Huang, N. Yuan, W. Lin, G. Long, F. Song, L. Yu, Y. Liu, L. Liu, Y. Zhou, X. Ye, et al. Architecture supported synchronization-based cache coherence protocol for many-core processors. In *CMP-MSI08, ISCA Workshop*, 2008. URL `https://www.cs.utah.edu/cmpmsi08/paper7.pdf`.

[6] J. Johnston and T. Fitzsimmons. The newlib homepage. http://sourceware.org/newlib/, 2011. URL `http://sourceware.org/newlib/`.

[7] A. Parashar, M. Pellauer, M. Adler, B. Ahsan, N. Crago, D. Lustig, V. Pavlov, A. Zhai, M. Gambhir, A. Jaleel, R. Allmon, R. Rayess, S. Maresh, and J. Emer. Triggered instructions: A control paradigm for spatially-programmed architectures. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 142–153, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2079-5. . URL `http://doi.acm.org/10.1145/2485922.2485935`.

[8] S. H. Pugsley, J. B. Spjut, D. W. Nellans, and R. Balasubramonian. Swel: Hardware cache coherence protocols to map shared data onto shared caches. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 465–476, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0178-7. . URL `http://doi.acm.org/10.1145/1854273.1854331`.

[9] E. Upton. Accelerating Fourier transforms using the GPU. http://www.raspberrypi.org/archives/5934, Jan 2014. URL `http://www.raspberrypi.org/archives/5934`.

[10] X. Zhou, H. Chen, S. Luo, Y. Gao, S. Yan, W. Liu, B. Lewis, and B. Saha. A case for software managed coherence in manycore processors. In *Poster on 2nd USENIX Workshop on Hot Topics in Parallelism HotPar10*, 2010. URL `http://usenix.org/events/hotpar10/final_posters/Zhou.pdf`.