Advanced Communication Pattern Support for Hardware Accelerators

Guillem López-Paradís[12], Nazerke Turtayeva[3], Guy Wilks[4], Tianrui Wei[5],
Vicenç Beltran[1], Adrià Armejach[12], Miquel Moretó[12], Jonathan Balkind[3]
[1] BSC [2] UPC [3] UC Santa Barbara [4] CMU [5] UC Berkeley

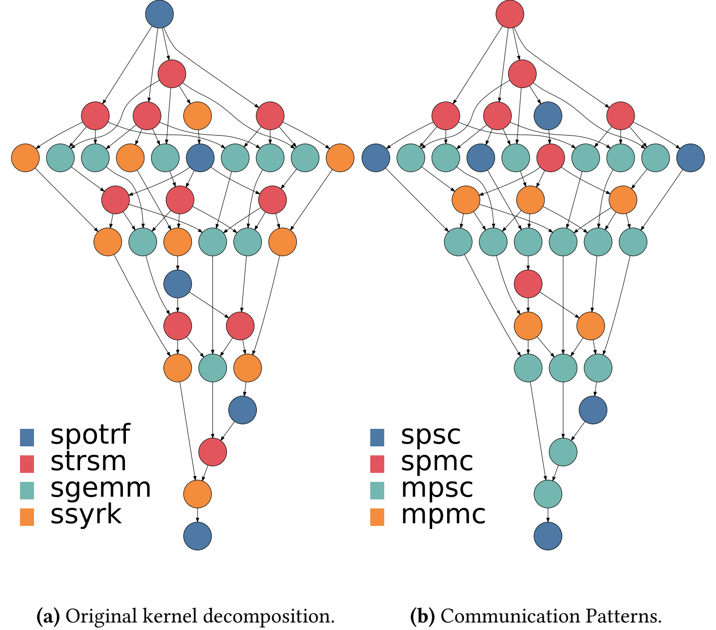
1 Problem statement

Today's systems-on-chip (SoCs) feature many application-specific accelerators that offer significant performance and energy efficiency benefits over general-purpose cores [1–4, 7]. However, current design paradigms fix hardware connectivity at chip fabrication time, limiting SoCs' flexibility.

Within SoCs, numerous approaches have been developed to construct domain-specific chips. One notable example is streaming protocols, such as AXI-Stream, which establish fixed topologies for interconnecting accelerators. Most SoC topologies use point-to-point communication channels, enabling opportunities for pipeline parallelism. Some further support limited forms of splitting and merging. Although these facilitate accelerator-level parallelism, they are inherently constrained by the fixed topology established at design time.

As software stacks continue to grow in complexity, SoCs' design-time topology limitations make application remapping challenging, thus restricting developers. Even the state-of-the-art software-oriented data movement mechanism for SoCs, Cohort [11], supports only a limited point-to-point communication pattern. This limits the usage of accelerators in more complex workloads where (a) **more than one accelerator is needed**; (b) **more than 1 producer/consumer coexist**; and (c) **complex communication patterns that combine (a) and (b) are used**. This paper discusses the infrastructure requirements and hardware support needed to enable flexible communication patterns in heterogeneous SoCs beyond Cohort. We present our current solutions and list open questions that we find important to consider when building the next generation of software and hardware stacks.

Use case with Dependency Graph. To better show the limitations of traditional point-to-point topologies, Figure 1 graphically presents the analysis of a common server application (Cholesky [9]) that could be desirable for acceleration. Figure 1a displays the data dependency graph for a part of the application with four dependent functions coloured differently. This information is taken into account when it is parallelised through software runtimes such as OpenMP. Figure 1b shows that many inter-function communication patterns coexist in the same application. Cholesky extensively uses Multiple-Producer-Single-Consumer (MPSC), marked in green, often called a reduction or merge pattern. Nevertheless, all communication patterns are present, making it clear that **we need to support more than typical point-to-point communications** if we want to accelerate such workloads.



(a) Original kernel decomposition. **(b)** Communication Patterns.

Figure 1. Cholesky Communication Pattern Analysis. (Legend: s-single, m-multiple, c-consumer, p-producer)

2 Base Infrastructure Requirements

For applications to interact with accelerators, configuration and data transfer are often done via special loads and stores to memory-mapped I/O (MMIO), Direct Memory Access (DMA) engines, or a hybrid approach leveraging shared memory. Recent proposals focus on accelerator programmability [5, 6, 11], and standing out from these is Cohort [11], with its *Software Oriented Acceleration, SOA* model. Cohort provides an accelerator integration layer enabling efficient hardware-software communication through standard SPSC FIFO queues derived from existing software. These queues are high-performance and offer a straightforward interface for interaction, making them a compelling choice. However, to accelerate applications with numerous accelerators within a heterogeneous system, we identify two essential extra requirements: a) **support for multiple input/output queues connected to the same accelerator**, and b) **support for more complex queue types such as SPMC, MPSC, and MPMC**. We must incorporate these key features to overcome the limitations of existing frameworks.

3 New Hardware Support

Cohort has been an inspiration for interfacing accelerators, offering a simple but elegant solution based on one of the most common data structures in software: queues. We propose leveraging Cohort's infrastructure while adding the necessary support for our requirements. By incorporating a generic MPMC queue and enabling multiple input/output queues, we can support all relevant communication patterns.

However, MPMC queues are known to be slower than SPSC due to the need for atomic synchronisation. This increases the minimum latency required for an accelerator to have an efficient

computation-to-communication ratio. We opt for a block-based, lock-free bounded queue: BBQ [10] that uses MAX operations instead of Compare-And-Swap (CAS), and leverage MAPLE [8] to initiate the necessary atomic operations. Further, we have also identified the need for multiple input/output software queues connected to a single accelerator and enabled this capability in the system.

4 Discussion

Our work proposes hardware support for diverse communication patterns between accelerators and/or software threads to overcome the mentioned limitations. Figure 2 shows the different patterns (a,b,c,d); how an application is mapped onto an SoC, where accelerators and CPUs coexist, enabling seamless data interleaving between software threads and accelerators. We envision that future SoCs will continue to incorporate more accelerators and may even offer eFPGAs in some tiles to provide greater flexibility. With this perspective, we now discuss the open challenges.

Does the order in the queues matter? In the single point-to-point communication pattern, a *SPSC FIFO* can be seen as a synchronisation point where data is serialised, ensuring the maintained order. However, when using any of *SPMC/MPSC/MPMC*, the FIFO order is no longer guaranteed since multiple consumers and producers are writing to the same queue, typically relying on synchronisation mechanisms, which does not always grant the same access pattern to consumers/producers. For example, in an Multiple-Producer, Multiple-Consumer (MPMC) scenario, the first entry in the queue may not necessarily be consumed by Consumer 1 if e.g. Consumer 2 is faster at grabbing the data.

This becomes particularly relevant when multiple accelerators execute the same kernel/function. Not all applications require strict ordering, but for ones that do, there are different methods to enforce it. One simple software-based approach can be to assign ordered IDs to data elements, although at an extra space cost.

Should we add Support for OpenMP/MPI?. The SOA proposal makes us question whether we should support some of the most common software runtimes used in distributed systems. Fundamentally, hardware-supported queues simplify the runtime support by directly sending messages to accelerators without extra software interaction. In this way, we facilitate easier integration with accelerators, which routinely rely on custom stacks.

Who is responsible for mapping the application to the system architecture? A particularly challenging and interesting debate is whether we should develop a compiler or OS that is able to understand the system with accelerators and map the application to the respective accelerators in an invisible way to developers. If that is the direction to move, what should be the minimum information and abstraction visible to the compiler or the OS to make the decision about autonomous mapping of the respective code to the most suitable accelerators? For example, regular processes use Process Control Block (PCB), to save the information about the ISA of the target hardware in addition to other metadata. Then, should we add the list of the accelerators that the application can be mapped into this PCB block as well?

On the other hand, if we choose to re-use NVSHMEM or NCCL like collective libraries to program the domain-specific accelerators together with GPUs, how do we want to embed heterogeneous set of accelerators into their mapping algorithm instead?

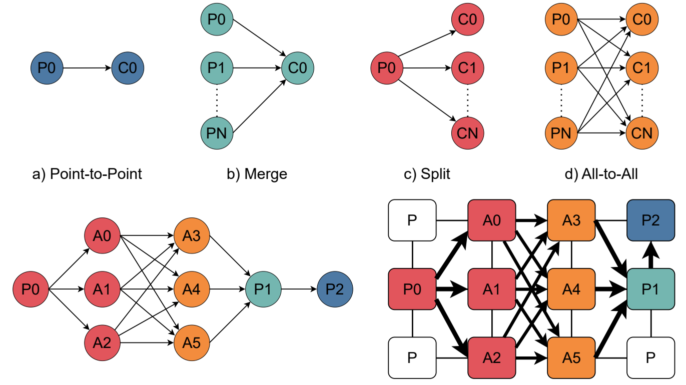


Figure 2. Application mapping in a heterogeneous system: (top) Communications patterns; (bottom, left) data-flow across different cores and accelerators; (bottom right) the communication pattern seen in the heterogeneous system in a 4x3 tiled mesh system where data in the system flows from Processors (P) to Accelerators (A).

Or should we keep programmers solely responsible for coding and mapping the program components to the respective accelerators on their own, as has been done in many state-of-the-art frameworks?

What other software and hardware features do we need to provide to accelerator designers and software developers who are using them? Apart from flexible communication patterns, runtime support, and application mapping, we must debate with the community additional features to support accelerators. Some of these features can be different methods of sharing the data structures and synchronising with respect to system events, better support for debugging and profiling for accelerators, fault detection and mitigation, and general OS support. These questions are important for selecting the right set of hardware features to truly enable accelerators to be first-class citizens of the system, without inadvertently building another CPU alongside each accelerator. Ideally, we imagine a modular solution to plug-and-play a respective hardware to enable a new software feature.

Do we need a new language for programming heterogenous SoCs? It is been always common to build new Domain Specific Languages (DSLs) for new behaviours, domain-specific applications and different hardware targets. However, since different accelerators feature different functionalities, it has been challenging to build even a common API for heterogeneous SoCs before proceeding to a common programming language that can encapsulate them all. On the other hand, it could be very elegant to have a DSL or generally a new programming language that is aware of the heterogeneity of the underlying hardware and that provides sufficient abstraction to program at the accelerator level coarse granularity. Then, the question is again about what those features should be, how new features should be ported with respect to each new accelerator, and what the guarantees should be for the compiler and the runtime of this new language. Given that the LATTE community is a unique blend of people from both computer architecture and programming languages background, we believe that this particular question can spur a very interesting discussion.

References

[1] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(2):292–308, 2019.

[2] William J. Dally, Yatish Turakhia, and Song Han. Domain-specific hardware accelerators. *Commun. ACM*, 63(7):48–57, jun 2020.

[3] Guy Eichler, Luca Piccolboni, Davide Giri, and Luca P Carloni. Mastermind: Many-accelerator soc architecture for real-time brain-computer interfaces. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*, pages 101–108. IEEE, 2021.

[4] Davide Giri, Kuan-Lin Chiu, Guy Eichler, Paolo Mantovani, and Luca P. Carloni. Accelerator integration for open-source soc design. *IEEE Micro*, 41(4):8–14, 2021.

[5] Davide Giri, Paolo Mantovani, and Luca P. Carloni. Accelerators and coherence: An soc perspective. *IEEE Micro*, 38(6):36–45, 2018.

[6] Guillem López-Paradís, Balaji Venu, Adriá Armejach, and Miquel Moretó. Characterization of a coherent hardware accelerator framework for socs. In Cristina Silvano, Christian Pilato, and Marc Reichenbach, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 91–106, Cham, 2023. Springer Nature Switzerland.

[7] Amir Morad, Tomer Y. Morad, Yavits Leonid, Ran Ginosar, and Uri Weiser. Generalized multiamdahl: Optimization of heterogeneous multi-accelerator soc. *IEEE Computer Architecture Letters*, 13(1):37–40, 2014.

[8] Marcelo Orenes-Vera, Aninda Manocha, Jonathan Balkind, Fei Gao, Juan L. Aragón, David Wentzlaff, and Margaret Martonosi. Tiny but mighty: designing and realizing scalable latency tolerance for manycore socs. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 817–830, New York, NY, USA, 2022. Association for Computing Machinery.

[9] Isaac Sánchez Barrera, Miquel Moretó, Eduard Ayguadé, Jesús Labarta, Mateo Valero, and Marc Casas. Reducing data movement on large shared memory systems by exploiting computation dependencies. In *Proceedings of the 2018 International Conference on Supercomputing*, ICS '18, page 207–217, New York, NY, USA, 2018. Association for Computing Machinery.

[10] Jiawei Wang, Diogo Behrens, Ming Fu, Lilith Oberhauser, Jonas Oberhauser, Jitang Lei, Geng Chen, Hermann Härtig, and Haibo Chen. BBQ: A block-based bounded queue for exchanging data and profiling. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 249–262, Carlsbad, CA, July 2022. USENIX Association.

[11] Tianrui Wei, Nazerke Turtayeva, Marcelo Orenes-Vera, Omkar Lonkar, and Jonathan Balkind. Cohort: Software-oriented acceleration for heterogeneous socs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 105–117, New York, NY, USA, 2023. Association for Computing Machinery.