Aladdin: A Pre-RTL, Power-Performance Accelerator Simulator Enabling Large Design Space Exploration of Customized Architectures

Yakun Sophia Shao Brandon Reagen Gu-Yeon Wei David Brooks
Harvard University
{shao,reagen,guyeon,dbrooks}@eecs.harvard.edu

Abstract

Hardware specialization, in the form of accelerators that provide custom datapath and control for specific algorithms and applications, promises impressive performance and energy advantages compared to traditional architectures. Current research in accelerator analysis relies on RTL-based synthesis flows to produce accurate timing, power, and area estimates. Such techniques not only require significant effort and expertise but are also slow and tedious to use, making large design space exploration infeasible. To overcome this problem, we present Aladdin, a pre-RTL, power-performance accelerator modeling framework and demonstrate its application to system-on-chip (SoC) simulation. Aladdin estimates performance, power, and area of accelerators within 0.9%, 4.9%, and 6.6% with respect to RTL implementations. Integrated with architecture-level core and memory hierarchy simulators, Aladdin provides researchers an approach to model the power and performance of accelerators in an SoC environment.

1. Introduction

As we near the end of Dennard scaling, traditional performance and power scaling benefits based on technology improvements no longer exist. At the same time, transistor density improvements continue; the result is the dark silicon problem in which chips now have more transistors than a system can fully power at any point in time [18, 52]. To overcome these challenges, hardware acceleration in the form of datapath and control circuitry customized to particular algorithms or applications has surfaced as a promising approach, as it delivers orders of magnitude performance and energy benefits compared to general purpose solutions. Customized architectures composed of CPUs, GPUs, and accelerators are already seen in mobile systems and are beginning to emerge in servers and desktops.

The natural evolution of this trend will lead to a growing volume and diversity of customized accelerators in future systems (Figure 1), where a comprehensive assessment of potential benefits and trade-offs across the entire system will be critical for system designers. However, current customized architectures only contain a handful of accelerators, as large design space exploration is currently infeasible due to the lack of a fast simulation infrastructure for accelerator-centric systems.

978-1-4799-4394-4/14/$31.00 © 2014 IEEE

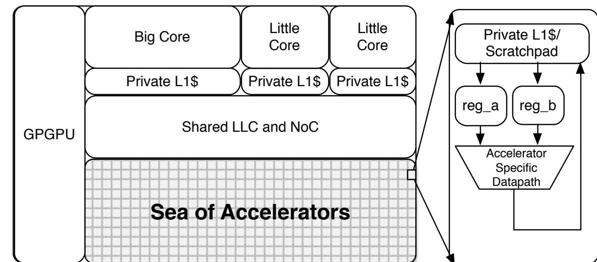


Figure 1: Future Heterogeneous Architecture.

Computer architects have long been developing and leveraging high-level power [8, 37] and performance [3, 7] simulation frameworks for general-purpose cores and GPUs [5, 36]. In contrast, current accelerator-related research primarily relies on creating RTL implementations, a tedious and time-consuming process. It takes hours, if not days, to generate, simulate, and synthesize RTL to get the power and performance of a single accelerator design, even with the help of high-level synthesis (HLS) tools. Such a low-level, RTL infrastructure cannot support architecture-level design space exploration that sweeps parameters across traditional general-purpose cores, accelerators, and shared resources such as cache hierarchies and on-chip networks. Hence, there is a clear need for a high-level design flow that abstracts RTL implementations of accelerators to enable broad design space exploration of next-generation customized architectures.

In this paper, we introduce Aladdin, a pre-RTL, power-performance simulator designed to enable rapid design space search of accelerator-centric systems. This framework takes high-level language descriptions of algorithms as inputs, and uses dynamic data dependence graphs (DDDG) as a representation of an accelerator without having to generate RTL. Starting with an unconstrained program DDDG, which corresponds to an initial representation of accelerator hardware, Aladdin applies optimizations as well as constraints to the graph to create a realistic model of accelerator activity. We rigorously validated Aladdin against RTL implementations of accelerators from both handwritten Verilog and a commercial HLS tool for a range of applications, including accelerators in Memcached [38], HARP [55], NPU [19], and a commonly used throughput-oriented benchmark suite, SHOC [17]. Our results show that Aladdin can model performance within 0.9%, power within 4.9%, and area within 6.6% compared to accelerator designs generated by traditional RTL flows. In addition, Aladdin provides these estimates over 100× faster.

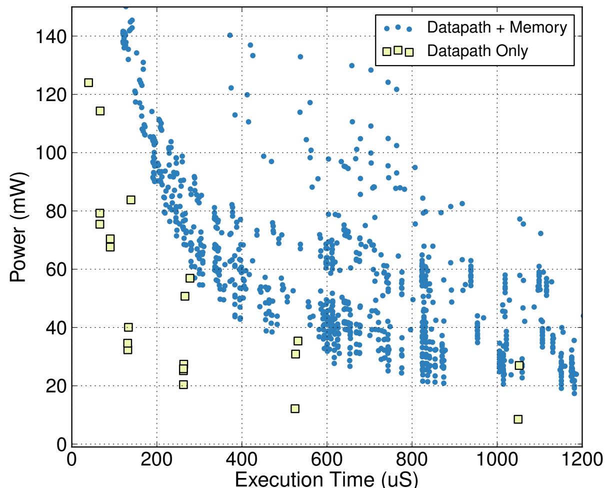


Figure 2: GEMM design space w/ and w/o memory hierarchy.

Aladdin captures accelerator design trade-offs, enabling new architectural research directions in heterogeneous systems composed of accelerators, general-purpose cores, and the shared memory hierarchy seen in today's mobile SoCs and for future customized architectures; we demonstrate this capability by integrating Aladdin with a full cache hierarchy model and DRAMSim2 [46]. Such infrastructure allows users to explore customized and shared memory hierarchies for accelerators in a heterogeneous environment. In a case study with the GEMM benchmark, Aladdin uncovers significant, high-level, design trade-offs by evaluating a broader design space of the entire system. Such analysis results in more than 3× performance improvements compared to the conventional approach of designing accelerators in isolation.

2. Background and Motivation

Hardware acceleration exists in many forms, such as analog accelerators [6, 50], static [13, 19, 28, 38, 43, 52, 55] and dynamic datapath accelerators [14, 25, 27], and programmable accelerators, such as GPUs and DSPs. In this work, we focus on static datapath accelerators. Here we discuss the design flow, design space, and state-of-the-art research infrastructure of datapath accelerators, all in order to illustrate the challenges associated with current accelerator research and why a tool like Aladdin opens up new research opportunities for architects.

2.1. Accelerator Design Flow

The current accelerator design flow requires multiple CAD tools, which is inherently tedious and time-consuming. It starts with a high-level description of an algorithm, then designers either manually implement the algorithm in RTL or use HLS tools, such as Xilinx's Vivado HLS [2], to compile the high-level implementation (e.g., C/C++) to RTL. It takes significant effort to write RTL manually, the quality of which highly depends on designers' expertise. Although HLS tools offer opportunities to automatically generate the RTL implementation, extensively tuning C-code is still necessary to meet design requirements. After generating RTL, designers must use commercial CAD tools, such as Synopsys's Design Compiler and Mentor Graphics' ModelSim, to estimate power and cycle counts.

In contrast, Aladdin takes unmodified, high-level language descriptions of algorithms, to generate a DDDG representation of accelerators, which accurately models the cycle-level power, performance, and area of realistic accelerator designs. As a pre-RTL simulator, Aladdin is orders of magnitude faster than existing CAD flows.

	Novel Accelerator Design	Accelerator Datapath Trade-offs	Heterogeneous SoC Trade-offs
handwritten RTL	Buffer-int-Cache [20], Memcached [35, 38], Sonic Millip3De [47], HARP [55]	Inadequate	Inadequate
HLS	LINQits [12], Convolution Engine [43], Conservation Cores [52]	Cong [16], Liu [40], Reagen [45]	Inadequate

Table 1: Accelerator Research Infrastructure

2.2. Accelerator Design Space

Despite the application-specific nature of accelerators, the accelerator design space is large given a range of architecture- and circuit-level alternatives. Figure 2 illustrates a large power-performance design space of accelerator design points for the GEMM workload from the SHOC benchmark suite. The square points were generated from a commercial HLS flow sweeping datapath parameters, including loop-iteration parallelism, pipelining, array partitioning, and clock frequency. However, HLS flows generally provision a fixed latency for all memory accesses, implicitly assuming local scratchpad memory fed by DMA controllers.

Such simple designs are not well suited for capturing data locality or interactions with complex memory hierarchies. The circle points in Figure 2 were generated by Aladdin integrated with a full cache hierarchy model and DRAMSim2, sweeping not only datapath parameters but also memory parameters. By doing so, Aladdin exposes a rich design space that incorporates the realistic memory penalties in terms of time and power, impractical with existing HLS tools alone. Section 5 further demonstrates the importance of accelerator datapath and memory co-design using Aladdin.

2.3. State-of-the-art Accelerator Research Infrastructure

The ITRS predicts hundreds to thousands of customized accelerators by 2022 [1]. However, state-of-the-art accelerator research projects still only contain a handful of accelerators because of the cumbersome design flow that inhibits computer architects from evaluating large accelerator-centric systems. Table 1 categorizes accelerator-related research projects in the

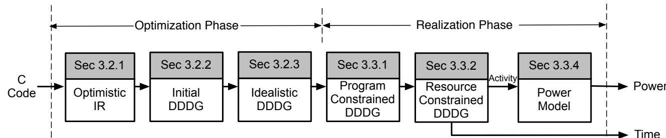


Figure 3: The Aladdin Framework Overview.

computer architecture community over the past 5 years based on the means of implementation (handwritten RTL vs. HLS tools) and the scope of possible design exploration.

We see that researchers have been able to propose novel implementations of accelerators for a wide range of applications, either writing RTL directly or using HLS tools despite the time-consuming process. With the help of the HLS flow, we have begun to see studies evaluating design trade-offs in accelerator datapaths, which is otherwise impractical using handwritten RTL. However, as discussed in Section 2.2, HLS tools cannot easily navigate large design spaces of customized architectures. This inadequacy in infrastructure has confined the exploratory scope of accelerator research.

2.4. Contributions

In summary, this work makes the following contributions:

1. We present Aladdin, a pre-RTL, power-performance simulator for fixed-function accelerators using dynamic data dependence graphs (Section 3).
2. We perform rigorous validation of Aladdin against handwritten RTL implementations and a commercial HLS design flow. We show that Aladdin can model the behavior of recently published accelerators [38, 55, 19] and typical accelerator kernels [17] (Section 4).
3. We demonstrate a large design space exploration of customized architectures, enabled by Aladdin, identifies high-level accelerator design trade-offs (Section 5).

3. The Aladdin Framework

3.1. Modeling Methodology

The foundation of the Aladdin infrastructure is the use of dynamic data dependence graphs (DDDG) to represent accelerators. A DDDG is a directed, acyclic graph, where nodes represent computation and edges represent dynamic data dependences between nodes. The dataflow nature of hardware accelerators makes the DDDG a good candidate to model their behavior. Figure 3 illustrates the overall structure of Aladdin, starting from a C description of an algorithm and passing through an *optimization* phase, described in Section 3.2, where the DDDG is constructed and optimized to derive an idealized representation of the algorithm. The idealized DDDG then passes to a *realization* phase, discussed in Section 3.3, that restricts the DDDG by applying realistic program dependences and resource constraints. User-defined configurations allow wide design space exploration of accelerator implementations. The outcome of these two phases is a pre-RTL, power-performance model for accelerators.

Aladdin uses a DDDG to represent program behaviors so that it can take arbitrary C code descriptions of an algorithm—without any modifications—to expose algorithmic parallelism. This fundamental feature allows users to rapidly investigate different algorithms and accelerator implementations. Due to its optimistic nature, dynamic analysis has been previously deployed in parallelism research exploring the limits of ILP [4, 22, 44, 53] and recent modeling frameworks for multicore processors [24, 31]. These studies sought to quickly measure the upper bound of performance achievable on an ideal parallel machine [33]. Our work has two main distinctions from these efforts. First, previous efforts model traditional Von Neumann machines where instructions are fetched, decoded, and executed on a fixed, but programmable architecture. In contrast, Aladdin models a vast palette of different accelerator implementation alternatives for the DDDG; the optimization phase incorporates typical hardware optimizations, such as removing memory operations via customized storage inside the datapath and reducing the bitwidth of functional units. The second distinction is that Aladdin provides a realistic power-performance model of accelerators across a range of design alternatives during its realization phase, unlike previous work that offered an upper-bound performance estimate.

In contrast to dynamic approaches, parallelizing compilers and HLS tools use program dependence graphs (PDG) [15, 23] that statically capture both control and data dependences [21, 26]. Static analysis is inherently conservative in its dependence analysis, because it is used for generating code and hardware that works in all circumstances and is built without run-time information. A classic example of this conservatism is the enforcement of false dependences that restrict algorithmic parallelism. For instance, programmers often use pointers to navigate arrays, and disambiguating these memory references is a challenge for HLS tools. Such situations frequently lead to designs that are more sequential compared to what a human RTL programmer would develop. Therefore, although HLS tools offer the opportunity to automatically generate RTL, designers still need to extensively tune their C code to expose parallelism explicitly (Section 4). Thus, *Aladdin is different from HLS tools*; Aladdin is simply a realistic, accurate representation of accelerators, whereas HLS is burdened with generating actual, correct hardware.

This section describes details of the optimization phase (Section 3.2) and realization phase (Section 3.3) of Aladdin. We then discuss how to integrate Aladdin with memory systems (Section 3.4) and limitations of the approach (Section 3.5).

3.2. Optimization Phase

The optimization phase forms an idealized DDDG that only represents the fundamental dependences of the algorithm. An idealized DDDG for accelerators must satisfy three requirements: (a) only express necessary computation and memory accesses, (b) only capture true read-after-write dependences, and (c) remove unnecessary dependences in the context of cus-

tomized accelerators. This section describes how Aladdin's optimization phase addresses these requirements.

3.2.1. Optimistic IR Aladdin builds the DDDG from a dynamic instruction trace, where the choice of the ISA significantly impacts the complexity and granularity of the nodes in the graph. In fact, a trace using a machine-specific ISA contains instructions that are not part of the program but produced due to the artifacts of the ISA [49], *e.g.*, register spills. To avoid such artifacts, Aladdin uses a high-level, machine-independent intermediate representation (IR) provided by the ILDJIT compiler [10]. ILDJIT IR is optimistic because it allows an unlimited number of registers, eliminating additional instructions generated due to stack overheads and register spilling. The IR contains 80 opcodes ranging from simple primitives, *e.g.*, add and multiply, to complex operators, *e.g.*, sine and square root, so that we can easily detect the functional units needed based on the program's IR trace and model them using pre-characterized hardware. We use a customized interpreter for the ILDJIT IR to emit fully-optimized IR instructions in a trace file. The trace includes dynamic instruction information such as opcodes, register IDs, parameter data types, and parameter data values. We also profile the dynamic addresses of memory operations.

3.2.2. Initial DDDG Aladdin analyzes both register and memory dependences based on the IR trace. Only true read-after-write data dependences are respected in the initial DDDG construction. This DDDG is optimistic enough for the purpose of ILP limit studies but is missing several characteristics of hardware accelerators; the next section discusses how Aladdin idealizes the DDDG further.

3.2.3. Idealized DDDG Hardware accelerators have considerable flexibility to customize datapaths for application-specific features, which is not modeled in the initial DDDG. Such customization can change the attributes of the datapath, as in the case of bitwidth reduction where functional units can be tuned to the value range of the problem. Aladdin also removes operations that are not required for hardware implementations. For example, to reduce memory bandwidth, small, frequently accessed arrays, such as filters, can be stored directly in registers inside the datapath instead of in external memory. Cost models are used to automatically perform all of these transformations.

We categorize our optimizations into node-level, loop-level, and memory-level transformations to produce an idealized DDDG representation.

Node-Level Optimization. In addition to bitwidth analysis, we also model other node-level optimizations, such as strength reduction and tree-height reduction, by changing the nodes' attributes and performing standard graph transformations [29].

Loop-Level Optimization. The initial DDDG captures the true dependences between successive iterations of the loop index variables, which means each index variable can only be incremented once per cycle. Such dependence constraints do not apply to hardware accelerators or parallel processors since it is entirely possible that they can initiate multiple iterations of a loop simultaneously [51]. Aladdin removes all dependences between loop index variables, including basic and derived induction variables, to expose loop parallelism.

Parameters	Example Range
Loop Rolling Factor	[1::2::Trip count]
Clock Period (ns)	[1::2::16]
FU latency	Single-Cycle, Pipelined
Memory Ports	[1::2::64]

Table 2: Realization Phase User-Defined Parameters, *i*::*j*::*k* denotes a set of values from *i* to *k* by a stepping factor *j*.

Memory Optimization. The goal is to remove unnecessary load/store operations. In addition to the memory-to-register conversion example described above, Aladdin also performs store-load forwarding inside the DDDG, which eliminates load operations by buffering data in internal registers within hardware accelerators. This is different from store-load forwarding in general-purpose CPUs, where the load operation must still be executed [48].

Extensibility Hardware design is open-ended, and Aladdin can be extended to incorporate other accelerator-specific optimizations, analogous to adding new microarchitectural structures to CPU simulators. We demonstrate this extensibility by considering CAM hardware to optimize data matching. A CAM is an example of a custom circuit structure that is often used to accelerate hash tables in network routers and datatype specific accelerators [56]. Unlike software, CAMs can automatically compare a key against all of the entries in one cycle. On the other hand, large CAMs are power hungry, resulting in an energy trade-off when hash tables reach a certain size. Aladdin incorporates CAMs into its customization strategy by automatically replacing software-managed hash tables with CAM. Aladdin can detect a linear search for a key by looking for chained sequential memory look-ups and comparison. Section 4.2.2 demonstrates an example with a Memcached accelerator in which CAMs are used as a victim cache to a regular hash table during hash conflicts [38].

3.3. Realization Phase

The realization phase uses program and resource parameters, defined by users, to constrain the idealized DDDG generated in the optimization phase.

3.3.1. Program-Constrained DDDG The idealized DDDG optimistically assumes that hardware designers can eliminate all control and false data dependences at design time. Aladdin's realization phase models actual control and memory dependences to create the program-constrained DDDG.

Control Dependence. The idealized DDDG does not include control dependences, assuming that branch outcomes can be known in advance and operations can start before branches are resolved, which is unrealistic even for hardware accelerators. The costs and benefits of control flow speculation for accelerators have not been extensively studied yet,

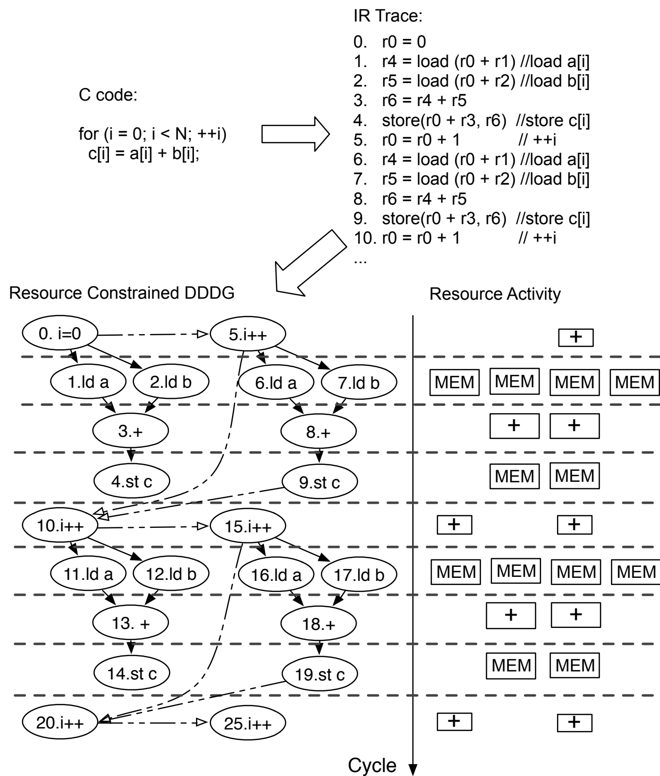


Figure 4: C, IR, Resource Constrained DDDG, and Activity.

and one solution to minimize control dependences relies on predicated execution to simultaneously execute both taken and not taken paths until branch resolution [34]. While this approach minimizes serialization, the cost of speculation is very high—it requires hardware resources that grow exponentially with the number of outstanding branches. Aladdin models control dependence by bringing code from the not-taken path into the program-constrained DDDG to account for additional power and resources. Aladdin is flexible enough to model the costs of different mechanisms for handling control flow. For energy efficiency, Aladdin models one outstanding branch at a time, serializing control dependences for multiple simultaneous branches.

Memory Dependence. The idealized DDDG optimistically removes all false memory dependences between dynamic instructions, keeping true read-after-write dependences. This is realistic for memory accesses with addresses that can be resolved at design time. However, some algorithms have input-dependent memory accesses, *e.g.*, histogram, where different inputs result in different dynamic dependences. Without runtime memory disambiguation support, designers have to make conservative assumptions about memory dependences to ensure correctness. To model realistic memory dependences, the realization phase includes memory ambiguation that constrains the input-dependent memory accesses by adding dependences between all dynamic instances of a load-store pair, as long as a true memory dependence is observed for any pair. This is similar to the dynamic dependence profiling approach adopted by parallelization efforts [24, 32].

3.3.2. Resource-Constrained DDDG Finally, Aladdin accounts for user-specified hardware resource constraints, a subset of which are shown in Table 2. Users specify the type and size of hardware resources in an input configuration file. Aladdin then binds the program-constrained DDDG onto the hardware resources, leading to the resource-constrained DDDG. Aladdin can easily sweep resource parameters to explore the design space of an algorithm, which is fast because only resource constraints need to be applied for each design point. These resource parameters are set with respect to the following three factors: loop rolling, loop pipelining, and memory ports.

Loop Rolling. The optimization phase removes dependences between loop index variables, assuming completely unrolled loops that execute all iterations in parallel. In reality, for loops with large trip counts, this leads to large resource requirements. Aladdin's loop rolling factor re-rolls loops by adding dependences between loop index variables.

Loop Pipelining. The DDDG representation fully pipelines loop iterations by default, though sometimes pipelined implementation leads to high resource requirements as well as high power consumption. Aladdin offers users the option to turn off loop pipelining by adding dependences between the entry and exit nodes of successive loop iterations.

Memory ports. The number of memory ports constrains the data transfer rate between the accelerator datapath and the closest memory hierarchy, generally either a scratchpad memory or L1 cache. Aladdin uses this parameter to abstractly model the number of memory requests the datapath can issue concurrently, and Section 3.4 discusses how the memory ports interface with memory simulators.

3.3.3. An Example Figure 4 illustrates different phases of Aladdin transformations using a microbenchmark as an example. After the IR trace of the C code has been produced, the optimization and realization phases generate the resource-constrained DDDG that models accelerator behavior. In this example, we assume the user wants an accelerator with a factor-of-2 loop-iteration parallelism and without loop pipelining. The solid arrows in the DDDG are true data dependences, and the dashed arrows represent resource constraints, such as loop rolling and turning off loop pipelining. The horizontal dashed lines represent clock cycle boundaries. The corresponding resource activities are shown to the right of the DDDG example. We see that the DDDG reflects the dataflow nature of the accelerator. Aladdin can accurately capture dynamic behavior of accelerators without having to generate RTL by carefully modeling the opportunities and constraints of the customized datapath in the DDDG.

3.3.4. Power and Area Models We now describe the construction and application of Aladdin's power and area models to capture the resource requirements of accelerators.

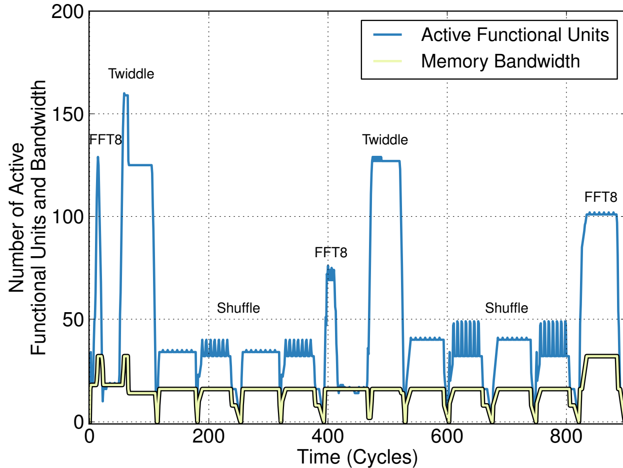


Figure 5: Cycle-by-Cycle FU and Memory Activity of FFT.

Power Model. To accurately model the power of accelerators, we need: (a) precise activities and (b) accurate power characterization of different DDDG components. We uniquely characterize switching, internal, and leakage power from Design Compiler for each type of DDDG node (multipliers, adders, shifters, etc.) and registers. The characterization accounts for different timing requirements, bitwidths, and switching activity. Switching and internal power are due to capacitive charging/discharging of output load and internal transistors of the logic gates, respectively. While switching and internal power are both dynamic, we found internal power weakly dependent on activity because internal nodes can switch without the gate output switching.

We construct a detailed power model by synthesizing microbenchmarks that exercise the functional units. Our microbenchmarks cover all of the compute instructions in IR so that there is a one-to-one mapping between nodes in the DDDG and functional units in the power model. We synthesize these microbenchmarks using Synopsys's Design Compiler in conjunction with a commercial 40nm standard cell library to characterize the switching, internal, and leakage power of each functional unit. This characterization is fully automated in order to easily migrate to new technologies.

Aladdin's power modeling library also accounts for cell selection variances during netlist synthesis. Different pipeline stages within a datapath contain varying amounts of logic and, in order to meet timing requirements, different standard cells and logic implementations of functional units are often selected at synthesis time. Aladdin approximates the impact of cell selection by training the model for a variety of timing constraints and using a first-order model to choose the correct design. This also accounts for logic flattening that Design Compiler performs across small collections of functional units.

Area Model. To accurately model area, we construct an area library similar to the previously described power library for each DDDG component. This model was obtained using the same set of microbenchmarks to characterize the area for each functional unit as well as for registers.

Cycle-Level Activity. Figure 5 shows the cycle-level resource activity for one implementation of the FFT benchmark. Aladdin accurately captures the distinct phases of FFT. The number of functional units required is estimated using the maximum number of parallel functional units per cycle for each program phase; this approximation provides the power and area models with the total resources allocated to the accelerators. The cycle-level activity is an input to the power model to represent the dynamic activity of the accelerators.

3.4. Integration with Memory System

Aladdin can easily integrate with architectural cache and memory simulators to model their behavior with a particular memory hierarchy. Within the context of memory hierarchy for accelerators, we discuss three types of memory models with which Aladdin can integrate.

Ideal Memory guarantees that all memory requests can be serviced in one cycle, which is only realistic for a system with small memory size. Aladdin models the ideal memory system by assuming load and store nodes in the DDDG take one cycle.

Scratchpad Memory is commonly used in accelerator-centric systems where accelerator designers explicitly manage memory accesses so that each request has a fixed latency. However, this approach requires a detailed understanding of workload memory characteristics. This potentially increases design time but leads to more efficient implementation. Aladdin can take a parametrized memory latency as an input to model the latency of load and store operations matching the characteristics of scratchpad memory.

Cache Hierarchy applies a hardware-managed cache system to capture locality of the accelerated workload. Such a cache hierarchy relies on the hardware to exploit the locality of the workload, potentially easing the design of systems with a large number of accelerators. On the other hand, a cache introduces variable memory latency. Existing cache simulators can be integrated with Aladdin to evaluate how variable latency memory accesses affect accelerator behaviors.

In order to integrate with a cache hierarchy, the accelerator must include certain mechanisms to react to possible cache misses. Aladdin models several approaches to handle this variable latency, which resemble pipeline control mechanisms in general-purpose processors. The simplest policy is local or global pipeline stalls on miss events. We also consider a more complex mechanism for non-blocking behavior in which a new loop iteration is started when a miss occurs, and only the loop ID is stored for re-execution when the miss resolves.

Memory Power Model. The memory power model is based on a commercial register file and SRAM memory compiler that accompanies our standard cell library. We have compared the memory power model to CACTI [54] and found consistent trends, but we retain the memory compiler model for consistency with the standard cell library.

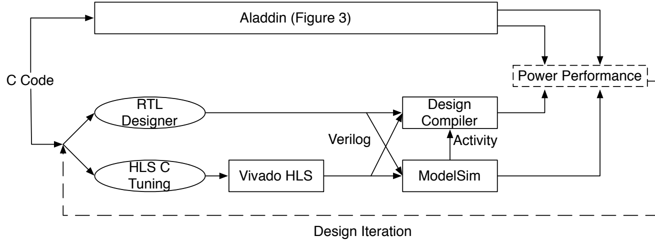


Figure 6: Validation Flow.

3.5. Limitations

Algorithm choices. Aladdin does not automatically sweep different algorithms. Rather, it provides a framework to quickly explore various hardware designs of a particular algorithm. This means designers can use Aladdin to quickly and quantitatively compare the design spaces of multiple algorithms for the same application to find the most suitable algorithm choice.

Input Dependent. Like other dynamic analysis frameworks [30, 41], Aladdin only models operations that appear in the dynamic trace, which means it does not instantiate hardware for code not executed with a specific input. For Aladdin to completely model the hardware cost of a program, users must provide inputs that exercise all paths of the code.

Input C code. Aladdin can create a DDDG for any C code. However, in terms of modeling accelerators, C constructs that require resources outside the accelerator, such as system calls and dynamic memory allocation, are not modeled. In fact, understanding how to handle such events is a research direction that Aladdin facilitates.

4. Aladdin Validation

We begin this section with a detailed description of the traditional RTL design flow and workloads used to validate Aladdin. Validation results show Aladdin has modest error rates within 0.9% for performance, 4.9% for power, and 6.5% for area. Aladdin generates the design space more than 100× faster than the traditional RTL-based flow.

4.1. Validation Flow

Figure 6 outlines the methodology used to validate Aladdin. The power and area estimates of Aladdin are compared against synthesized Verilog generated by Design Compiler using commercial 40nm standard cells. Aladdin's performance model is validated against ModelSim Verilog simulations. The SAIF activity file generated from ModelSim is fed to Design Compiler to capture the switching activity at the gate level. To generate Verilog, we either hand-code RTL or use Xilinx's Vivado HLS tool. The RTL design flow is an iterative process and requires extensive tuning of both RTL and C code.

4.1.1. HLS Tuning We use HLS to generate the accelerator design space for SHOC benchmarks to demonstrate Aladdin's ability to explore a large design space of an accelerator's datapath, which is infeasible with handwritten RTL. To produce

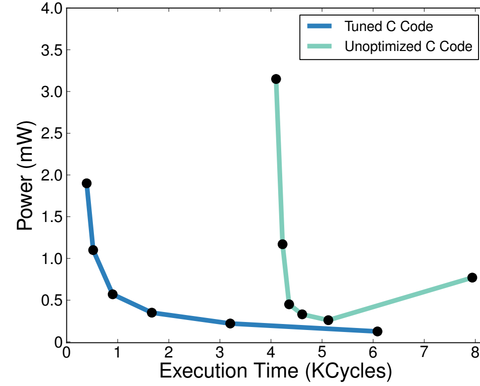


Figure 7: Unoptimized vs. Tuned Scan.

high-quality Verilog, HLS requires significant tuning of the input C code to expose parallelism and remove false dependences. In contrast, Aladdin produces the power-performance optimal design points without modifying the input C code.

Figure 7 demonstrates the quantitative difference that code quality can have on power and performance by comparing Pareto frontiers of optimized and unoptimized versions for the *Scan* benchmark. Both curves were generated by sweeping loop unrolling factors, memory bandwidth, and resource sharing and applying loop pipelining, similar to the parameters discussed in Section 3. The unoptimized C code hits a performance wall at around 4000 cycles where neither increasing bandwidth nor loop parallelism yields better performance but continues to burn more power. The reason is that when striding over a partitioned array being read from and written to in the same cycle, though accessing different elements of the array, the HLS compiler conservatively adds loop-carried dependences. This in turn increases the iteration interval of loop pipelining, limiting performance. To overcome HLS's conservative assumptions, we partition the array differently which consequently simplifies the access patterns to resolve false dependences. Similar tuning was necessary to generate well-performing designs for each of the SHOC benchmarks, which are then used to validate Aladdin in Section 4.3.

4.2. Applications

We implemented a collection of benchmarks, both by hand and using HLS, to validate Aladdin. HLS enabled the validation of the Pareto optimal designs for the SHOC benchmarks, overcoming the impracticality of hand coding each design point. We also validate Aladdin against handwritten RTL for benchmarks ill-suited for HLS. Examples are taken from recently published accelerator research: NPU [19], Memcached [38], and HARP [55].

4.2.1. SHOC The SHOC benchmark suite is representative of many typical accelerator workloads, which includes compute intensive benchmarks where functional units often dominate execution time and power, *e.g.*, Stencil, as well as memory-bound workloads, *e.g.*, Sort, stressing Aladdin's modeling capabilities across multiple dimensions. To ensure valid, well

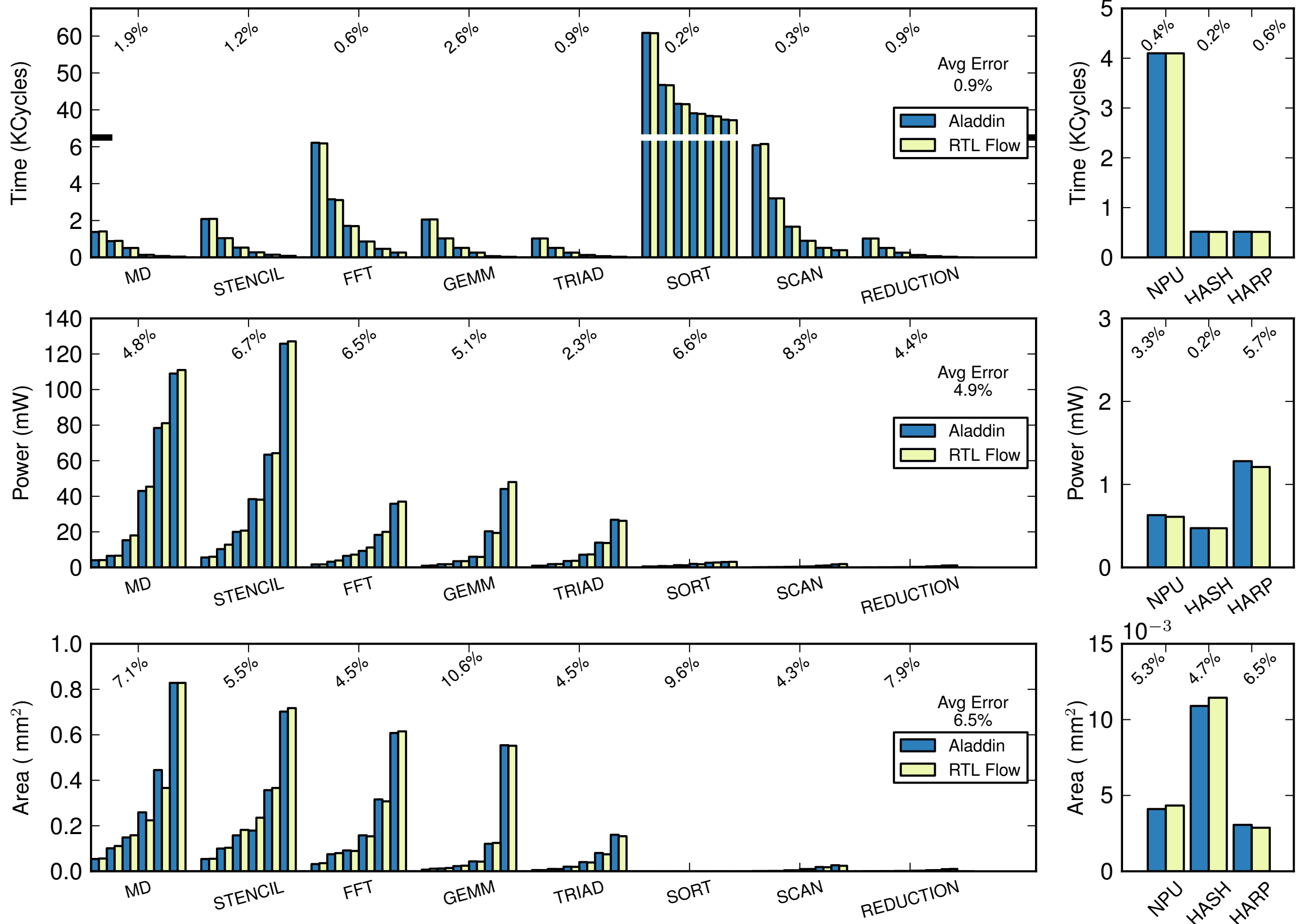


Figure 8: Performance (top), Power (middle), and Area (bottom) Validation.

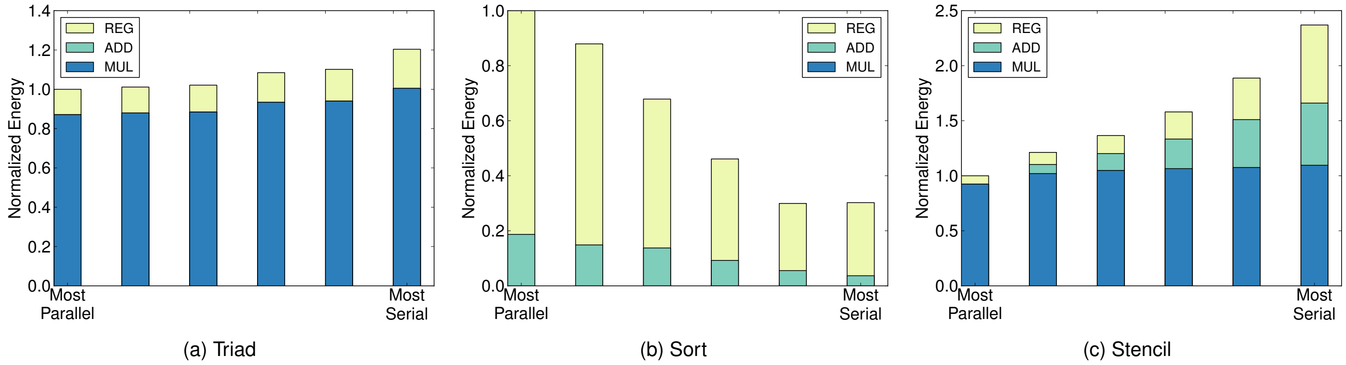
performing HLS results, we carefully tuned each implementation as described earlier. By sweeping loop unrolling factors and resource constraints such as memory bandwidth, a large design space of accelerator datapaths for each benchmark is generated.

4.2.2. Single Accelerators In some instances, the expressiveness of C limits the ability for HLS to reasonably match hand-coded RTL. Therefore, we hand coded RTL for HARP, NPU, and Memcached to further demonstrate Aladdin's modeling capabilities. For Aladdin, we rely on generic C implementations that describe the behavior of each accelerator.

HARP is a partitioning accelerator for big data [55]. Essentially, given a stream of inputs, it uses a pipeline of comparators to check each input against a splitter value at each stage and categorize the inputs. HARP is a control-intensive workload where its activity highly depends on the input values, which makes it a good candidate to exercise Aladdin's ability to model control behavior. Our handwritten Verilog for HARP properly expresses the pipelined comparisons. Aladdin was able to match the Verilog implementation through the loop rolling and pipelining parameters.

NPU is a network of individual neurons connected through a shared bus, which communicates with each other in a carefully orchestrated, compiler-generated pattern. The design hinges on an input FIFO to buffer computations. Although HLS has FIFO support, the ability to finely share data efficiently between compute engines is a shortcoming of most HLS tools. An individual neuron was implemented in Verilog, and a synthetic input was used to stimulate the neuron. Aladdin's memory-to-register transformation successfully captures such FIFO-type structure.

Memcached is a distributed key-value store system, the central of which is a hash function and CAM lookup. Given an input key, a hash accelerator computes the value using a hash algorithm described in [11]. The value is then used to index four SRAMs whose content is compared against the input key to determine a hit. If one of the SRAMs returns a match, it returns that SRAM's data. On a miss, the value is sent to a CAM where all possible locations of the key are checked in parallel and the correct value is returned. This benchmark serves two purposes—to demonstrate Aladdin's ability to model variable bitwidth computations (the hash function) and to model a different customization strategy (CAM).



(a) Triad (b) Sort (c) Stencil

Figure 9: Energy Characterization of SHOC.

4.3. Validation

Figure 8 shows that Aladdin accurately models performance, power, and area compared against RTL implementations across all of the presented benchmarks with average errors of 0.9%, 4.9%, and 6.5%, respectively. For each SHOC workload, we validated six points on the Pareto frontier, *e.g.*, points in Figure 7. The SHOC validation results show Aladdin accurately models entire design spaces, while for single accelerator designs, Aladdin is not subject to HLS shortcomings and can accurately model different customization strategies.

Pareto Analysis The Pareto optimal designs of the SHOC benchmarks reveal interesting program characteristics in the context of hardware accelerators. Bars in Figure 9 correspond to six designs along each benchmark's Pareto frontier, which were also used for validation. In each graph, the leftmost bar is the most parallel, highest performing design while the rightmost bar is the most serial and lowest performing design. For each design, we calculate energy using power and performance estimates from Aladdin. Aladdin's detailed power model enables energy breakdowns for adders, multipliers, and registers. The six bars of each benchmark are normalized to the leftmost bar to facilitate comparisons.

Each of the three benchmarks in Figure 9 exhibits different energy trends across the Pareto frontier. Triad, shown in Figure 9a, demonstrates good energy proportionality, meaning more parallel hardware leads to better performance with a proportional power cost. In contrast, Sort has a strong sequential component such that energy increases for more parallel designs without improving performance. Finally, while the multiplier energy for Stencil shows similar energy proportionality to Triad, the adders and registers required for loop control are amortized with more parallelism. Non-intuitively, this leads to better energy efficiency for these faster designs.

	Hand-Coded RTL	HLS	Aladdin
Programming Effort	High	Medium	N/A
RTL Generation	Designer Dependent	37 mins	
RTL Simulation Time	5 mins		
RTL Synthesis Time	45 mins		
Time to Solution per Design	87 mins		1 min
Time to Solution (36 Designs)	52 hours		7 mins

Table 3: Algorithm-to-Solution Time per Design.

4.4. Algorithm-to-Solution Time

Aladdin enables rapid design space exploration of accelerator designs. Table 3 quantifies the differences in algorithm-to-solution time to explore a design space of the FFT benchmark with 36 points. Compared to traditional RTL flows, Aladdin skips the time-consuming RTL generation, synthesis, and simulation process. On average, it takes 87 mins to generate a single design using the RTL flow but only 1 min for Aladdin, including both of Aladdin's optimization phase (50 seconds) and realization phase (12 seconds). However, because Aladdin only needs to perform the optimization phase once for each algorithm, this optimization time can be amortized across large design spaces. Consequently, it only takes 7 mins to enumerate the full design space with Aladdin compared to 52 hours with the RTL flow. The HLS RTL generation time per design is comparable to that reported by other researchers [39].

5. Case Study: GEMM Design Space

We now present a case study that demonstrates how Aladdin enables architecture research and why it is invaluable to future heterogeneous SoC designs. We focus our analysis on GEMM as it has complex memory behavior and consider a problem size of 196 KB. In this case study, we present:

1. Execution Time Decomposition: Understand design tradeoffs of an accelerator's execution time with respect to *compute time* and *memory time*.
2. Accelerator Design Space: Characterize the design space of GEMM accelerators, including memory hierarchy, to understand how different parameters affect the design space.
3. Heterogeneous SoC: Demonstrate the impact of resource contention in an SoC-like system of a single accelerator, resulting in different optimal designs that would be unknown without system-level analysis.

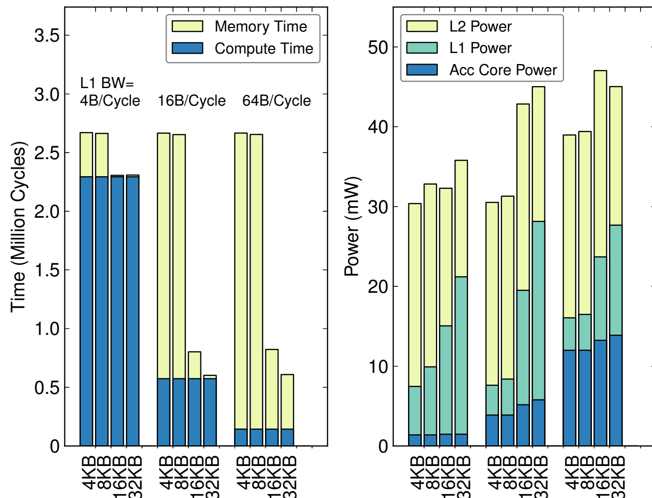


Figure 10: GEMM Time and Power Decomposition

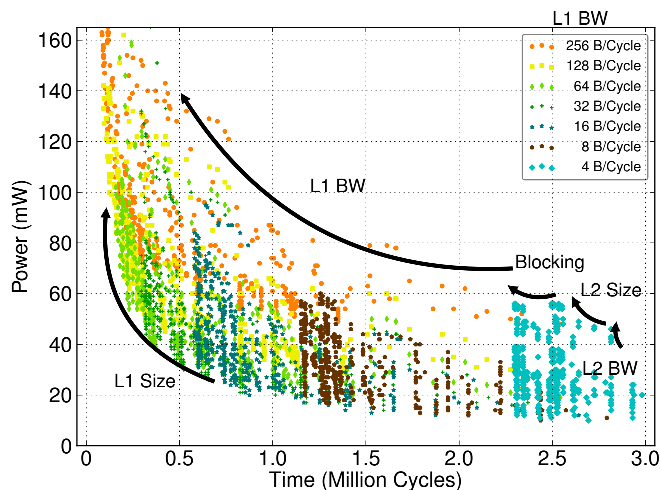


Figure 11: GEMM design space.

5.1. Execution Time Decomposition

So far, Aladdin has been evaluated as a standalone accelerator simulator with an ideal memory hierarchy (one cycle memory access latency). However, it is not always possible to retrieve data in one cycle in real designs with large problem sizes. The efficiency of accelerators highly depends on the memory system. To quantify the impact of a memory system on accelerators, we integrate Aladdin with a standard cache simulator and the DRAMSim2 memory simulator [46].

We divide the accelerator's execution time into *compute time* and *memory time*. Compute time is defined as the execution time of an accelerator when there is only one cycle memory latency. Memory time is defined as cycles lost to a non-ideal memory, which includes both memory bandwidth and memory latency constraints.

In order to decompose the accelerator's execution time, we run simulations with both an ideal memory and a realistic memory hierarchy including L1, L2, and DRAM. The compute time is the execution time with ideal memory; the delta of execution times between the two simulations is the memory time to get data into accelerators [9].

Table 4 lists all of the parameters in the design space. In this section, we focus on the bandwidth and size of L1. Figure 10 shows the execution time and power breakdown of the GEMM benchmark when sweeping L1 size and bandwidth. On the left of Figure 10, we observe that memory time takes a significant portion of the execution time, especially as L1 bandwidth increases. With the same L1 bandwidth, execution time decreases as the L1 size increases from 8 KB to 16 KB; this phenomenon occurs because 8 KB is not large enough to hold the blocked data size (a 32×32 matrix).

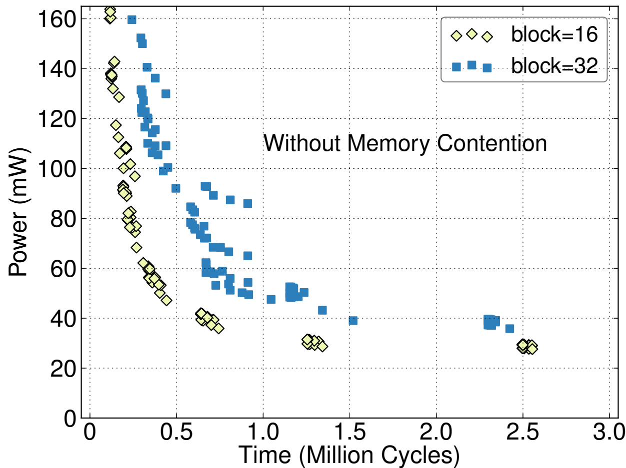
Type	Parameters	Values
Core	Blocking Factor	[16, 32]
L1	L1 Bandwidth (Bytes/Cycle)	[4::2::128]
	L1 Size (KB)	[4::2::32]
	MSHR Entries	[4::2::64]
L2	L2 Bandwidth (Bytes/Cycle)	[4::2::128]
	L2 Size (KB)	[64::2::256]
	L2 Assoc	16

Table 4: Single Accelerator Design Space, where *i*::*j*::*k* denotes a set of values from *i* to *k* by a stepping factor of *j*.

The plot on the right shows the power breakdown of the accelerator datapath, L1, and L2. The accelerator datapath power increases with L1 bandwidth, because higher bandwidth enables more-parallel implementations. As L1 size increases, its power also increases as accesses become more expensive. At the same time, L2 power decreases because more accesses are coalesced by the L1, lowering the L2 cache's activity. In fact, cache power consumes more than half of the total power, even for more parallel designs where datapath power is significant. Therefore, design efforts focusing on the accelerator datapath alone do not alleviate memory power, which dominates the overall power cost.

5.2. Accelerator Design Space

Section 5.1 explored a subset of the design space for accelerators and memory systems. Here, we use Aladdin to explore the comprehensive design space with parameters in Table 4. Figure 11 plots the power and execution time of the GEMM accelerator designs resulting from the exhaustive sweep. The design space contains several overlapping clusters of similar designs. The arrows in Figure 11 identify correlations in power/performance trends with respect to each parameter. For example, GEMM experiences substantial performance benefits from a larger L1 cache, but with a significant power penalty. In contrast, increasing L2 size only modestly increases both power and performance.



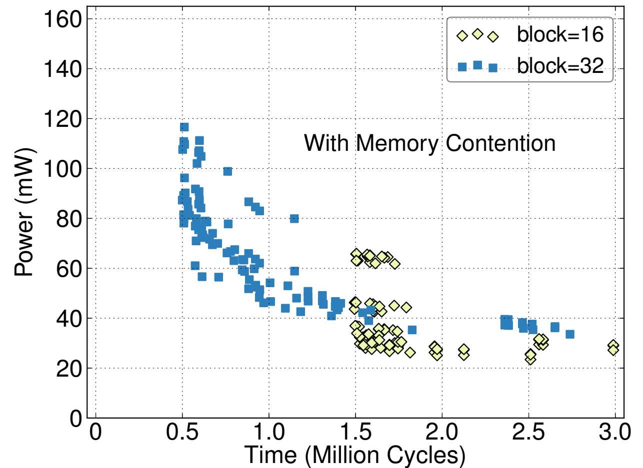


Figure 12: Design Space of GEMM without and with contention in L2 cache.

5.3. Resource-Sharing Effects in Heterogeneous SoC

In a heterogeneous system, shared resources, such as a last-level cache, can be accessed by both general-purpose cores and accelerators. We consider the case of a heterogeneous system consisting of a shared 256 KB L2 cache, one general-purpose core, and a GEMM accelerator with a private 16 KB L1 cache. From an accelerator designer's perspective, an important algorithmic parameter is the blocking factor of GEMM; a larger blocking factor exposes more algorithmic parallelism, however, achieving good locality requires a larger cache.

Figure 12(left) shows the accelerator design space without memory contention from the general purpose core. We modulate the algorithmic blocking factor and find that a blocking factor of 16 is always better than 32 with respect to both power and performance. This occurs because a 16 KB L1 cache is large enough to capture the locality of the blocking factor 16 but not 32. Therefore, it is preferable to build the accelerator with blocking factor 16 when there is no contention for shared resources.

To model resource contention between the general-purpose core and the accelerator, we use Pin [42] to profile an x86 memory trace and then use the trace to issue requests that pollute the memory hierarchy while simultaneously running the accelerator. The design space for the accelerator under contention is shown in Figure 12(right). We see that performance degrades for both blocking factors of 16 and 32 due to pollution in the L2 cache; however, blocking factor 32 suffers much less than blocking factor 16. When there is contention, capacity misses increase for the shared L2 cache, which incurs large main memory latency penalties. With a larger blocking factor, the accelerator requires fewer references to the matrices in total and, thus, fewer data requests from the L2 cache. Consequently, the effects of resource contention suggest building an accelerator with a larger blocking factor, where the accelerator performance can achieve around 0.5 million cycles. On the other hand, without considering the contention, designers may pick a design with blocking factor 16, the highest performance of which is 1.5 million cycles in the contention scenario. Such design choice leads to a 3× performance degradation. Aladdin can easily evaluate these types of system-wide accelerator design trade-offs, a task that is not tractable with other current accelerator design tools.

6. Conclusion

We have presented Aladdin, a pre-RTL, power-performance accelerator simulator offering architects the ability to quickly and accurately model accelerators without generating RTL. Validation of Aladdin with respect to designs generated by handwritten RTL and a commercial HLS flow confirmed high accuracy with average power, performance, and area errors within 0.9%, 4.9%, and 6.5%, respectively. Furthermore, Aladdin runs more than 100× faster than traditional RTL design flows. Aladdin's speed and accuracy open up opportunities for large design space exploration of customized architectures. Our case study shows that Aladdin can highlight how system-level parameters affect accelerator design trade-offs when integrated with a standard cache and DRAM simulator.

Acknowledgments

We would like to thank Glenn Holloway for his help revising this work. This work was partially supported by C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, the National Science Foundation (NSF) Expeditions in Computing Award #: CCF-0926148, DARPA under Contract #: HR0011-13-C-0022, and a Google Faculty Research Award. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

References

[1] "The international technology roadmap for semiconductors (itrs), system drivers, 2007, http://www.itrs.net/."

[2] "Xilinx vivado high-level synthesis," http://www.xilinx.com/products/design-tools/vivado/.

[3] T. M. Austin, E. Larson, and D. Ernst, "Simplescalar: An infrastructure for computer system modeling," *IEEE Computer*, 2002.

[4] T. M. Austin and G. S. Sohi, "Dynamic dependency analysis of ordinary programs," in *ISCA*, 1992.

[5] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *ISPASS*, 2009.

[6] B. Belhadj, A. Joubert, Z. Li, R. Héliot, and O. Temam, "Continuous real-world inputs can open up alternative accelerator designs," in *ISCA*, 2013.

[7] N. L. Binkert, B. M. Beckmann, G. Black, S. K. Reinhardt, A. G. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Computer Architecture News*, 2011.

[8] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: a framework for architectural-level power analysis and optimizations," in *ISCA*, 2000.

[9] D. Burger, J. R. Goodman, and A. Kagi, "Memory bandwidth limitations of future microprocessors," in *ISCA*, 1996.

[10] S. Campanoni, G. Agosta, S. Crespi-Reghizzi, and A. D. Biagio, "A highly flexible, parallel virtual machine: Design and experience of ildjit," *Software Practice Expererience*, 2010.

[11] S. R. Chalamalasetti, K. Lim, M. Wright, A. AuYoung, P. Ranganathan, and M. Margala, "An fpga memcached appliance," in *FPGA*, 2013.

[12] E. S. Chung, J. D. Davis, and J. Lee, "Linqits: big data on little clients," *ISCA*, 2013.

[13] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai, "Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus?" in *MICRO*, 2010.

[14] N. Clark, A. Hormati, and S. A. Mahlke, "Veal: Virtualized execution accelerator for loops," in *ISCA*, 2008.

[15] J. Cong, Y. Fan, G. Han, and Z. Zhang, "Application-specific instruction generation for configurable processor architectures," in *FPGA*, 2004.

[16] J. Cong, K. Gururaj, and G. Han, "Synthesis of reconfigurable high-performance multicore systems," in *FPGA*, 2009.

[17] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (shoc) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, 2010.

[18] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," *Micro, IEEE*, 2012.

[19] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *MICRO*, 2012.

[20] C. F. Fajardo, Z. Fang, R. Iyer, G. F. Garcia, S. E. Lee, and L. Zhao, "Buffer-integrated-cache: a cost-effective sram architecture for handheld and embedded platforms," in *DAC*, 2011.

[21] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," in *Symposium on Programming*, 1984.

[22] B. A. Fields, R. Bodík, and M. D. Hill, "Slack: Maximizing performance under technological constraints," in *ISCA*, 2002.

[23] M. Fingeroff, *High-Level Synthesis Blue Book*, 2010.

[24] S. Garcia, D. Jeon, C. M. Louie, and M. B. Taylor, "Kremlin: rethinking and rebooting gprof for the multicore age," in *PLDI*, 2011.

[25] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, "Dyser: Unifying functionality and parallelism specialization for energy-efficient computing," *IEEE Micro*, 2012.

[26] V. Govindaraju, T. Nowatzki, and K. Sankaralingam, "Breaking simd shackles with an exposed flexible microarchitecture and the access execute pdg," in *PACT*, 2013.

[27] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August, "Bundled execution of recurring traces for energy-efficient general purpose processing," in *MICRO*, 2011.

[28] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding sources of inefficiency in general-purpose chips," in *ISCA*, 2010.

[29] W. Hunt, B. A. Maher, D. Burger, and K. S. Mckinley, "Optimal huffman tree-height reduction for instruction-level parallelism," *Technical Report TR-08-34, Department of Computer Sciences The University of Texas at Austin*, 2008.

[30] H. C. Hunter and W. mei W. Hwu, "Code coverage and input variability: effects on architecture and compiler research," in *CASES*, 2002.

[31] D. Jeon, S. Garcia, C. M. Louie, and M. B. Taylor, "Kismet: parallel speedup estimates for serial programs," in *OOPSLA*, 2011.

[32] M. Kim, H. Kim, and C.-K. Luk, "Sd3: A scalable approach to dynamic data-dependence profiling," in *MICRO*, 2010.

[33] M. Kumar, "Measuring parallelism in computation-intensive scientific/engineering applications," *IEEE Trans. Computers*, 1988.

[34] M. S. Lam and R. P. Wilson, "Limits of control flow on parallelism," in *ISCA*, 1992.

[35] M. Lavasani, H. Angepat, and D. Chiou, "An fpga-based in-line accelerator for memcached," *IEEE Computer Architecture Letters*, 2013.

[36] J. Leng, T. H. Hetherington, A. ElTantawy, S. Z. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "Gpuwattch: enabling energy optimizations in gpgpus," in *ISCA*, 2013.

[37] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO*, 2009.

[38] K. T. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, "Thin servers with smart pipes: designing soc accelerators for memcached," in *ISCA*, 2013.

[39] H.-Y. Liu and L. P. Carloni, "On learning-based methods for design-space exploration with high-level synthesis," in *DAC*, 2013.

[40] H.-Y. Liu, M. Petracca, and L. P. Carloni, "Compositional system-level design exploration with planning of high-level synthesis," in *DATE*, 2012.

[41] S. Lu, P. Zhou, W. Liu, Y. Zhou, and J. Torrellas, "Pathexpander: Architectural support for increasing the path coverage of dynamic bug detection," in *MICRO*, 2006.

[42] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *PLDI*, 2005.

[43] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, "Convolution engine: balancing efficiency & flexibility in specialized computing," in *ISCA*, 2013.

[44] L. Rauchwerger, P. K. Dubey, and R. Nair, "Measuring limits of parallelism and characterizing its vulnerability to resource constraints," in *MICRO*, 1993.

[45] B. Reagen, Y. S. Shao, G.-Y. Wei, and D. Brooks, "Quantifying acceleration: Power/performance trade-offs of application kernels in hardware," in *ISLPED*, 2013.

[46] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *IEEE Comput. Archit. Lett.*, 2011.

[47] R. Sampson, M. Yang, S. Wei, C. Chakrabarti, and T. F. Wenisch, "Sonic millip3de: A massively parallel 3d-stacked accelerator for 3d ultrasound," in *HPCA*, 2013.

[48] T. Sha, M. M. K. Martin, and A. Roth, "Nosq: Store-load communication without a store queue," in *MICRO*, 2006.

[49] Y. S. Shao and D. Brooks, "Isa-independent workload characterization and its implications for specialized architectures," in *ISPASS*, 2013.

[50] O. Temam, "A defect-tolerant accelerator for emerging high-performance applications," in *ISCA*, 2012.

[51] K. B. Theobald, G. R. Gao, and L. J. Hendren, "On the limits of program parallelism and its smoothability," in *MICRO*, 1992.

[52] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: reducing the energy of mature computations," *ASPLOS*, 2010.

[53] D. W. Wall, "Limits of instruction-level parallelism," in *ASPLOS*, 1991.

[54] S. J. E. Wilton and N. P. Jouppi, "Cacti: An enhanced cache access and cycle time model," *IEEE Journal of Solid-State Circuits*, 1996.

[55] L. Wu, R. J. Barker, M. A. Kim, and K. A. Ross, "Navigating big data with high-throughput, energy-efficient data partitioning," in *ISCA*, 2013.

[56] Q. Zhu, B. Akin, H. E. Sumbul, F. Sadi, J. Hoe, L. Pileggi, and F. Franchetti, "A 3d-stacked logic-in-memory accelerator for application-specific data intensive computing," in *3DIC*, 2013.