DOI:10.1145/3361682

DSAs gain efficiency from specialization and performance from parallelism.

BY WILLIAM J. DALLY, YATISH TURAKHIA, AND SONG HAN

Domain-Specific Hardware Accelerators

FROM THE SIMPLE embedded processor in your washing machine to powerful processors in data center servers, most computing today takes place on general-purpose programmable processors or CPUs. CPUs are attractive because they are easy to program and because large code basesexist for them. The programmability of CPUs stems from their execution of sequences of simple instructions, such as ADD or BRANCH; however, the energy required to fetch and interpret an instruction is 10× to 4000× more than that required to perform a simple operation such as ADD. This high overhead was acceptable when processor performance and efficiency were scaling according to Moore's Law.[32] One could simply wait and an existing application would run faster and more efficiently. Our economy has become dependent on these increases in computing performance and efficiency to enable new features and new applications. Today, Moore's Law has largely ended,[12] and we must look to alternative architectures with lower overhead, such as domain-specific accelerators, to continue scaling of performance and efficiency. There are several ways to realize domain-specific accelerators as discussed in the sidebar on accelerator options.

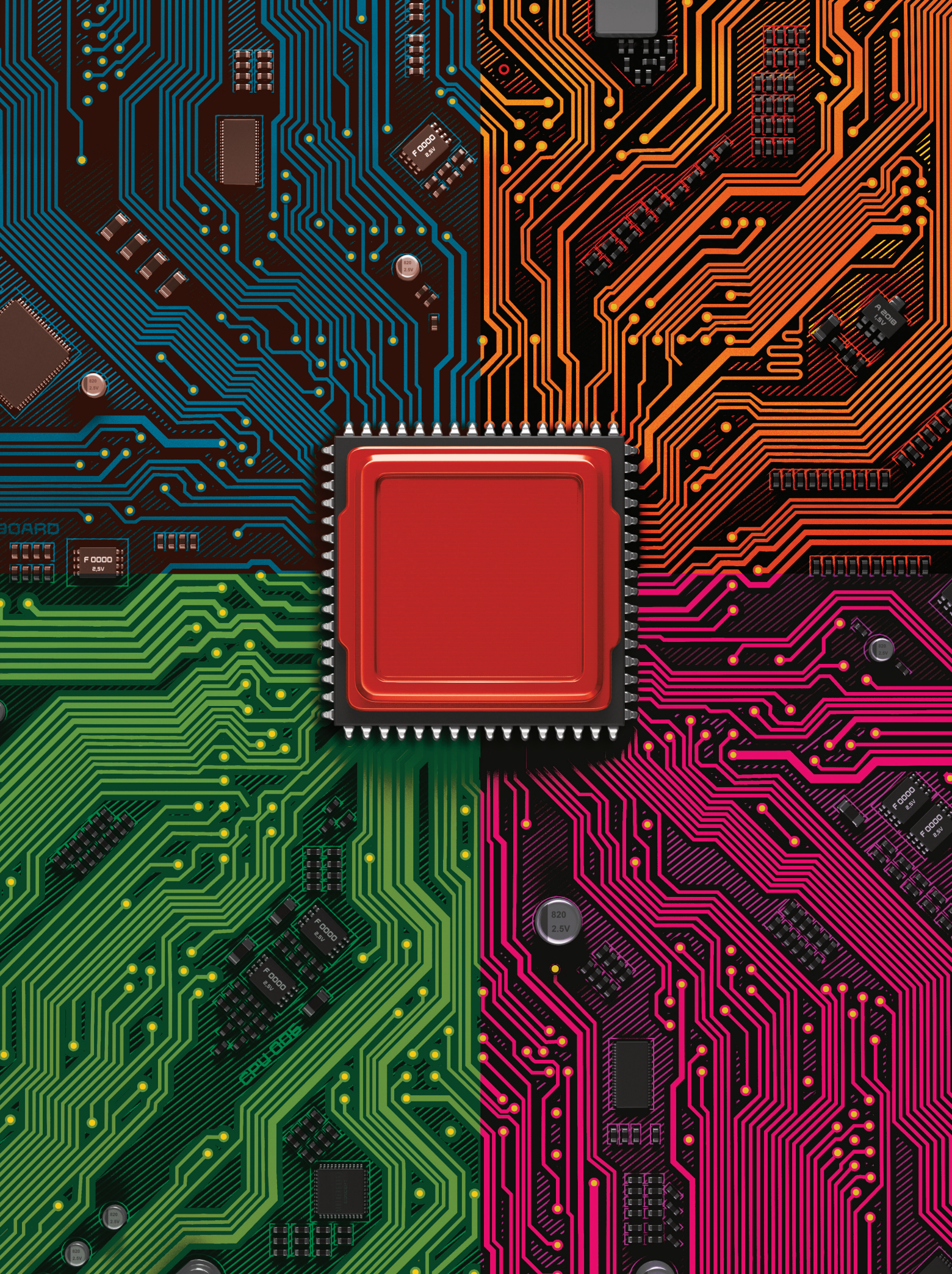
A domain-specific accelerator is a hardware computing engine that is specialized for a particular domain of applications. Accelerators have been designed for graphics,[26] deep learning,[16] simulation,[2] bioinformatics,[49] image processing,[38] and many other tasks. Accelerators can offer orders of magnitude improvements in performance/cost and performance/W compared to general-purpose computers. For example, our bioinformatics accelerator, Darwin,[49] is up to 15,000× faster than a CPU at reference-based, long-read assembly. The performance and efficiency of accelerators is due to a combination of specialized operations, parallelism, efficient memory systems, and reduction of overhead. Domain-specific accelerators[7] are becoming more pervasive and more visible, because they are one of the few remaining ways to continue to improve performance and efficiency now that Moore's Law has ended.[22]

Most applications require modifications to achieve high speed up on

» key insights

- **Most speedup comes from parallelism enabled by specialization—the main source of efficiency.**
- **The underlying algorithms often have to change—trading increased hardware-friendly computation for reduced memory bandwidth demands.**
- **Accelerator design is really parallel programming guided by a cost model—arithmetic is free and global memory is expensive.**
- **Memory typically dominates both area and power of domain-specific accelerators.**
- **Specialized instructions give much of the advantage of a DSA at a fraction of the development cost and while retaining programmability.**
- **Domain-specific accelerators are one of the few ways to continue scaling the performance and efficiency of computing hardware.**

ILLUSTRATION BY MATT HERRING, USING IMAGERY BY BET_NOIRE/GETTY IMAGES


F 0000
2,5V
820
2.5V
BOARD

domain-specific accelerators. These applications are highly tuned to balance the performance of conventional processors with their memory systems. When specialization reduces the cost of processing to near zero, they become memory limited. The application must be reworked, codesigning the application with the accelerator, to reduce memory bandwidth and memory footprint. Even after rework, many domain-specific accelerators remain memory dominated.

A well-designed accelerator covers the broadest possible space of applications—accelerating a domain rather than a sing le application. Adding domain-specific instructions to a programmable processor provides the efficiency of the specialized instruction while retaining flexibility. Complex instructions give better efficiency because they amortize the high overhead of programmability. Building a parallel computer from domain-specific processing elements can also accelerate a large domain of applications with only a small loss of efficiency.

The design of a domain-specific accelerator is really a form of parallel programming, but with a cost model very different from what most programmers use. Arithmetic and logical operations are nearly free, and memory accesses have a cost that is a function of the size of the memory being accessed. Most of the effort in designing an accelerator is refactoring the application to optimize efficiency under this model. We envision future programming systems where the programmer specifies the algorithm and a mapping to hardware in space and time. From this description, the detailed design of the accelerator would be largely automated. Such tools will facilitate the rapid exploration of the accelerator design space and eliminate many of today's obstacles to accelerator design.

The remainder of this article describes the current state of the art in domain-specific accelerators. We start by discussing the four techniques accelerators employ to achieve performance and efficiency: specialization, parallelism, local and optimized memory, and reduced overhead. We then explore the process of codesigning applications and accelerators and we discuss how most accelerators are memory dominated. The challenge of balancing specialization with generality is examined, and later we describe how accelerator design can be viewed as designing parallel programs with a set of costs reflecting modern hardware.

Sources of Acceleration

Domain-specific accelerators exploit four main techniques for performance and efficiency gains:

Data specialization: Specialized operations on domain-specific data types can do in one cycle what may take tens of cycles on a conventional computer. Specialized logic to perform an inner-loop function gains in both performance and efficiency.

Parallelism: High degrees of parallelism, often exploited at several levels, provide gains in performance. To be effective, the parallel units must exploit locality and make very few global memory references or their performance will be memory bound.

Local and optimized memory: By storing key data structures in many small, local memories, very high memory bandwidth can be achieved with low cost and energy. Access patterns to global memory are optimized to achieve the greatest possible memory bandwidth. Key data structures may be compressed to multiply bandwidth. Memory accesses are load-balanced across memory channels and carefully scheduled to maximize memory utilization.

Reduced overhead: Specializing hardware eliminates or reduces the overhead of program interpretation.

The speedup gains from specialization and parallelism are multiplicative. The dynamic programming engine described here, for example, gets a 37× speedup from specialization and an additional 4034× speedup from parallelism for a net 150,000× speedup compared to a conventional processor. Some of these factors are also dependent. Achieving high degrees of parallelism, for example, depends on locality. The 4096 processing elements in the dynamic programming engine only reference small local *traceback* memories. This degree of parallelism would not be possible if global memory references were required. Optimizing memory may also rely on specialization. Compressing data structures may only make sense if specialized logic is available to do the compression.

Data specialization. The defining feature of many domain-specific accelerators is a set of hardware operations specialized to the application domain. The inner loops of many demanding applications perform tens to hundreds of arithmetic and logical operations with only very local memory references. In many cases, specialized logic can perform the entire inner loop in a single cycle with a small amount of area and power. This logic is fed by specialized registers and communication links that provide and consume data with very low energy. As an example, consider the Smith-Waterman algorithm[44] with affine gap penalties.[14] This algorithm is widely used in genome analysis to align two gene sequences. Each iteration of the inner loop computes the following recurrence equations:

$$I(i,j) = \max\{H(i,j-1) - o, I(i,j-1) - e\} \quad (1)$$

$$D(i,j) = max\{H(i-1,j) - o, D(i-1,j) - e\} \quad (2)$$

$$H(i,j) = max\begin{cases} 0 \\ I(i,j) \\ D(i,j) \\ H(i-1,j-1) + W(r_i, q_j) \end{cases} \quad (3)$$

Here $H(i, j)$ is the maximum score for an alignment ending at (i,j), o and e are the penalties for opening and extending an insertion or deletion, and $W(r, q)$ is the cost of substituting base r for base q. The computation is performed in 16-bit integer arithmetic.

Performing this computation on a conventional x86 processor without SIMD vectorization takes around 35 arithmetic and logical operations and 15 load/store operations. On an Intel Xeon E5-2620 4-issue, out-of-order 14nm CPU, each iteration takes about 37 cycles and consumes 81nJ. On our 40 nm Darwin accelerator, each iteration takes a single cycle, a 37× speedup, and consumes 3.1pJ, a 26,000× reduction in energy. Of the 3.1pJ, only 0.3pJ is consumed computing the recurrence equations. The balance of 2.8pJ is used for a single

memory access to store a 4-bit "traceback pointer" that identifies which preceding cell was used to compute the value.

A large fraction of the area and energy savings of specialization are due to elimination of overhead. Much of the 81nJ consumed by the x86 processor and much of its area are spent fetching, decoding, and reordering instructions. This overhead is largely eliminated by specialization. The processing element that computes the recurrence equations takes only 0.004mm^2 of die area in a 40nm process. Despite being three technology nodes behind the 14nm CPU, the specialized operations of the accelerator offer orders of magnitude improvement in performance, power, and area.

Specialization also enhances locality by reducing the cost of memory compression. In our EIE accelerator for sparse neural networks,[16] we store 10%–30% dense networks in compressed-sparse-column (CSC) format. We further compress the row pointers to 4-bits each using run-length coding and compress the network weights using a 16-entry codebook. The compressed-sparse representation of network weights results in a 30× reduction in size allowing the weights of most networks to fit into efficient, local, on-chip memories, which takes two orders of magnitude less energy to access than off-chip memories.

On a conventional processor, the extra operations required to walk the pointers of the sparse-matrix data structure make such representations inefficient for densities above 1%. Similarly, the overhead of the run-length and codebook compression would be prohibitive on a general-purpose processor. With specialized logic, the pointer walking is done in a dedicated pipeline stage, with the pointers fetched from a dedicated, local memory. The decompression, both for the run-length pointer encoding and the codebook lookup, is also done in a dedicated pipeline stage. The area needed to support sparsity and compression with specialized logic is relatively small: the 16-entry weight decoder takes less than 1% of the die area; the pointer RAM takes about 20% of the area and power. On a general-purpose processor, the overhead is prohibitive.

Parallelism. Most domain-specific accelerators exploit parallelism at one or more levels. By specializing the parallelism to the application domain, the synchronization and communication between processing elements are greatly simplified. Only the communication and synchronization patterns in the application being accelerated need to be supported. By eliminating overhead, the parallel processing elements can be made very simple and very small. As an example, the alignment portion of our Darwin accelerator exploits parallelism at two levels. At the outer-loop level, $A = 64$ systolic arrays of processing elements process 64 separate alignment problems in parallel. There is no communi-

Acceleration Options

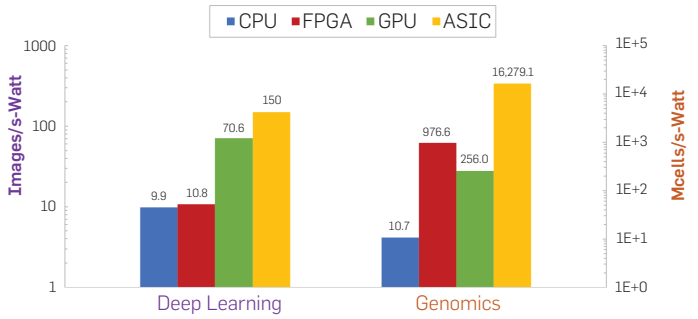
Domains of applications can be accelerated with ASICs, FPGAs, or GPUs each offering different trade-offs between development cost, programmability, and efficiency. ASICs (application-specific integrated circuits) provide the highest efficiency but have a high nonrecurring engineering (NRE) cost and poor programmability. Their logic is hardwired at design time for a single application domain. Soft logic in FPGAs lowers the efficiency for specific tasks by 10–100×[29] but enables the same chip to be dynamically configured for different applications, for example, for deep learning or genomics. Soft logic also allows for deeper specialization (for example, constant folding specific values of weights in a neural network[5]) and allows for an accelerator to be instantiated near the data it operates on, reducing communication costs.[47] GPUs are platforms that accelerate multiple domains by incorporating specialized operations (such as HMMA[4]) and memory optimizations (such as compressed surface storage[3]). For the applications they accelerate, they provide near-ASIC efficiency. For other applications, their SIMT execution model[33] offers order of magnitude better efficiency than CPUs at the expense of single-thread performance.

Figure 2 compares the efficiency of FPGAs, GPUs, and ASICs for two domains: deep learning and genomics. For domains where GPUs have specialized logic, such as deep learning, they provide near-ASIC efficiency.[a] In other domains, such as genomics, GPUs provide lower efficiency than FPGAs but offer faster development time. For genomics, we coded the banded Smith-Waterman algorithm[50] in CUDA for the GPU in one day—giving 25× improvement in efficiency over the CPU. Bringing this algorithm up on an FPGA took two months of RTL design and performance tuning—to achieve four times the efficiency of the GPU. Hardening this RTL into an ASIC gives 16× the efficiency of the FPGA but with significant nonrecurring costs and lack of flexibility. Adding a dynamic-programming step (DPS) instruction to the GPU matches the efficiency of the ASIC and with no loss of efficiency.

There are architectures that provide intermediate trade-off points for programmability and efficiency between FPGAs and ASICs. CHARM[6] uses a number of programmable domain-specific accelerator building blocks (ABBs), organized into ABB islands, to compose domain-specific acceleration. CGRAs[20] provide coarse grain reconfigurability, at word level or operator level instead of bit level, and incur lower overhead compared to FPGAs.

a The NVIDIA T4 and Habana Goya have nearly identical arithmetic performance per Watt. The difference in the figure is due to the difference in memory interface, GDDR on the T4 and LPDDR on the Goya.

Figure 2. Comparison of computation efficiency (in Tasks/s-Watt) for CPU, FPGA, GPU, and ASIC for deep learning and genomics domains.[a]



a Deep learning efficiency is measured in terms of inference images/s-Watt for Resnet-50[21] (sources: CPU,[43] FPGA,[53] GPU,[36] ASIC[15]). Genomics efficiency is measured in Mcells/s-Watt for Banded Smith-Waterman algorithm (sources: CPU,[9] FPGA,[50] GPU (our implementation on NVIDIA V100), ASIC[50]).

cation between the subproblems, and the only synchronization required is upon completion of each subproblem. A typical reference-based assembly performs billions of alignments, so there is ample outer-loop parallelism.

At the inner-loop level, each array consists of $P = 64$ processing elements that compute 64 elements of the H, I, and D matrices in parallel. The computation is performed along an antidiagonal of the matrices as originally suggested in Lipton and Lopresti.[30] On cycle t, processing element p computes the matrix elements at $(p, t - p)$. Matrices with more than P rows are processed in swaths P rows at a time. Because matrix element (i, j) depends only on the elements directly above $(i - 1, j)$, directly to the left $(i, j - 1)$, and above and to the left $(i - 1, j - 1)$, only systolic nearest neighbor communication between the processing elements is required. As with all systolic arrays, synchronization is implicit.

The parallelism exploited by special-purpose accelerators typically has very high utilization. Utilization at the outer-loop level is close to 100%. Until the very end of the computation, there is always another subproblem to process as soon as one finishes. With double buffering of the inputs and outputs, the arrays are working continuously. At the inner-loop level, utilization is 98.5%. Computation is performed in 512 × 512 tiles. At the start of each tile, only a single PE, at the upper left corner, is active. Each cycle another PE becomes active until all 64 are operating. Similarly, at the bottom right corner, the number of active PEs ramps linearly down from 64 to 0. Although it is possible to have the idle PEs start on the next alignment immediately, this is not done in Darwin. Idling of PEs at the start and end makes the average PE utilization 98.5% and the overall speedup due to parallelism 4034×. This speedup due to parallelism is multiplicative with the 37× speedup due to specialization giving an overall speedup on alignment of 150,000×.

In EIE, we parallelize a sparse matrix × sparse vector multiplication by partitioning the rows of the matrix across 256 PEs. Each nonzero input activation and its column are broadcast to all PEs. Upon receipt of each activation, each PE walks the nonzero row entries for that column in its subset of rows, accumulating row sums locally. Other than the activation broadcast, there is no communication between the PEs. A FIFO queue of pending input activations at each PE load balances work across the PEs, improving PE utilization from 50% without the FIFO to 90% with the FIFO.

Local and Optimized Memory. The gains from specialization and parallelism are dependent on keeping the computation supplied from small, local memories. Each cycle, each of the 4096 PEs in the Darwin alignment engine stores a *traceback pointer* to memory, achieving a net write bandwidth of nearly 2TBps. These pointers are used to construct the optimal alignment when the dynamic programming completes. If traceback pointers were stored to global memory, the computation would be bottlenecked by memory bandwidth. Instead, the traceback pointers are stored in 4096 small SRAMs, one associated with each PE. A conventional memory subsystem, even one with many levels of caches, is largely serial and would limit the achievable parallelism to a very small number.

In a similar manner, the filtering stage of the Darwin accelerator uses 16 dedicated SRAMs to store the bin counts, the number of seeds that match within a range of a candidate alignment. Although at most four bins are incremented each cycle, the speedup here is more than four times because the bin-count updates are random and cause interference with the sequential accesses to the seed position tables. With the bin-count updates removed from the memory stream, the sequential reads of the seed tables achieve nearly ideal memory throughput. Overall, the speedup from memory access optimization is 9×–24×—3× speedup from fewer accesses to DRAM (moving bin-count updates to SRAM) and 3×–8× speedup from the increased bandwidth by changing a random access pattern to mostly sequential. Four DRAM memory channels were added to the accelerator providing another four times the speed up from memory parallelism.

Data compression can be employed to both increase the effective size of a local memory and to increase the effective bandwidth of a memory interface. The NVDLA,[35] EIE,[16] and SCNN,[37] for example, all store the weights of a neural network as sparse data structures giving an average 3×–10× increase in the effective capacity of on-chip memories. The EIE and SCNN also run-length encode the pointers of the sparse data structure as 4-bit increments. This gives a density advantage of 4×–8× compared to storing these pointers in full 16- or 32-bit form. The weights in EIE are further compressed using a 16-entry codebook. Each weight is represented by a 4-bit codeword, giving an 8× savings compared to a 32-bit float. The savings in the number of weights and the number of bits per weight is multiplicative giving an overall compression rate of 32×–64×. Whenever weights are loaded from off-chip DRAM memory, the effective off-chip bandwidth is increased by this rate—compared to loading uncompressed data. GPUs have long stored surfaces in lossless compressed form[3] to increase effective memory bandwidth.

Overhead reduction. Overhead reduction is an important aspect of specialization. Even a simple in-order processor spends over 90% of its energy on *overhead*: instruction fetch, instruction decode, data supply, and control.[10] A modern out-of-order processor spends over 99.9% of its energy on overhead[51] adding costs for branch prediction, speculation, register renaming, and instruction scheduling. Performing a 32-b integer add takes only 63 fJ in 28nm CMOS.[24] Performing an integer add instruction on a 28nm ARM A-15 takes over 250pJ,[51] about 4000× the energy of the add itself. Special purpose engines such as Darwin and EIE completely eliminate this overhead. Moreover, most adds do not need full 32-bit precision and just the number of bits needed are added, further saving energy. There are no instructions to be fetched and hence no instructions fetch and decode energy. There is no speculation, and hence no work lost due to mis-speculation. Most data is supplied directly from dedicated registers and thus no energy is required to read from a cache or from a large, multiported register file.

The high energy and area costs of instruction and data supply overhead motivate complex instructions. The energy of a single add operation is swamped by the instruction overhead energy. A complex instruction, such as the matrix-multiply-accumulate instruction (HMMA) of the NVIDIA Volta V100,[4] on the other

hand, performs 128 floating-point operations in a single instruction and thus has an operation energy that is many times the instruction overhead. Using complex, specialized instructions, one can build efficient, specialized, programmable computer systems. We revisit the concept of complex, specialized instructions later.

Codesign is Needed

Achieving high speedups and gains in efficiency from specialized hardware usually requires modifying the underlying algorithm. Because existing algorithms are highly tuned for conventional general-purpose processors, they are rarely the optimal approach for a specialized solution. Instead, the algorithm and hardware must be *codesigned* to jointly optimize performance and efficiency while preserving or enhancing accuracy.

Many existing algorithms are tuned to balance the performance of conventional processors with their memory systems. When the cost of the processing is made nearly zero via specialization, they become completely memory dominated. To get significant speedup, such algorithms must be refactored to reduce the bandwidth demands on global memory. Although methods such as tiling[52] and compression can be used to reduce global bandwidth to some degree, often more fundamental restructuring is required.

One approach to codesign is to trade more of an operation that is inexpensive in hardware (that is, logic limited) for less of an operation that is expensive (that is, memory limited). For example, conventional applications for long-read genomic sequence alignment such as GraphMap[45] spend most of their compute time on filtering and relatively little time on alignment. This approach makes sense for a general-purpose processor where filtering is relatively cheap and alignment is expensive. It is exactly the wrong optimization for specialized hardware where alignment can be made extremely efficient (26,000× more efficient and 150,000× faster than on a general-purpose processor) but filtering is fundamentally limited by global memory bandwidth. If we were to apply hardware specialization to the existing algorithm, we would be limited to a speedup of 4–5× due to the memory bandwidth required.

To exploit this difference in costs, Darwin spends 200× less time on filtering than GraphMap. This results in a 560× increase in candidate positions to be aligned and hence 560× more work for the alignment stage. However, because alignment is accelerated by 150,000×, the net result is a speedup of more than 200×. Darwin's parameters for filtering and alignment are adjusted so that the new alignment-heavy algorithm has equal or higher sensitivity than the filtering-heavy algorithm it replaces.

Codesign may also be used to reduce memory footprint—to make the use of small local memories feasible. The conventional Smith-Waterman algorithm for long, 10^4 base-pair, reads, for example, would require a prohibitively large, 10^8 entry, store for traceback pointers. To reduce the memory footprint to more feasible 2×10^5 entries, we developed the GACT algorithm that performs the dynamic programming in overlapping tiles.[49] Tiling reduces the memory footprint to that of a single 512 × 512-entry tile. Overlapping the tiles by an amount $O = 128$ that is larger than the largest expected deviation of the optimal path from the diagonal in practice gives optimal alignments.

In other cases, codesign enables algorithms that would otherwise be inefficient on conventional hardware. For example, in Han et al.,[17,18] we showed how neural networks could be pruned to 10%–30% density and compressed by 30×. The overhead of sparse methods on conventional hardware made these algorithms uninteresting except for memory compression. Codesigning special-purpose hardware for sparse operations enables these algorithms to be used to reduce computation as well.

As another example, software for whole genome alignment, such as LASTZ,[19] uses ungapped extension to filter seed hits because gapped extension is prohibitively expensive on conventional hardware. With specialized hardware, gapped extension becomes feasible[50] giving much better sensitivity when comparing the genomes of distantly-related species—where the alignments have frequent gaps.

Memory Dominates Accelerators

The area and power of most accelerators are dominated by memory, and their performance is often memory limited. As a result, much of the codesign described earlier is developing algorithms that have a small memory footprint. Most of their memory bandwidth requirements can be satisfied by small, local memories. They require only modest bandwidth from large, global memories.

Table 1 shows the relative area and power for memory and logic in the Darwin GACT accelerator, the Darwin D-SOFT accelerator, and the EIE sparse neural network accelerator. The EIE numbers are shown for 64 processing elements (PEs). D-SOFT and EIE, which accelerate a memory-limited application (seed filtering and matrix-vector multiplication, respectively) using large local memories, use over 90% of their die area for memory. In D-SOFT, the power component is over 90% as well, because the bin update logic is relatively simple, consisting of on-chip routing and simple arithmetic operations (add and compare). Even in the EIE, where the 16-bit multiply operations are more expensive, memory still consumes more than half of the chip power. For the GACT accelerator, which performs a compute-intensive dynamic programming operation, the memory that stores the traceback pointers consumes about 80% of the die area and over 75% of the power. The simple 16-bit additions and comparisons at the core of the dynamic programming recurrence equations take very little area and power. The low area and power of simple logic and arithmetic

Table 1. Breakdown of chip area and power into logic and memory units for the GACT and D-SOFT accelerators in Darwin[49] and for the EIE accelerator[16] using TSMC 40 nm.

	Unit	Area (mm²)	(%)	Power (W)	(%)
GACT	Logic	17.6	20.5	1.04	23.6
	Memory	68.0	79.5	3.36	76.4
D-SOFT	Logic	6.2	1.8	0.41	4.4
	Memory	320.3	98.2	8.80	95.6
EIE	Logic	2.8	6.9	0.23	40.3
	Memory	38.0	93.1	0.34	59.7

make domain-specific accelerators efficient. However, it also makes them memory limited. When logic is "free," memory dominates.

Because the area and power of most accelerators are memory dominated, a reasonable first estimate of these costs can be made by considering only the memory. This allows rapid design space exploration.

Because global memory bandwidth is extremely expensive, many accelerators are designed to be global memory limited—to keep this expensive resource busy. In Darwin, for example, the four DRAM memory channels provide at most four seeds per cycle. We sized the D-SOFT filtering hardware with 16 bin-count banks so it can always keep up with the four seeds per cycle from external DRAM. Similarly, the 4K GACT PEs are provisioned so that alignment can keep ahead of the filtering stage.

Because external memory bandwidth is so critical, it should be optimized. Memory schedulers should be employed that maximize memory throughput[41] and memory contents should be compressed where possible.

Balancing Specialization and Generality

In the design of domain-specific accelerators, there is a tension between generality and efficiency. Building an engine specialized for just one application can give the highest possible efficiency. However, its range of use may be too limited to generate enough volume to recover design costs, or a new algorithm may be developed rendering the accelerator obsolete. Building a completely general-purpose computer, on the other hand, would result in poor efficiency. A happy medium lies in building an engine that accelerates a *domain* of applications where the breadth of applications is increased while retaining most of the efficiency of the completely specialized accelerator.

Special instructions vs. special engines. One approach to building accelerators for broad domains is to add specialized instructions to a general-purpose processor. A hardware block is built to accelerate the core operations for a domain of algorithms—matrix multiply for deep learning or dynamic programming for genomics—and the operations are made available as instructions on a general-purpose processor. This approach makes the core operation as efficient as a completely specialized accelerator but allows the use of the programmable general-purpose processor to adapt its use to different algorithms and applications.

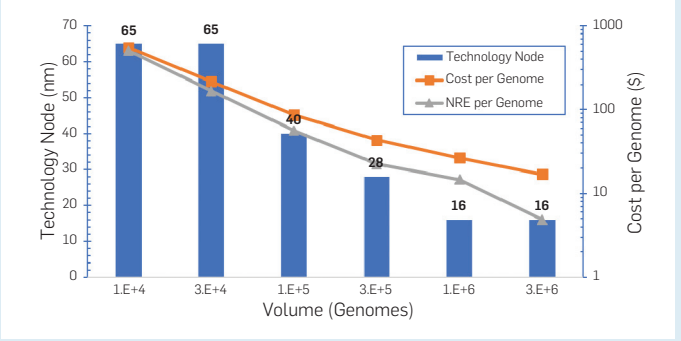
The HMMA (half-precision matrix multiply-accumulate) instruction in the NVIDIA Volta V100 GPU[4] is an example of adding a specialized instruction to a general-purpose processor. The instruction multiplies two 4×4 half-precision (16-bit) floating-point matrices accumulating the results in a 4×4 single-precision (32-bit) floating-point matrix. The Turing IMMA (integer matrix multiply accumulate) instruction performs this same operation on 8×8 8-bit integer matrices accumulating an 8×8 32-bit integer result matrix.[26] These operations accelerate the inner loops of both training and inference for convolutional, fully-connected, and recurrent layers of deep neural networks. A single HMMA instruction performs 128 floating-point operations: 64 half-precision multiplies and 64 single-precision adds. An IMMA instruction performs 1024 integer operations. This large amount of math amortizes the overhead. Using data from Horowitz,[24] we estimate that when executing an HMMA (IMMA) instruction, 77% (87%) of the energy is consumed by arithmetic. The balance of the energy is consumed by instruction overhead and fetching the data operands from the large GPU register files and shared memory. A dedicated accelerator, such as the Google TPU,[27] could be at most 23% (13%) more efficient on half-precision (8-bit integer) matrix multiply. This bound is just for the core matrix multiply operation. The accelerator may be more efficient at staging data in on-chip memories and in optimizing data movement. Also, the GPU die will be larger and hence more expensive, because it includes area for the general-purpose functions, and for other accelerators, which are unused when doing matrix multiply. This die cost factors into the recurring portion of total cost of ownership.

The advantage of implementing the accelerator as an instruction is that the full power of the general-purpose processor is available to implement other layers of the network. Pooling, normalization, batch-normalization, sparsity-mask, and nonlinear function layers are easily implemented. As new algorithms and methods are developed, they are easily incorporated as custom layers while retaining the efficiency of the accelerator for the bulk of the operations.

In a similar manner, one could implement a special-purpose dynamic programming instruction to accelerate genomics calculations. A dynamic-programming step (DPS) instruction would take as input the values of H, I, and D (Eqs. (1)–(3)) for a portion of the current diagonal and generate corresponding values for a portion of the next diagonal along with their traceback pointers. Adding such an instruction to a general-purpose GPU or CPU would provide most of the efficiency gains of a hardwired accelerator such as Darwin.

One advantage of building a specialized instruction rather than an entire engine is that only the instruction must be developed, not the entire system. Most of the complexity of a computing

Figure 1. TCO for a genomics accelerator as a function of volume. At low volumes, older technology nodes give a lower TCO because of their lower nonrecurring costs (NRE).



engine, specialized or general purpose, is in the memory system, on-chip interconnect, I/O system, and global control. When a DSA is implemented by adding an instruction to a general-purpose GPU or CPU, it can leverage the existing system components. The complexity of the domain-specific block is 100s–1000s of times smaller than the complexity of the system (as measured by lines of code). With a dedicated engine, the entire system must be developed.

Today, architects pressed to increase efficiency are turning to complex instructions, such as HMMA, IMMA, and DPS to amortize fixed instruction overhead. Complex, domain-specific instructions enable the efficiency of domain-specific accelerators to be combined with the generality of a programmable processor and with development costs a fraction of that required to develop a dedicated accelerator.

Domain-specific parallel computers. The ability to increase generality with little loss of efficiency is illustrated by comparing two simulation accelerators: the MSE[11] and MARS.[2] The MSE was a parallel computer where processing elements were highly specialized to different stages of the simulation pipeline. The MSE was 300× faster than a contemporary general-purpose computer (a VAX 11-780), but it could only accelerate switch-level simulation.

MARS used a single domain-specific programmable processing element that could serve as any of the pipeline stages. In a single cycle, each MARS PE could read a word from the input queue, perform an address calculation, read or write a word from external SRAM, extract a bit field from a word, perform an arithmetic or logical operation on the bit field, insert the resulting bit field into another word, and write a word to the output queue. The net result was a speedup of about 200× compared to a contemporary general-purpose computer (a Sun 3/260),[2] and this performance doubled over a period of years as the pipeline was refactored to eliminate bottlenecks and individual pipeline stages were tuned. MARS was nearly as fast as a hardwired engine, and because the area was dominated by memory, smaller, hardwired PEs would have made little difference to the overall area.

A key factor in the success of MARS was the low-overhead associated with horizontal microcode control. The energy overhead of programmability was largely the cost of fetching a 64-bit microinstruction from a 64-bit × 64-word microinstruction store. This energy was small compared to the access to a much larger SRAM made almost every cycle by most pipeline stages. The instruction fetch and control overhead of a conventional processor would have been prohibitive.

By defining the right domain-specific architecture, MARS was able to implement many simulation pipelines (and other tasks, such as speech recognition) with performance and efficiency approaching that of full-custom hardware. MARS proved so useful that it was reimplemented in five different generations of CMOS technology from 1.25 μm to 0.35 μm.

Total Cost of Ownership (TCO)

The technology node used to implement an accelerator should be chosen to give minimum total cost of ownership (TCO). As shown in Figure 1 for a genomics accelerator, at low volumes, the minimum TCO occurs at older (larger geometry) technology nodes. For each volume (number of de-novo, long-read genome assemblies), the bar shows the technology node that gives the minimum TCO. Nonrecurring costs are from Khazraee et al.[28] The lines show the nonrecurring engineering (NRE) cost per genome and TCO per genome for the minimum TCO technology. Darwin's 40nm technology gives minimum TCO at 10^5 genomes assembled. Paying the high nonrecurring costs for a 16nm technology is not justified until a volume of 10^6 genomes is reached. De-novo, long-read assembly of noisy reads on a CPU would cost around $1,500 per genome,[25] so a custom accelerator gives lower TCO even for a volume as low as 10^4 genomes—in a 65nm technology. At this point, the cost per genome is almost entirely NRE. Backend development costs are roughly the same for 130 nm and 65 nm ($4.3M) and dominate mask costs, so nodes older than 65 nm do not offer a material savings in NRE.

A similar TCO calculation can be used to compare the cost of a dedicated accelerator to the cost of adding specialized instructions to a CPU or GPU, or to the cost of combining several accelerators—perhaps sharing memory and I/O systems—on a single chip. Adding specialized instructions or combining accelerators gives a higher recurring cost but gives a larger volume over which to amortize the nonrecurring costs.

Accelerator Design

The design of a domain-specific accelerator is really the design of a fine-grained, memory-constrained, parallel program for a limited set of tasks. Most of the effort is in crafting an algorithm that achieves high parallelism with a small local memory footprint and low global memory bandwidth. Once a highly-parallel, highly local algorithm is developed, the design of the hardware is straightforward—and is largely dominated by memory as described previously.

The major difference between designing a DSA and writing a parallel program for a conventional parallel machine such as Summit[23] is the cost model. The cost model in turn drives differences in granularity and memory footprint. Most programmers use a cost model based loosely on the PRAM model.[40] Arithmetic functions and accesses to anywhere in a large global memory are all counted as unit-cost operations.

Even on a conventional x86 processor, the PRAM model is highly unrealistic, and on a modern GPU, even more so. Global memory operations are hundreds of times more expensive than arithmetic operations and local memory operations—those that hit in the cache. On multinode machines such as Summit, communication between nodes takes microseconds, the equivalent of thousands of operations. This leads to very coarse-grained parallelism and communication.[a] Adjusting the PRAM model for the realities of conventional parallel machines has led to models such as log P[8] that weight global accesses and communication with approximations of their actual cost.

a Many parallel machines have been built with very efficient hardware communication and synchronization.[13,34,42] Unfortunately the need to use commodity processors for the nodes of mainstream machines has prevented most programmers from benefiting from such efficient mechanisms.

Accelerator costs. A DSA has a very different cost model than a machine such as Summit. If we use energy and area as a proxy for cost, a simple model is that arithmetic is free and accessing memory has a cost dependent on the size of memory being accessed. A more accurate cost model is as follows:

Arithmetic: In 14 nm technology, arithmetic costs range from 10fJ and 4 μm^2 for an 8-bit add operation to 5pJ and 3600 μm^2 for a double-precision floating-point multiply.[24] As described earlier, these costs are usually small compared to those of memory.

Local memory: Accessing a small (8KByte) local memory in 14nm costs 50fJ/bit and SRAM memory has an area of 0.013 μm^2 per bit. The additional cost of accessing larger on-chip memories is the communication cost of getting to and from a small 8KByte subarray. This communication costs 100fJ/bit-mm, so the cost of accessing a memory of size S (in bits) is $50+0.022\sqrt{S}$ fJ. On-chip memories of up to several hundred megabytes can be realized in today's technology. A 100MB (800Mbit) memory has an access cost of about 0.7pJ/bit.

Global memory: Off-chip global memory is even more expensive. Accessing a relatively energy-efficient LPDDR4 memory costs about 4pJ/bit and higher-speed SDDR4 memory costs about 20pJ/bit.[31] Global memory is also bandwidth limited. Memory bandwidth off of an accelerator chip is limited to about 400GB/s. Placing memories on interposers can give bandwidths up to 1TB/s, but at the expense of limited capacity.

Local Communication: Communication between blocks on chip has an energy cost that increases linearly with distance at a rate of 100fJ/bit-mm.

Global communication: High-speed off-chip channels use SerDes that have an energy of about 10pJ/bit.

Logic and local memory energies scale linearly with technology—as the capacitance of the devices scales down while supply voltage is held constant.[b] Communication energy remains roughly constant. This nonuniform scaling makes communication—such as nonlocal memory access—even more critical in future systems.

Programming accelerators. Each DSA requires firmware and a software development kit (SDK) to facilitate programming. Darwin-WGA,[50] for example, uses the OpenCL programming framework,[46] which provides a software API (in C/C++) for the two kernels it accelerates in hardware, Banded Smith-Waterman and GACT-X, along with a memory model API for exchanging data between the host and accelerator global memory. The application is then written in C++ with calls to this API. The API allows Darwin-WGA to be repurposed for different genomic applications, such as reference-guided assembly, *de novo* assembly, and cross-species whole genome alignments. Accelerators that support a more flexible domain-specific language (DSL), such as Halide,[39] or a broad software library, such as Tensorflow,[1] require adding a back-end to the domain-specific compiler to map the compiler IR to the accelerator. Back-end optimizations, particularly those that minimize off-chip data transfers, significantly impact accelerator performance.

Creating accelerators with programs. Although the accelerators we have built to date have been designed by directly writing Verilog RTL,[48] we envision a future in which an accelerator is designed by writing a parallel program describing the function of the accelerator along with mapping directives that specify how the computation and state of the program is mapped to hardware in space and time.

For example, for our dynamic programming accelerator, the program is largely Eqs. (1) through (3), along with a write to a traceback memory. We describe all possible parallelism and rely on dependence analysis to serialize the computation as required:

Algorithm 1: GACT

tb ← **GACT(r, q)**

```
input  : r[TS], q[TS]
output : tb[TS,TS]
for i = 0..TS-1 do
  for j = 0..TS-1 do
    in (i,j) ← Max (h (i,j-1) - O, in
      (i, j-1) - E)
    del (i,j) ← Max (h(i-1,j) - O, del
      (i-i,j) -E)
    h (i,j) ← Max (0, in(i,j), del (i,j),
      h (i-1, j-1) + W (r[i],q[j]) )
    tb [i,j] ← ComputeTb (h (i,j),
      in (i,j), del (i,j) )
  end
end
```

In this pseudocode, the curved brackets (for example , `in(i,j)`) specify abstract indices. The square brackets indicate memory (for example, `tb[i,j]`). The recurrence matrices, `in, del`, and `h` are never fully materialized. Only the diagonal of indices needed for the current computation is held in storage at any given time. The input strings `r` and `q` are vectors of size `TS` (tile size) and the resulting traceback array `tb` is an array of size `TS × TS`.

To map this computation to a processor array, we first declare the array and then specify the mapping. A straightforward mapping is described here. We declare an array of `AS` (array size) processing elements and an array of `AS` memory arrays each of the size `STRIPES×TS`. We then map `h(i,j)` to processing elements by row i and specify the time t each element is computed according to the diagonal wavefront. The `in` and `del` matrices (not shown) are mapped identically. The traceback matrix is mapped across the traceback memories by row.

Algorithm 2: Mapping

```
STRIPES ← TS / AS
processor_array p (AS)
memory_array tbm (AS)[STRIPES, TS ]
Map h (i,j) → p (i % AS)
    at t = (i % AS) · TS + j - i / AS
Map tb [i,j] → tbm (i % AS) [i / AS, j]
```

We expect that having a programming system for accelerators of this type will facilitate the rapid exploration of alternative algorithms and mappings. A compilation tool can quickly determine the execution time and energy associated with a particular mapping. Once an efficient algorithm and mapping are settled on, the tool can generate the

b For recent technology nodes, scaling linear dimensions by 0.7× has given only a 0.8–0.9× reduction in logic energy due to the complexity and overhead of current multi-patterned design rules.

RTL. More advanced tools could automate the generation of the mapping given constraints on time and space.

Conclusion

With the end of Moore's Law, domain-specific accelerators (DSAs) remain one of the few paths to continuing to increase the performance and efficiency of computing hardware. This paper has explored how DSAs achieve performance and efficiency drawing on the authors' designs of DSAs for genomics, deep learning, simulation, and graphics spanning four decades. DSAs gain much of their efficiency from specialization and elimination of overhead. This efficiency, in turn, enables parallelism, which accounts for much of the performance of DSAs. Most accelerators are memory dominated with much of their die area and power dissipation dominated by local memories. To benefit from specialization, many existing applications must be refactored to reduce their bandwidth demands on global memory.

A successful DSA accelerates a broad domain of applications. It may achieve such flexibility by adding specialized instructions to a programmable processor such as a GPU or CPU. Breadth can also be achieved by building a domain-specific parallel computer where domain-specific programmable processing elements carry out the processing of each pipeline stage in place of specialized logic. Such processing elements must be very lean to avoid losing much of the advantage of specialization to overhead.

Although DSAs today are designed at the RTL level, we envision a future where DSAs are designed by writing a parallel program and specifying the mapping of this program to hardware resources in time and space. Most of the intellectual effort in designing a DSA is a programming task: developing algorithms that give good performance and efficiency with the DSA cost model. Lowering this program to detailed hardware can be largely automated.

In the future, we expect many programmers will become designers of DSAs. An ecosystem will emerge to support these programmers with better tools to describe and evaluate their programs. Ultimately, we expect that computer science curricula will evolve to teach algorithms and complexity with a cost model that more accurately reflects the reality of modern computing hardware. C

References

1. Abadi, M., et al. Tensorflow: A system for large-scale machine learning. In *OSDI* (2016), 265–283.
2. Agrawal, P., Dally, W.J. A hardware logic simulation system. *IEEE TCAD 9*, 1 (1990), 19–29.
3. Beers, A.C., Agrawala, M., Chaddha, N. Rendering from compressed textures. *ACM Trans. Graph. (SIGGRAPH)* 96 (1996), 373–378.
4. Choquette, J., Giroux, O., Foley, D. Volta: Performance and programmability. *IEEE Micro 38*, 2 (2018), 42–52.
5. Chung, E., Fowers, J., et al. Serving DNNs in real time at datacenter scale with project brainwave. *IEEE Micro 38*, 2 (2018), 8–20.
6. Cong, J., et al. Charm: A composable heterogeneous accelerator-rich microprocessor. In *ISLPED* (2012). ACM, 379–384.
7. Cong, J., Sarkar, V., Reinman, G., Bui, A. Customizable domain-specific computing. *IEEE Des. Test Comput. 28*, 2 (2010), 6–15.
8. Culler, D., Karp, R., et al. LogP: Towards a realistic model of parallel computation. In *ACM.Sigplan Notices, Vol. 28* (1993). ACM, 1–12.
9. Daily, J. Parasail: SIMD C library for global, semi-global, and local pairwise sequence alignments. *BMC Bioinform. 17*, 1 (2016), 81.
10. Dally, W.J., Balfour, J., Black-Shaffer, D., Chen, J., Harting, R.C., Parikh, V., Park, J., Sheffield, D. Efficient embedded computing. *Computer 41*, 7 (2008), 27–32.
11. Dally, W.J., Bryant, R.E. A hardware architecture for switch-level simulation. *IEEE TCAD 4*, 3 (1985), 239–250.
12. Esmaeilzadeh, H., Blem, E., Amant, R.S., et al. Dark silicon and the end of multicore scaling. In *ISCA* (2011). IEEE, 365–376.
13. Fillo, M., Keckler, S.W., Dally, W.J., Carter, N.P., Chang, A., Gurevich, Y., Lee, W.S. The M-machine multicomputer. *Int. J. Parallel Program. 25*, 3 (1997), 183–212.
14. Gotoh, O. An improved algorithm for matching biological sequences. *J. Mol. Biol. 162*, 3 (1982), 705–708.
15. Habana Labs. Goya Inference Platform White Paper v1.7, 2019. https://tinyurl.com/yxlcfx54
16. Han, S., Liu, X., Mao, H., Pu, J., Pedram, A., Horowitz, M.A., Dally, W.J. EIE: Efficient inference engine on compressed deep neural network. In *ISCA* (2016). IEEE, 243–254.
17. Han, S., Mao, H., Dally, W.J. Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. In *ICLR* (2016).
18. Han, S., Pool, J., Tran, J., Dally, W. Learning both weights and connections for efficient neural network. In *NIPS* (2015), 1135–1143.
19. Harris, R.S. Improved pairwise alignment of genomic DNA. PhD thesis, The Pennsylvania State University (2007).
20. Hartenstein, R. Coarse grain re-configurable architecture (Embedded Tutorial). In *ASPDAC* (2001), ACM, 564–570.
21. He, K., Zhang, X., Ren, S., Sun, J. Deep residual learning for image recognition. In *CVPR* (2016), 770–778.
22. Hennessy, J.L., Patterson, D.A. A new golden age for computer architecture. *Commun. ACM 62*, 2 (2019), 48–60.
23. Hines, J. Stepping up to summit. *Comput. Sci. Eng. 20*, 2 (2018), 78–82.
24. Horowitz, M. Computing's energy problem (and what we can do about it). In *ISSCC* (2014), IEEE, 10–14.
25. Jain, M., Koren, S., Miga, K.H., Quick, J., Rand, A.C., Sasani, T.A., Tyson, J.R., Beggs, A.D., Dilthey, A.T., Fiddes, I.T., et al. Nanopore sequencing and assembly of a human genome with ultra-long reads. *Nat. Biotechnol. 36*, 4 (2018), 338.
26. Jia, Z., Maggioni, M., Smith, J., Scarpazza, D.P. Dissecting the NVIDIA turing T4 GPU via microbenchmarking. arXiv:1903.07486 (2019).
27. Jouppi, N.P., Young, C., Patil, N., Patterson, D. A domain-specific architecture for deep neural networks. *Commun. ACM 61*, 9 (2018), 50–59.
28. Khazraee, M., et al. Moonwalk: NRE optimization in ASIC clouds. In *Computer Architecture News*, Vol. 45 (2017), ACM, 511–526.
29. Kuon, I., Rose, J. Measuring the gap between FPGAs and ASICs. *IEEE TCAD 26*, 2 (2007), 203–215.
30. Lipton, R.J., Lopresti, D.P. Comparing Long Strings on a Short Systolic Array. Princeton University, Department of Computer Science, 1986.
31. MICRON. System power calculators, 2019. https://tinyurl.com/y5cvl857
32. Moore, G.E., et al. Cramming more components onto integrated circuits. 1965
33. Nickolls, J., Dally, W.J. The GPU computing era. *IEEE Micro 30*, 2 (2010), 56–69.
34. Noakes, M.D., Wallach, D.A., Dally, W.J. The J-machine multicomputer: An architectural evaluation. *Comput. Arch. News 21*, 2 (1993), 224–235.
35. NVIDIA. NVIDIA deep learning accelerator (NVDLA), 2017. http://nvdla.org
36. NVIDIA. NVIDIA Tesla deep learning product performance, 2019. https://tinyurl.com/yyu9amxh
37. Parashar, A., et al. SCNN: An accelerator for compressed-sparse convolutional neural networks. In *ISCA* (2017), IEEE, 27–40.
38. Qadeer, W., et al. Convolution engine: Balancing efficiency & flexibility in specialized computing. In *Computer Architecture News, Vol. 41* (2013). ACM, 24–35.
39. Ragan-Kelley, J., et al. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *ACM Sigplan Notices, Vol. 48* (2013). ACM, 519–530.
40. Karpand, R.M., Ramachandran, V., Karpand, V., Karp, R.M. A survey of parallel algorithms for shared-memory machines. In *Handbook of Theoretical Computer Science*. North-Holland, 1988
41. Rixner, S., Dally, W.J., Kapasi, U.J., Mattson, P., Owens, J.D. Memory access scheduling. In *Computer Architecture News, Vol. 28* (2000). ACMe, 128–138.
42. Scott, S.L. Synchronization and communication in the T3E multiprocessor. In *ACM SIGPLAN Notices, Vol. 31* (1996). ACM, 26–36.
43. Shen, H., et al. Intel CPU outperforms NVIDIA GPU on ResNet-50 deep learning inference, 2019. https://tinyurl.com/y6xewz8r
44. Smith, T.F., Waterman, M.S. Identification of common molecular subsequences. *J. Mol. Biol.* 147, 1 (1981), 195–197.
45. Sović, I., Šikić, M., Wilm, A., Fenlon, S.N., Chen, S., Nagarajan, N. Fast and sensitive mapping of nanopore sequencing reads with GraphMap. *Nat. Commun. 7*, (2016).
46. Stone, J.E., et al. OpenCL: A parallel programming standard for heterogeneous computing systems. *Comput. Sci. Eng. 12*, 3 (2010), 66.
47. Tan, T., Nurvitadhi, E., Chiou, D. Dark wires and the opportunities for reconfigurable logic. *IEEE Comput. Architect. Lett.* (2019).
48. Thomas, D., Moorby, P. The Verilog® Hardware Description Language. Springer Science & Business Media, 2008.
49. Turakhia, Y., Bejerano, G., Dally, W.J. Darwin: A genomics co-processor provides up to 15,000× acceleration on long read assembly. In *ASPLOS* (2018). ACM, 199–213.
50. Turakhia, Y., Goenka, S.D., Bejerano, G., Dally, W.J. Darwin-WGA: A co-processor provides increased sensitivity in whole genome alignments with high speedup. In *HPCA* (2019). IEEE, 359–372.
51. Vasilakis, E. An instruction level energy characterization of arm processors. Tech. Rep. FORTH-ICS/TR-450. Foundation of Research and Technology Hellas, Institute of Computer Science, 2015.
52. Wolfe, M. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing* (1987), Society for Industrial and Applied Mathematics, 357–361.
53. Xilinx. Xilinx Imagenet Benchmarks, 2019. https://tinyurl.com/y5l4ajff

William J. Dally (bdally@nvidia.com), NVIDIA, Stanford University, CA, USA.

Yatish Turakhia (yturakhi@ucsc.edu), University of California, Santa Cruz, CA, USA.

Song Han (songhan@mit.edu), Massachusetts Institute of Technology, Cambridge, MA, USA.

Watch the authors discuss this work in this exclusive *Communications* video. https://cacm.acm.org/videos/domain-specific-accelerators