Generality is the Key Dimension in Accelerator Design

Jian Weng Vidushi Dadu Sihao Liu Tony Nowatzki
{jian.weng,vidushi.dadu,sihao,tjn}@cs.ucla.edu
University of California, Los Angeles

ABSTRACT

Automating the hardware and software stack design of domain-specific accelerators can enable a much broader applicability of efficient accelerator architectures.

We take the position that what distinguishes domain-specific accelerators is their degree of generality along key dimensions (eg. generality of control patterns, memory access, reuse, and parallelism). Generality is expensive in terms of hardware overhead, so accelerator designers carefully choose which dimensions to be general.

However, automated accelerator design tools (eg. high-level synthesis) typically focus their analysis on optimizing a single program region (allocating resources, executing operations in parallel, pipelining and orchestrating data, etc.). Generality, if it is needed, is left to the programmer to reason about in an awkward way. We argue that a new approach is needed, where generality is an integral and explicit aspect of automated accelerator design.

This position raises difficult questions of how should generality be expressed in design exploration and how the hardware designer should convey the types of generality required. We discuss with possible solutions based on our experiences with the DSAGEN accelerator design framework.

1 GENERALITY DEFINES ACCELERATORS

One key challenge in automated accelerator generation is designing for *generality*. In fact, we posit that it is the degrees of generality along various dimensions that are the key distinguishing features of existing manually designed accelerators. Figure 1 overviews possible generality dimensions:

- **Inst./Datatype:** Breadth of compute units/datatypes.
- **Control:** The degree to which arbitrary forms of control flow are supported. For example, the ability to execute data-dependent control efficiently subsumes static control.
- **Memory Access:** How effective are arbitrary memory access patterns. For example, indirect access can be viewed as a generalization of simpler affine access patterns.

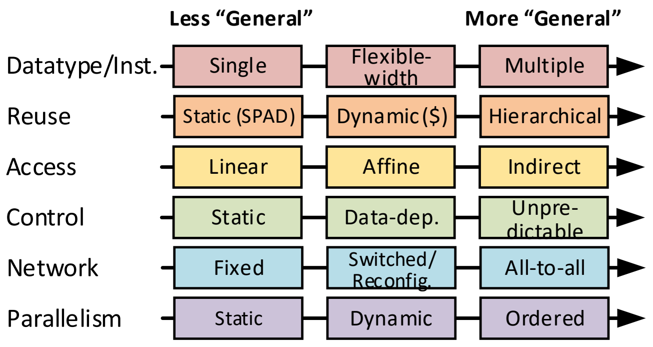

Figure 1: Dimensions of Generality in Accelerators

- **Reuse:** The degree to which dynamic data-reuse is supported. For example, this could mean the difference between the use of scratchpads and caches.
- **Network:** The flexibility in routing between hardware units. Eg. the difference between a fixed network (eg. systolic array) compared to a reconfigurable network (eg. static CGRA network or a dynamically routed Network on Chip).
- **Parallelism:** To what degree can irregular parallelism be executed efficiently. For example, the support for task parallelism with programmable scheduling policies and load balancing.

These dimensions of generality help explain the tradeoffs that accelerators make. Take the DianNao [2] accelerator for dense deep learning kernels: it provides just enough access-generality to support the different patterns required for matrix multiplication, and just enough network/instruction generality to support fused non-linear transforms. An architecture like Chronos [1] is extremely flexible in its parallelism support, but uses fixed-function PEs. Q100 [6] has very efficient support for data-dependent control flow (joins/partitions), but has no support for general memory access (only contiguous). Graphdyns [7] has general support for indirect memory, remote atomics, and flexible load balancing, but can only support synchronous parallelism and programmer-controlled scratchpads for exploiting reuse.

Further, the aspects of the design which are general are the primary source of hardware cost. The task queue in Chronos costs more resources than any processing element. The crossbar that enables indirect access in graph accelerators is a major source of area overhead (often the largest source [7]). The partitioners in Q100 cost roughly 50% area in most designs. In the era of specialization, generality has to be applied judiciously.

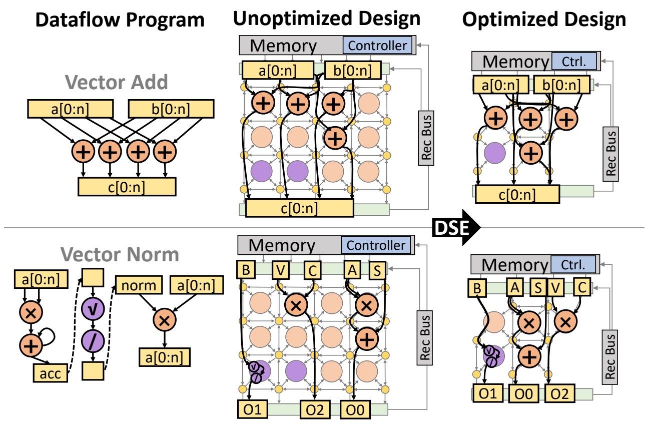

Figure 2: Maintaining Generality During DSE

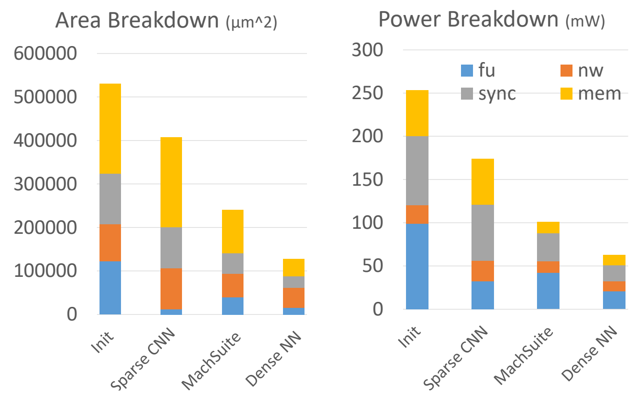

Figure 3: Results of DSE with Graph-based Representation

2 REPRESENTING GENERALITY

Because generality is so critical to the design process, we argue that it must be somehow represented and optimized for during hardware exploration and generation. Certain forms of generality are simple to express as categorical traits: ie. supports affine memory access, supports coherent caches, etc. Cross-cutting concerns are perhaps best-expressed in this simple manner, and one could imagine these being parameters of a template architecture.

However, other characteristics cannot be expressed in such broad strokes. Take the network generality as an example, specifically in the datapath of the accelerator. Generality manifests in the flexibility of routing between datapath processing elements, perhaps literally in the instantiation of routers or muxes. The effect of datapath flexibility depends on the specific connectivity of each router, so no simple parameter could describe the whole design space.

There are other reasons why a richer representation may be useful. For example, a more detailed representation would be able to better express heterogeneity (either within the core or across cores), which can mean more opportunities for specialization.

Our Approach and Experience: In our own prior work, we have explored the use of richer representations of the architecture design space using graphs. In the architecture description graph of DSAGEN [5], nodes in the graph are hardware primitives like PEs, switches, routers, and memories. Edges represent direct communication between elements. The role of the compiler is to spatially map aspects of the target applications onto this hardware.

Figure 2 shows how this graph-based program representation is useful during design space exploration (DSE). The input application is represented as a dataflow graph, and two examples are shown on the left pane. These applications are mapped to an unoptimized hardware in the middle pane; here the network and instruction generality is far too high for a low-overhead accelerator. The mapping of program to hardware can guide the DSE; a just-general-enough design, which has far less flexible routing and instructions, is shown in the right pane. Figure 3 shows the effectiveness of DSAGEN using a graph-based hardware representation; The "init" bar shows the area (or power) of a highly-general initial design, while the following three bars shows the same for DSAGEN-generated designs targeting different workload suites [2–4].

While DSAGEN's implementation does not consider all dimensions of generality described earlier (it focuses on generality within the datapath and memory patterns), we believe that a graph-based representation for such capabilities would be useful. For example, a graph representing the network-on-chip, heterogeneous cores, and cache/scratchpad hierarchy would yield a rich design space.

There are of course many challenges with such a rich design space, not the least of which is the difficulty in searching this space to meet some objective. Another notable challenge is that an irregular hardware graph with only modest generality can be a challenging target for the compiler – it is neither representable as a simple loop nest (eg. like a tensor core), nor is it flexible enough to provide arbitrary mapping (eg. like a Von Neumann architecture).

3 UNCERTAINTY DEMANDS GENERALITY

A key aspect of design automation is how the user conveys what kinds of generality are required. The approach we take in DSAGEN is that the set of input applications forms a proxy for this generality: If target programs require a variety of unique topologies to be effective, then a general network will be favored during DSE.

However, this approach leaves no room for expressing uncertainty on what types of workloads will be important for the future, causing poor over-specialized hardware choices. To explain, often a designer knows of some programs that will certainly run on the accelerator, while only knowing vague facts about other future target programs: eg. all future programs will be floating-point; no future programs will have control flow. How to convey these facts to the design space explorer is not necessarily trivial. For example, the designer may be concerned that the datapaths of existing kernels will change dramatically in future kernels.

Therefore, to avoid overfitting, we believe that uncertainty needs to be valued during exploration. While this is still an open question, one possible approach is to mutate target programs during design-space exploration, according to the distribution of uncertainty along different parameters. For example, if the datapath topology is uncertain, then perhaps the datapath mutates randomly during DSE, so that a overly-specific network becomes insufficient.

Summary: Generality is a first-order consideration for accelerator design. Generality needs to be reasoned about during accelerator generation, while also taking into account the uncertainty of behaviors in potential future programs.

REFERENCES

[1] Maleen Abeydeera and Daniel Sanchez. 2020. Chronos: Efficient Speculative Parallelism for Accelerators. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1247–1262. https://doi.org/10.1145/3373376.3378454

[2] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning. In *19th ASPLOS*. ACM, 16 pages. https://doi.org/10.1145/2541940.2541967

[3] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *44th ISCA*. 14 pages. https://doi.org/10.1145/3079856.3080254

[4] B. Reagen, R. Adolf, Y. S. Shao, G. Wei, and D. Brooks. 2014. MachSuite: Benchmarks for accelerator design and customized architectures. In *IISWC*. https://doi.org/10.1109/IISWC.2014.6983050

[5] Jian Weng, Sihao Liu, Vidushi Dadu, Zhengrong Wang, Preyas Shah, and Tony Nowatzki. 2020. DSAGEN: Synthesizing Programmable Spatial Accelerators. In *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020*. IEEE, 268–281. https://doi.org/10.1109/ISCA45697.2020.00032

[6] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. 2014. Q100: The Architecture and Design of a Database Processing Unit. In *19th ASPLOS* (Salt Lake City, Utah, USA). 14 pages. https://doi.org/10.1145/2541940.2541961

[7] Mingyu Yan, Xing Hu, Shuangchen Li, Abanti Basak, Han Li, Xin Ma, Itir Akgun, Yujing Feng, Peng Gu, Lei Deng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. 2019. Alleviating Irregularity in Graph Analytics Acceleration: a Hardware/Software Co-Design Approach. In *52nd MICRO*. https://doi.org/10.1145/3352460.3358318