Duality Cache for Data Parallel Acceleration

Daichi Fujiki
dfujiki@umich.edu
University of Michigan

Scott Mahlke
mahlke@umich.edu
University of Michigan

Reetuparna Das
reetudas@umich.edu
University of Michigan

Abstract

Duality Cache is an in-cache computation architecture that enables general purpose data parallel applications to run on caches. This paper presents a holistic approach of building *Duality Cache* system stack with techniques of performing in-cache floating point arithmetic and transcendental functions, enabling a data-parallel execution model, designing a compiler that accepts existing CUDA programs, and providing flexibility in adopting for various workload characteristics.

Exposure to massive parallelism that exists in the *Duality Cache* architecture improves performance of GPU benchmarks by 3.6× and OpenACC benchmarks by 4.0× over a server class GPU. Re-purposing existing caches provides 72.6× better performance for CPUs with only 3.5% of area cost. *Duality Cache* reduces energy by 5.8× over GPUs and 21× over CPUs.

ACM Reference Format:
Daichi Fujiki, Scott Mahlke, and Reetuparna Das. 2019. Duality Cache for Data Parallel Acceleration. In *ISCA '19: The 46th International Symposium on Computer Architecture, June 22–26, 2019, Phoenix, AZ*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3307650.3322257

1 Introduction

Modern general purpose processors and accelerators are integrated with large on-chip caches to fully exploit locality. They are utilized as a low-latency temporary storage and occupy a large fraction (over 70%) of the die area. For example, the latest Intel's server class Xeon processors devote more than 30MB SRAM just for the last level cache (LLC). Furthermore, data-movement over the cache hierarchy is costly, both in terms of time and energy.

To tackle these inefficiencies, recent works re-purpose the elements in cache structures and transform them into large data-parallel compute units. Compute Caches [2] introduces an in-SRAM computing technique referred to as bit-line computing, which activates multiple word lines and performs *logical operations*. Neural Cache [10] further augments compute capability to efficiently support *fixed point arithmetic* operations. Neural Cache transforms a 35 MB Xeon Cache into 1,146,880 *bit-line ALUs* with a die area overhead of 2%. The proposed *bit-line ALU* operates on transposed or vertically aligned data in a bit-serial manner. These additional compute resources improve the efficiency of Convolutional Neural Networks (CNNs) by 679× (speedup 18.3×, energy savings 37.1×) over a CPU (Xeon E5) and 128× over a GPU (Titan Xp). The source of the efficiency is the combined effect of reduced data movement and massive parallelism.

While compute-capable caches offer significant benefits, previous works have just provided low-level interface for in-cache operation [2] or relied on a manual mapping of convolution kernels to the cache arrays [10]. This paper proposes the *Duality Cache* system stack that makes in-cache computing accessible to general purpose data-parallel programs.

Our proposed system solves several challenges to make caches capable of general purpose data processing. *First*, to address a wide set of data-intensive applications, having a rich set of computation primitives is essential. Prior work is limited to logical and fixed-point arithmetic operations. Most data-parallel workloads require floating point operations. Manipulation of mantissa based on exponents in an in-cache vector architecture is a non-trivial challenge. We devise techniques that support bit-serial floating point operations for applications with high precision or large dynamic range demands. We present techniques that reduce the latency of bit-serial operations based on the dynamic range of operands. The proposed techniques can support 1,146,880 parallel floating-point operations at 3.5% processor die area overhead for a Xeon E5-2697 with 35MB cache. CORDIC algorithms [37, 38] are leveraged to support in-cache transcendental functions.

Second, a critical challenge for in-cache computing is the design of the interface between the CPU cores and compute caches, execution model, and cache addressing structure. Operands of in-cache operations need to be aligned on a bit-line ALU (constraining them to specific locations in cache). We address these problems by developing a single instruction multiple thread (SIMT) architecture, where each thread is mapped to bit-line ALUs. The data bit-cells on a bit-line ALU become thread-local bit-serial registers which are directly addressable in the instruction set architecture (ISA). Compute

ISCA '19, June 22–26, 2019, Phoenix, AZ

ACM ISBN 978-1-4503-6669-4/19/06…$15.00
https://doi.org/10.1145/3307650.3322257

operations are allowed only on the thread-local registers. *Duality Cache* threads are organized into control blocks and mapped to cache ways. We design a micro-architecture that orchestrates control block SIMT instructions. The processor can switch between cache mode and accelerator mode. The SIMT *Duality Cache* architecture is activated only in accelerator mode. *Duality Cache* extensions incur a modest area overhead (3.5%) but do not affect the functionality or performance of conventional cache mode operation.

Finally, compute capable caches require a programming model and compiler that are capable of *exposing parallelism* in applications to the underlying hardware and harnessing its full potential. We adopt CUDA/OpenACC as a programming model and develop a compiler which can translate arbitrary CUDA/OpenACC programs to the *Duality Cache* ISA. The compiler allocates resources, schedules VLIW instructions, and conducts several optimizations exploiting unique opportunities in our in-cache architecture. We also develop compiler assisted techniques to flexibly allocate a fraction of cache to be used as SIMT compute units and regular cache storage.

In summary, this paper offers the following contributions:

- We design *Duality Cache* architecture that re-purposes caches on demand to data-parallel accelerators capable of executing arbitrary programs. The proposed architecture adopts the SIMT execution model. Cache data arrays act both as vector processing units and register file. Each thread supports in-order VLIW instructions.
- *Duality Cache* features a Turing complete ISA similar to NVIDIA's PTX [27]. We extend SRAM arrays to support floating point operations and leverage the dynamic range of operands to reduce bit-serial operation latency. In-SRAM transcendental functions (sin, cos, etc.) are supported using CORDIC algorithms [37, 38].
- We develop a compiler that translates CUDA/OpenACC programs to native *Duality Cache* ISA. The compiler implements several optimizations to enhance parallelism and efficiency within the constraints of in-cache computation, exploiting the unique architectural features.
- We observe some applications exploit locality through on-chip storage such as shared memory in CUDA. To respond to these demands, we develop a compiler-assisted framework that adjusts the portion of cache which is used as SIMT compute units. The remaining operates as cache.
- We compare a Xeon server with *Duality Cache* extensions to a Xeon server with a Titan Xp GPU. Our experimental results show that *Duality Cache* can provide an overall speedup of 3.6× for Rodinia benchmarks and 4.0× for OpenACC benchmarks over the GPU. Compared to a CPU, *Duality Cache* provides 72.6× speedup for Rodinia benchmarks and 9.6× for OpenACC benchmarks. The proposed architecture improves the energy efficiency by 5.9× and reduces the average power by 1.6× compared to a GPU. *Duality Cache* extensions incur an area of 15.8 mm^2 in 22 *nm* (resulting in a 3.5% area cost over a CPU die), while the evaluated GPU die area is 471 mm^2 in 16 *nm*.

2 Background and Motivation

2.1 Bit Serial In-Cache Computation

Compute Caches [2] introduces an in-cache computation framework that supports `copy`, `zeroing`, `xor`, `compare`, and `search`. Jeloka *et al.* [19] shows data corruption due to multi-row access is prevented by lowering the word-line voltage to bias against write of SRAM array. Their measurement across 20 test chips fabricated using 28 *nm* technology demonstrates no data corruption even with activating 64 word-lines simultaneously for in-place computation. They also demonstrate the stability of six sigma robustness, equivalent to industry standard robustness against process variation, by Monte Carlo simulation.

Neural Cache [10] expands on compute cache's logical operation capabilities to support arithmetic operations inside the SRAM arrays for machine learning workloads. Neural Cache chooses to implement a bit-serial architecture as opposed to bit-parallel. Bit-parallel requires communicating data across bit-lines to propagate a carry. By using a bit-serial format, carries can be stored in a latch along the bit-line, saving the complexity of communication across bit-lines and also allowing configurable precision. A few hardware transpose memory units (TMUs) are placed in the cache control box and used to transpose the inputs to allow bit-serial computations. TMU design is based on an 8T SRAM array.

Data is mapped to a transposed layout where different bit-lines hold data from different elements in the operand vector. Each n-bit element is stored across n word-lines, and thus each word-line holds one *bit-slice* from 256 vector elements as shown in Figure 1 (c). The bits in each bit-slice are of the same bit position.

By activating two word-lines in the SRAM, we are able to sense `logical and` at bit-line (BL) and `logical nor` at bit-line complement (BLB). Note, a re-configurable differential senseamp [2] is used to sense BL and BLB independently. A 1-bit *full adder* can be created by augmenting few gates to the ends of the sense amps as shown in Figure 1 (d). Thus, when activating two word-lines, we can add the values in each word-line together with the carry in the latch and generate a new sum and carry. The sum can be written to a new word-line in the same cycle. By adding each bit iteratively, we can perform the addition of two n bit numbers in n cycles. Multiplication takes $n^2 + 3n - 2$ cycles and is performed as a series of additions of partial products.

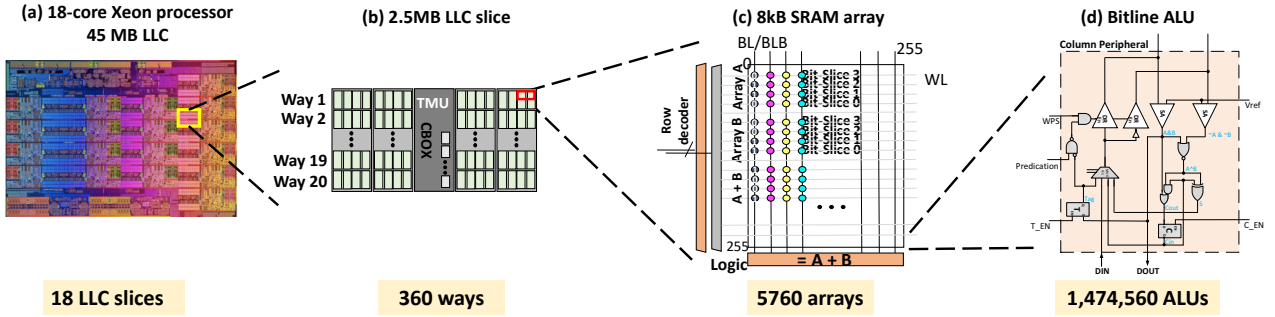


Figure 1. Neural Cache architecture [10].

Bit-serial computing in cache provides massive throughput. In the above SRAM architecture, 256 bit-lines in one 8 KB SRAM array are turned into 256 *bit-line ALUs* in a vector unit, and 5760 such 8 KB arrays in a 45MB LLC transform to 1,474,560 bit-serial ALUs (Figure 1) operating at frequency of 2.5 GHz when computing. Note, while a 45 MB LLC cache access from core takes 20-30 ns, the smaller 8 KB SRAM arrays can themselves operate at a frequency up to 4 GHz [16].

2.2 Motivation

Duality Cache can morph general purpose processors into data-parallel accelerators. In this context, its compute resources are comparable to GPGPU, a representative throughput-oriented parallel accelerator. Although performance bottlenecks of GPGPU are workload dependent, commonly claimed causes include CPU-GPU communication through PCIe bus, load imbalance, on-chip storage size, bandwidth utilization, and compute flops [6, 23, 40]. *Duality Cache* can alleviate these bottlenecks.

Data movement between CPU host and accelerator device. Since disjoint address space of GPU and CPU necessitates explicit data transfer through PCIe, workloads with fine-grained interleaving of serial and parallel phases are difficult to achieve speedups due to communication overheads. Likewise, initial data transfer between the host and device memory is costly especially when data reuse is not high. *Duality Cache* has an advantage of tight integration with the host memory hierarchy and can minimize these overheads.

Cost. While tighter integration of GPU and CPU can alleviate the above problems, the area of modern GPUs (e.g. Titan Xp die area is 471 mm^2 in 16 nm) makes on-die integration with CPU impractical. In contrast, *Duality Cache* extensions require an area of 15.8 mm^2 in 22 nm, while providing nearly 9.3× more compute resources, making it a cost effective solution. Besides area savings, cost manifests itself in terms of power usage and maintenance. The TDP of a server with Xeon E5 dual socket processor and Titan XP GPU is 640 W, whereas TDP of a server with Xeon E5 processor extended with *Duality Cache* is 296 W (Table 2).

Increased on-chip memory capacity. On-chip SRAM can alleviate external memory bandwidth pressure and help reduce memory access latency. GPU's cache size is limited compared to CPU as its die area is dominated by compute units. *Duality Cache* can provide flexible partitioning of compute and cache allocation, which enables memory bounded applications to benefit from a large cache allocation. GPU's memory bandwidth resources can potentially be underutilized by not having enough kernels that request memory accesses. In such case, *Duality Cache* can increase bandwidth utilization by having enough active kernels exploiting its higher compute resources.

3 System Stack

In this section, we present a system stack for *Duality Cache* for accelerating data-parallel applications. This section discusses the proposed bit serial arithmetic primitives, execution model, microarchitecture, compiler, and programming model.

3.1 ISA

Prior works support a limited set of logical and integer operations. Compute caches [2] introduces basic bit-parallel operations which perform logical operations to horizontally aligned data in caches. Neural cache [10] proposes bit-serial computation that enables several integer operations of vertically aligned data. Our work proposes general ISA for in-cache architecture, leveraging the bit-serial computation scheme. Proposed ISA adopts an early version of PTX (SM2.x), an ISA for low-level parallel thread execution virtual machine employed in NVIDIA's compiler for Fermi GPU family [27]. Our ISA is thus Turing complete. This design choice is made to maximize portability of existing source code while minimizing hardware complexity; any other operations that are not natively supported by the ISA are dealt with by compiler lowering and/or a software library.

Table 1 lists major arithmetic operations supported by our ISA, their algorithm, and baseline latency. Machine learning workloads, which Neural Cache targets, can provide reasonable results using reduced precision datatypes (e.g. 8-bit fixed point). However, a class of scientific applications requires more precision in computation, which necessitates full 32-bit integer or floating point support. In this work, we develop in-cache floating point arithmetic. Since some operations listed take latency that scales quadratic with the size of data, native implementation of the algorithms presented in the

Operation	Type	Algorithm	Latency
add	uint, int	[10]	n
sub	uint, int	Bit-serial*	$2n$
mul	uint	[10]	$n^2 + 3n - 2$
mul	int	Bit-serial*	$n^2 + 5n$
div, rem	uint	[10]	$1.5n^2 + 5.5n$
div, rem	int	Bit-serial*	$1.5n^2 + 9.5n$
and, or, xor	uint	[2]	n
shl, shr	uint, int	Bit-serial*	n^2
add, sub	float	Bit-serial*	$O(n^2)$ - variable
mul	float	Bit-serial*	$O(n^2)$ - variable
div	float	Bit-serial*	$O(n^2)$ - variable
sin, cos	fixed point	CORDIC*	$(7k + 1)n + 7k + 1$
exp	fixed point	CORDIC*	$4kn + 4k + 2$
log	fixed point	CORDIC*	$4kn + 4k$
sqrt	fixed point	CORDIC*	$4kn + 4k$
rsqrt	float	Fast inverse square root*	$O(n^2)$ - variable

Table 1. Supported in-cache arithmetic operations.
* = This work. n = #of bits of datatype. k = iteration count.

past work may critically impact performance. Below we discuss our techniques to minimize the bit-serial latency for these operations based on their dynamic range.

3.1.1 Floating Point Arithmetic

Prior in-cache architectures do not support floating point (FP) arithmetic. Unlike integer and fixed-point arithmetic, floating point needs normalization of exponents, which requires shift operations of mantissa by an arbitrary value for addition and subtraction.

The proposed algorithm for floating point addition is shown in Algorithm 1. The algorithm leverages bit-serial fixed point addition and subtraction operations discussed in Neural Cache [10]. A floating point addition first requires normalization or shifting of mantissa by the difference of exponents. Note a compute SRAM array is a SIMD unit which does exactly same operation on 256 operands (vectors `A` and `B`) at the same time (Figure 2). The proposed design first computes the difference in exponents for all vector elements (vector `ediff`).

The next step needs to shift second operands' mantissa (`B[i].mnt`) by the difference of exponents (`ediff[i]`) for all i such that `ediff[i]` > 0 and then add it to the first operand's mantissa (`A[i].mnt`). We introduce `arshadd` (arithmetic right shift and add) primitive to accomplish this in few cycles. `arshadd` is equivalent to $a + (b \gg d)$. *For given d, shift operation is free for bit-serial architecture.* For example, $a + (b \gg 1)$ can be performed by activating correct bits (a_i and b_{i+1}) and adding them.

Since the *vector* architecture of compute SRAM arrays forces all threads in an array to perform exactly the same operation, arithmetic shift by the values in `ediff` vector may take in the worst case $O(n^2)$ cycles for n-bit data, since there are 256 values in `ediff` vector and each element of vector is shifted serially. We observe that the dynamic range of exponents is small in real-world workloads and leverage this to reduce the operation latency. The algorithm takes $O(dn)$ cycles by searching for all unique d ediff values instead of the worst-case. Note that the worst case variation of d is equal to the number of mantissa bits (23 for IEEE 754 FP32). We do a leading zero search to find the upper-bound value of `ediff` to be searched.

Algorithm 1 Floating Point Add (C=A+B)

```
1:  procedure VECTOR_FPADD
2:      ARSHADD(X, Y, k) = X + (Y ≫ k)
3:      type float {.exp, .mnt}
4:      vector <float> A, B, C
5:      vector ediff ← A.exp − B.exp
6:      if ediff[i] < 0 then
7:          SWAP(A[i], B[i])
8:          ediff[i] ← A[i].exp − B[i].exp
9:      end if
10:     for each unique k in ediff do
11:         if ediff[i] == k then
12:             C[i].mnt =
13:             ARSHADD(A[i].mnt, B[i].mnt, k)
14:         end if
15:     end for
16:     if overflow_i then
17:         C[i].exp = A[i].exp + 1; C[i].mnt ≫ 1
18:     else
19:         C[i].exp = A[i].exp
20:     end if
21: end procedure
```

The `search` operation for each unique value is executed in two cycles as follows. In the first cycle, all word-lines that correspond to bit 1s in the search value are activated and logical `AND` of the bit-positions is sensed on each bit-line. In the second cycle, all word-lines that correspond to bit 0s in the search value are activated and `NOR` result is sensed on each bit-line-bar. A logical `AND` of the results from these two cycles produces the final search hit vector.

Additionally, we swap operands with negative `ediff` as shown in Figure 2, to avoid divergent execution. Without this swap, we have to repeat the for-block (Line 10 in Algorithm 1) twice, which dominates the processing time of the naive algorithm. Therefore, it is worth doing a swap.

Floating point addition and subtraction require conversion of sign bit format to 2's complement format, also unhiding the implicit leading digit. The mantissa of input values to Algorithm 1 is in 2's complement format, and this conversion is handled by an instruction that precedes. We also introduce an instruction that does re-conversion to sign bit format and mantissa normalization. We minimize the number of

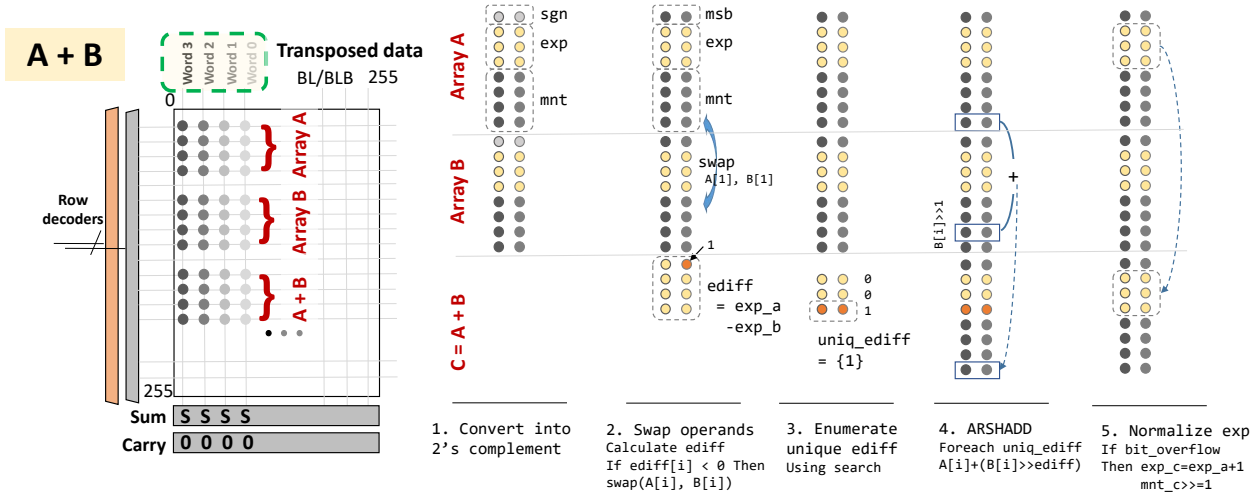


Figure 2. In-SRAM floating point addition overview. The mantissa (mnt) is normalized with the difference in exponents (exp) using a search operation.

conversions by skipping re-conversions between operations. This is helped by a compiler analysis, which scans through the input code and inserts these conversion operators.

We also support efficient floating point multiplication and division. Floating point multiplication (division) is a combination of addition (subtraction) of exponent bits and multiplication (division) of mantissa bits, where we can apply the same technique as integer multiplication (division) which we will discuss in the following section.

We do not support denormal floating point numbers (non-zero numbers with magnitude smaller than the smallest normal number). Note that since denormal number handling significantly reduces process speed in general, some systems omit this hardware support. Intel's SIMD instruction set handles it by calling a software exception, also providing a knob to disable the exception call [17].

3.1.2 Integer Arithmetic Optimization's

We apply several optimizations to the baseline integer algorithm to skip redundant cycles depending on the data. For example, when performing multiplication, we can avoid calculating partial sums if i-th bit are all zero across all the data entry. Below lists other optimizations we introduce for multiplication and division.

- We perform a leading zero search on multiplicands and dividends to identify the effective data size. Leading k zeros will reduce more than $n \times k$ cycles.
- We perform a leading zero search on divisors. Since we know the number of digits of quotient Q of A/B is at most $x = \lfloor \log A - \log B \rfloor + 1$, we can skip the first $n - x$ iterations, which saves more than $\sum_{i=0}^{x-1}(n+i)$ cycles. Leading zeros of divisors can also contribute to reducing cycles by interpreting it as data with a smaller datatype. Note that the leading zero searches can be done in parallel for both operands using `search`.
- We perform `search` on the partial residues to judge whether they are zero. For example, 1001/10 will see zero partial residue after calculating Q=01xx. We set 0 to the third MSB of Q without performing subtraction in the iteration.

3.1.3 Transcendental Functions

In addition to floating point operations, we support transcendental functions. Previous work on in-memory memristive computing [15] utilizes look-up tables (LUTs) for those functions to get initial guess and refines it by an iterative process such as the Newton-Raphson method. However, this approach not only requires a large area for LUTs for each cache bank but also makes LUTs a serialization point which ends up in limiting computation throughput.

For our in-cache architecture, we utilize a different algorithm called COordinate Rotation DIgital Computer or CORDIC [4, 24, 37, 38]. CORDIC does not require accessing LUTs for each operand value but calculates and refines the result digit-by-digit using pre-calculated constant numbers that can be shared by any operand value. CORDIC does not make any serialization point, which makes it highly efficient for the *Duality Cache* 's massively parallel vector architecture. Furthermore, with using pre-calculated constants, CORDIC only involves addition, subtraction, and fixed-amount-bitshift, but not multiplication, thus being suitable for bit-serial computing, as their latency is $O(n)$ (mul is $O(n^2)$). Further, our compiler exposes the ILP in CORDIC algorithms with its VLIW instruction scheduling.

CORDIC approach can be applied to various operations including exp, log, trigonometric / hyperbolic functions, and square root. We set the iteration count to 17 to retain the accuracy of FP32 format. While our CORDIC implementation accept fixed point numbers with fixed region (e.g. $[0°, 90°]$), it is trivial to normalize data to fit within the region (e.g. $\sin(120°) = \sin(120° - 90°)$, $\log 1234 = 3 \times \log 1.234$). This

normalization and type conversion to the fixed point are handled by a software library.

3.2 Programming Model

Programming model creates a direct and significant impact on programmability and architecture design. While simple models (e.g. wide SIMD [10]) simplify the hardware, it limits flexibility. On the other hand, guaranteeing too much freedom may result in over provisioning of hardware resources in order to handle all communication patterns. To expose the massive parallelism of *Duality Cache* to applications with irregular (or data-dependent) memory access, we adopt a SIMT programming model of CUDA (NVIDIA's GPGPU programming framework) and OpenACC.

CUDA describes kernels as multi-threaded programs and groups threads into warps. In a warp, threads are executed in a synchronized manner. Inter-thread synchronization and sharing are allowed within a group of threads called thread block or Cooperative Thread Array (CTA). In other words, different CTAs are independent and can be scheduled and executed in any order.

Proposed architecture benefits from this programming model from two aspects. First, CUDA is a popular and widely used framework across different fields spanning from scientific computing to machine learning. Leveraging it for *Duality Cache* architecture with direct translation or trivial source code changes will archive portability and opportunity to use the existing software. Second, having independent CTAs entails minimum network resources for inter-thread communications that happen locally within a CTA.

On top of CUDA, we support OpenACC. OpenACC provides OpenMP-like pragma to programmers, making it easier to convert existing serial programs to parallel programs. Currently, commodity OpenACC compilers support multi-thread, multi-core CPUs and NVIDIA GPUs. OpenACC is characterized by its ability to describe fine-grained interleaving of serial computation on the host and parallel kernels to be executed on an accelerator (e.g. GPU) using pragma. While GPUs tends to face communication bottleneck for those OpenACC programs with frequent host-device communication, *Duality Cache* enables seamless execution between host code and kernel code, as caches share the same memory hierarchy as the host.

3.3 Execution Model and Architecture

Our execution model reflects the programming model, but when compared to GPUs it is simpler and coarser-grained. The cache acts in two modes: accelerator mode and cache mode. In *accelerator mode* a bit-line in a cache array becomes one thread lane of a SIMT processor. In our architecture, registers and compute units are identical. We assign registers in a thread to the bit-line and perform computation in-place. Operands are vertically aligned within the registers mapped on the bit-line. In *cache mode*, the cache arrays are part of the processor's traditional multi-level memory hierarchy. Note that the accelerator mode does not change the functionality or performance of the cache mode.

SIMT Architecture. Instruction issue is performed at the **Control Block (CB)** granularity. Figure 3 shows the *Duality Cache* architecture. A CB consists of a group of 1,024 threads and is allocated to a single way of a cache slice. Each way consists of 4 banks, each bank is capable of executing 256 threads referred to as **Thread Block (TB)**. Hereafter, we use TB to refer to this 256 thread group, and CTA to *software* thread block of CUDA or a *gang* of OpenACC.

We choose to dedicate an entire bank with four SRAM arrays to a TB to provide a sufficient number of registers per thread and prevent frequent register spilling. One SRAM array has 256 bit-cells along a bit-line, thus can afford only 8 32-bit bit-serial registers as shown in Figure 3 (b). By allocating 256 threads to a bank of four SRAM arrays we can afford 32 32-bit bit-serial registers per thread. Thus each thread in a TB is virtually mapped to multiple arrays in a bank, and each member array has a slice of registers. The proposed architecture restricts the maximum number of threads in a CTA to 1,024.

We only allow inter-thread communication within CB. This design choice is made to balance programmability and hardware complexity. We utilize a 256×256 local crossbar in the C-Box to shuffle / broadcast CTA local data as shown in Figure 3 (a). Although the throughput of the crossbar is limited by the interconnect bandwidth, it can service arbitrary inter-thread communication within a CB in a fixed time-frame. Kernels that do not require inter-thread communication can span across multiple CBs.

A CB and GPU's Streaming Multiprocessor (SM) are similar in the thread and register capacity. While latest GPUs have large register files (64K 32-bit reg / SM) so that cores can time multiplex different warps in blocks assigned to the SM, we directly execute instructions in-situ in numerous *Duality Cache* threads mapped within the registers.

In-Order VLIW Architecture. Mapping a TB to a bank of four arrays opens up an interesting opportunity: each array can execute a different operation in the same cycle. Thus we can perform *VLIW-like instruction scheduling*, allowing *Duality Cache* to exploit ILP in the program. All the banks in a way (i.e. CB) process the same (VLIW) instructions. Instructions are buffered in the tag array in each way which is continuously fed entries by a host processor core. The buffered instructions are then decoded and issued through four issue windows where each instruction is broadcasted to corresponding member arrays of all threads in the CB. *Duality Cache* performs computation in a bit-serial manner. Each bit-line acts as a computation unit, and all bit-lines in an array perform the same operation as in a SIMD processor.

Instruction Sequencing. All threads in a Control Block (CB) perform blocking execution, including memory accesses,

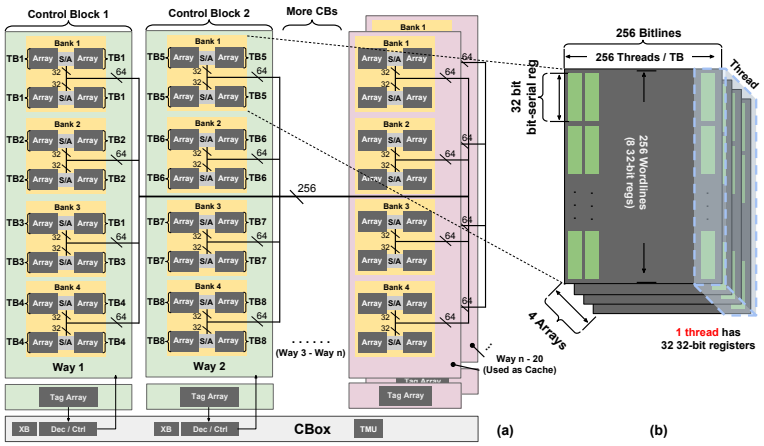


Figure 3. In-Cache SIMT execution model and architecture overview.

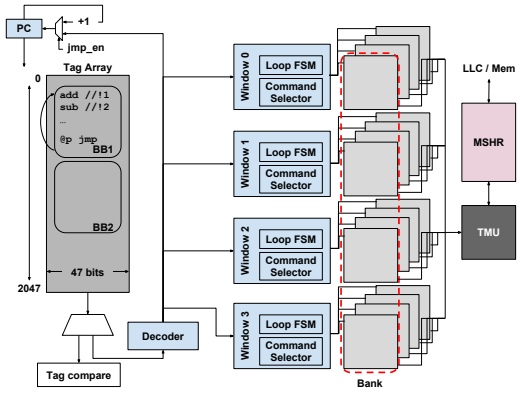


Figure 4. Frontend architecture.

with implicit synchronization between threads. On the other hand, different CBs can execute different instructions at a time. This implements compute and memory access overlap at a coarse level: while GPUs schedule for warps to overlap compute and memory access within the block, we fire a lot more CBs at one time using rich compute/register resources (9.3× more than Titan Xp) and overlap memory accesses with other CB's computations.

This means each CB maintains its own programming counter (PC). PC is incremented before fetching next instruction and points to an entry in the tag array, which works as a ring buffer. The frontier PCs are monitored by the host processor to prevent overwriting instructions that have not been executed. While fixed length loops are unrolled by host run-time or by the compiler, data-dependent loops in applications which iterate under a condition (e.g. convergence) are handled inside CB. For these loops, we maintain entire loop body block in the tag array, and a conditional jump (predicated jump) instruction resets the PC to the loop entry. The loop exits upon negative jump_en, which receives wired-OR of predicate bits stored in the cache peripheral. A CPU core can continue to fill successor instructions of the loop, but it cannot overwrite the loop block until after PC exits the loop.One CPU core is sufficient to launch and feed all CBs. Control flow is handled by predication, and indirect jumps (branches) are not supported, following the PTX language.

The control box in a cache slice is implemented with finite state machines (FSMs) in hardware that can dispatch low-level control signals to the cache banks for performing cycle-by-cycle bit-serial operations based on issued instructions.

Load and Store Instructions. *Duality Cache* interfaces with memory hierarchy through Transpose Memory Unit (TMU) [10]. TMUs have 8T transpose bit-cells which can read and write data in both horizontal and vertical directions to enable the conversion between regular bit-parallel layout and transposed bit-serial format. TMUs are placed in cache control box (CBox in Figure 3). When performing load instruction, target addresses are first read out from an array that belongs to one of the issue windows. Unlike other compute operations, only one memory instruction can be included in one VLIW instruction because of the interconnect bandwidth. The bit-serial addresses are transposed in TMU, registered in Miss Status Handling Register (MSHR), and sent to the memory. MSHR enables simple memory coalescing; duplicated accesses to a cache line are treated as an MSHR hit and suppressed. MSHR keeps track of source thread numbers. When the target cache line arrives from the memory, it is first sent to TMU. The destinations are set by configuring the local crossbars so that it can rearrange or multicast data into the data banks. The data is then read out from TMU in bit-serial format and sent to awaiting threads through the crossbar.

Data that can be accessed by *Duality Cache* has to be stored in specially allocated pages (DC-pages) in the main memory address space. A simple MMU placed at the memory controller performs address translation. The address translation is mainly aimed to balance the DRAM load by shuffling the physical address allocation.

3.4 Compiler

We develop a backend compiler that transforms PTX, NVIDIA's low-level parallel thread execution virtual machine ISA, into VLIW-style code for *Duality Cache* which we refer to as DC-PTX. Opcodes of DC-PTX are a subset of PTX opcodes; some instructions designed specifically to GPUs are eliminated. On the other hand, DC-PTX adds several fields to PTX to include operand locations.

Figure 5 shows the overall compilation flow. CUDA source code is first compiled by NVIDIA's CUDA compiler (nvcc). The output CUDA executable includes three kinds of object files (i.e. elf, PTX, and SASS). Our backend compiler extracts and parses the PTX files, applies several optimization passes to PTX IR, schedules instructions, allocates resources, and

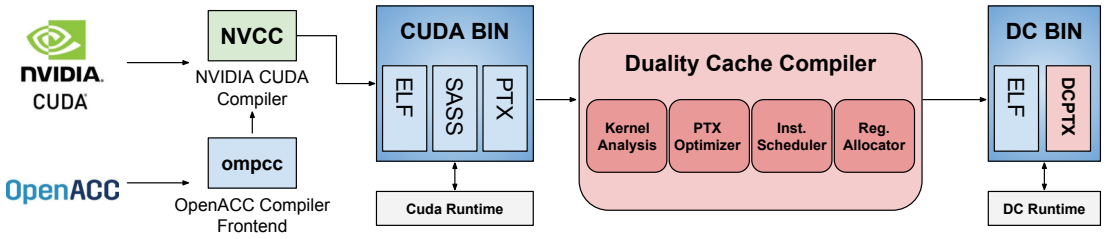


Figure 5. Compilation tool flow. CUDA source code is first compiled by NVIDIA CUDA compiler (nvcc). *Duality Cache* compiler extracts PTX assembly from CUDA executable and generates DC-PTX code. OpenACC program is compiled by an OpenACC compiler which generates GPU code that is then compiled by nvcc.

generates DC-PTX code. DC-PTX kernel is loaded and executed by API calls to DC-Runtime library in a similar way as CUDA runtime.

The compiler is currently built on top of GPU Ocelot dynamic compilation framework [9]. We choose PTX as IR since most of the CUDA compilation tool flow is closed-source (including ptxas which performs resource allocation and scheduling). Currently, GPU Ocelot is the only compilation framework academically available to work on GPU object files. We also utilize Rose compilation framework [8] to perform source-to-source compilation to apply optimization passes to the source code before nvcc compilation.

OpenACC programs can also be compiled using the same infrastructure, except that the source code is first compiled by an OpenACC compiler which extracts the accelerator code and generates GPU code that is then internally compiled by nvcc.

Duality Cache compiler framework translates a CUDA code to VLIW SIMD ISA. Although VLIW is not as efficient as out-of-order execution for exploiting ILP, it enables ILP to be exploited with lower hardware complexity since complicated ILP aware scheduling is handled by the compiler. Unlike traditional VLIW architecture, the proposed *Duality Cache* architecture has to take operand locality into account; all operands need to reside in the member array where the operation is executed, otherwise, we have to explicitly copy the operands to the member array.
Following are the implemented features of our compiler:

Register Pressure Aware Instruction Scheduling
Register pressure and efficient VLIW instruction scheduling are an inseparable problem. In our design, instruction scheduling is tightly coupled with resource allocation. While many compilers for VLIW architecture schedule instructions first before register allocation to maximize parallelism utilizing abundant register resources shared by many execution units, our execution model has limited number of private registers, which may result in frequent register spilling. On the other hand, resource-allocation-first approaches often introduce many false dependencies in return for minimized register usage, which can reduce available parallelism. We tackle this problem by performing resource allocation and instruction scheduling at the same time. We use Bottom-Up Greedy (BUG) [11] as the baseline scheduling algorithm, and linear scan register allocation as the baseline resource allocation algorithm. By taking register pressure into account while performing instruction scheduling, the compiler can pick better strategy to balance parallelism and register spilling. In our design, we allocate computation units considering register pressure as well as operand movement overhead. This approach balances the register pressure of each member array and maximizes parallelism as long as there are available registers. When register pressure is too high for all the member arrays, we start spilling a register according to the spill policy of the linear scan algorithm. Parallelism can be sacrificed due to high data movement cost.

PTX Optimizations
AST balancing: To maximize ILP, it is better to distribute operands of a chain of associative binary operations evenly to available VLIW slots. Generally compiler frontend left-folds an expression of binary operation chain if it does not have parentheses when constructing Abstract Syntax Tree (AST) (e.g. a + b + c + d ⇒ (+ (+ (+ a b) c) d)), making a true dependency between the temporary value (partial sum) and the next operand. One of the optimizations we apply reconstructs the AST to form a balanced tree (e.g. a + b + c + d ⇒ (+ (+ a b) (+ c d))) so that unnecessary dependencies will not hinder exploring ILP when scheduling instructions for our in-order VLIW architecture.

Thread independent variable isolation: We further include an optimization to reduce register pressure by not storing thread independent variables. For example, a fixed length loop is unrolled by *Duality Cache* runtime and the induction variable is provided as a constant if necessary. DC compiler identifies thread independent variables by conducting dependency analysis and affixes metadata as a marker for the instruction that only processes thread independent variables.

3.5 Cache Partitioning
Duality Cache architecture can utilize memory arrays in LLC for both computing and caching. Generally, CUDA programs are optimized for GPUs, which typically have 88-144KB SRAM storage in SM for L1+texture cache and shared memory (Pascal GPUs). Therefore, reserving one way (128KB) per CB provides a similar configuration as GPUs. However,

cache utilization is highly dependent on applications, and our architecture is able to flexibly adjust the cache resource allocation based on reuse patterns. Prior works [20, 39] have shown that some classes of GPU applications are known to receive small benefits from caches because of less locality. Also, compute intensive kernels can underutilize memory bandwidth. In those cases, we can increase the compute allocation in the cache. On the other hand, we observe many applications with irregular memory access patterns benefit from caches if the working set fits in the caches. Here we can increase the cache allocation to reduce memory bandwidth pressure.

Our compiler can analyze kernel dimension and shared memory usage to determine the cache allocation so that we can leverage the locality of the applications which is explicitly specified by programmer in the form of shared, constant, or texture memory (Note that these local memories are remapped to global memory by the compiler.) We also analyze memory access patterns, and estimate memory traffic and data reuse through static kernel code instrumentation.

4 Methodology

Benchmarks: We use applications from Rodinia GPU benchmark suite [6] and PathScale OpenACC benchmark [30] as listed in Table 3. We compile the CUDA applications using nvcc 4.2 using default compile options of the benchmark suite (except for the target architecture which we set to sm_20 to make CUDA binary compatible with GPU Ocelot [9]). The OpenACC applications are compiled using Omni Compiler [36], an open-source academic OpenACC compiler, which is internally linked with CUDA Toolkit. We modify the source code of Omni Compiler to disable the automatic insertion of cache configuration API calls which are not supported by CUDA Toolkit 4.2. While we use the old CUDA version to work with GPU Ocelot dynamic compilation tool [9], we use latest CUDA Toolkit 9.2 [26] and community standard PGI OpenACC compiler 18.10 [28] for the native run on GPU.

We mostly use the dataset preset by the benchmark suite. For some benchmarks, such as gaussian, lud, nn from Rodinia, we use the OpenMP dataset or a data-generator generated large dataset to augment the utilization of computation unit. Moreover, we modify the source code of some benchmarks to further expose the parallelism of *Duality Cache* . These custom optimizations are discussed in detail in Section 5.4.

Area and Power Model: Area and power parameters are summarized in Table 4. The energy and power model of *Duality Cache* peripherals and Transpose Memory Unit is from Neural Cache [10]. We synthesize the controller and state machine using Synopsis Design Compiler in a 45nm process. We assume average ILP 1.25 and 10% activity factor for TDP. We employ the energy and power model in [1] for the local crossbar and assume an activity factor of 5% for TDP. We use CACTI [25] to model energy and area for scratch SRAM used in MSHR.

Power and energy for CPU and DRAM activity are measured by profiling microbenchmarks using Intel Rapl interface. We use NVIDIA nvprof to measure GPU power.

Performance Model: We develop a *Duality Cache* timing model and functional model on GPU Ocelot's tracer framework and PTX emulator. Since some of the in-cache operations are data dependent, the timing model interacts with the functional model in the emulator. Target applications are executed on the DC-PTX emulator in GPU Ocelot and we obtain front-end and back-end traces using our tracer for each CTA. We then rerun the traces using our simulator and feed the trace files to Ramulator [22]. We perform CPU-trace driven DRAM simulation on Ramulator with a modified processor model.

5 Results

5.1 Configurations Studied

In this section, we evaluate the proposed *Duality Cache* and compare it to two baselines. The first baseline (CPU) uses Intel Xeon E5-2697 v3 multi-socket server. The second baseline (GPU) is a server with host Xeon E5 and NVIDIA Titan Xp GPU. The details of both configurations are shown in Table 2. We assume *Duality Cache* to be implemented in the 2-socket Xeon server system. When entire LLC geometry is allocated for computation, *Duality Cache* has 150× more threads than GPU. The massive parallelism comes at the cost of larger operation latency of the bit-serial algorithms (Section 5.5).

5.2 Performance

In this section, we study the application performance. The execution time for the GPU server and *Duality Cache* server is shown in Figure 6 (normalized to GPU, lower is better). It shows the breakdown of memcpy time and kernel execution time for GPU. We consider the memory transfer (cudaMemCopy) time, but time spent on GPU initialization, CUDA API calls (including CUDA malloc), and OpenACC API calls is not included. GPU's kernel time includes memory access time. For *Duality Cache* , we show compute and memory access time. The compute time of *Duality Cache* is the aggregate latency of issued instructions on the critical path, and the memory access time is total time − compute time.

Duality Cache provides a 3.6× average speedup for the Rodinia benchmarks and 4.0× speedup for the OpenACC benchmarks, compared to GPU. Figure 10 shows the average system speedup of *Duality Cache* compared with CPU. *Duality Cache* provides a 72.6× speedup compared to CPU for the Rodinia benchmarks, and 9.6× for the OpenACC benchmarks. The OpenACC benchmarks can have fine-grain serial and parallel interleaving making their GPU acceleration less effective compared to Rodinia benchmarks.

We discuss the key factors that contribute to the *Duality Cache* performance below:

A. Reduced Memcpy Time: Memcpy time takes a substantial portion of the GPU execution time for some applications.

Model	Die					Benchmark Servers			
	mm^2	nm	GHz	TDP	On-Chip Memory	Dies	DRAM Size	mm^2	TDP
Xeon E5-2697 v3	456	22	2.6	145 W	35 MB Shared LLC	2	64 GB DDR4	912	290 W
NVIDIA Titan Xp	471	16	1.6	250 W	3MB Shared LLC	1 + 2 (host)	12 GB GDDR5 + 64 GB DDR4	1,383	640 W
Duality Cache	471	22	2.6	148 W	35 MB Shared LLC	2	64 GB DDR4	942	296 W

Table 2. Benchmark server configuration.

	Applications	Dataset	Custom Optimization
Rodinia	backprop, bfs, b+tree, dwt2d, hotspot, hotspot3D, hybridsort, nw, streamcluster	default	None
	gaussian	omp_default	Increased CTA size
	heartwall, leukocyte	default	Loop unrolling
	lud, nn	large (1k, 2k)	CPU hybrid (lud)
acc	divergence, gradient, lapgsrb, laplacian, tricubic, tricubic2, uxx1, vecadd, wave13pt	256 128 1024	Increased CTA size
	gameoflife, gaussblur, matvec, whispering	256 1024	Increased CTA size

Table 3. Evaluated workloads. (acc = OpenACC Benchmark)

	Area (mm^2)	Power (W)	Area Overhead
CPU	456	145	-
Duality Cache Peripheral	3.15	2.96	0.69 %
TMU	5.32	0.06	1.17 %
Controller / FSM	6.16	0.33	1.35 %
MSHR	0.86	0.05	0.19 %
Local Crossbar	0.28	0.01	0.06 %
Total	471.77	148.40	3.5 %

Table 4. *Duality Cache* parameters.

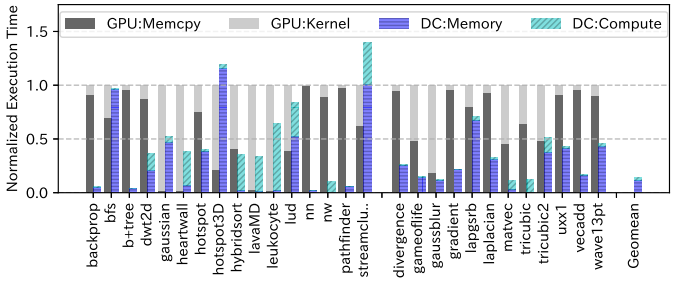


Figure 6. System performance. GPU:GDDR5+memcpy, DC:DDR4.

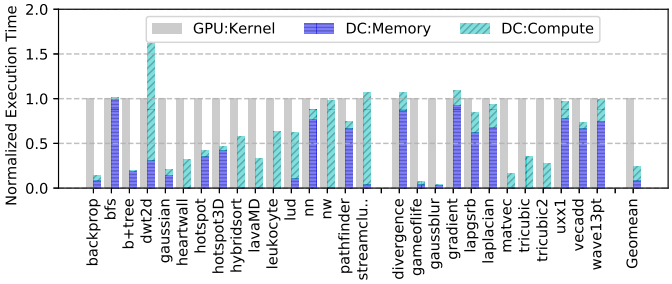


Figure 7. Kernel performance. GPU:GDDR5, DC:GDDR5.

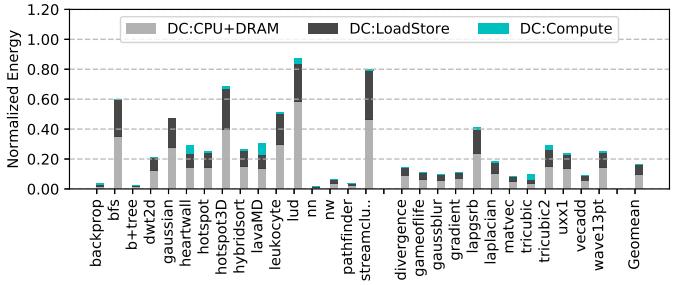


Figure 8. Energy efficiency (system).

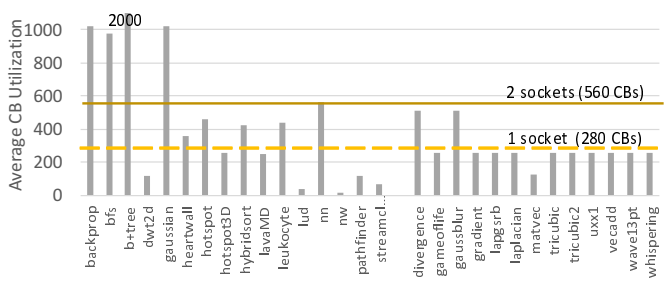


Figure 9. Control Block utilization.

This can be explained by the fact that some applications have very small reuse factor of data, which can make inter-DRAM data movement cost prominent as shown in Figure 6. *Duality Cache* is integrated into the same memory hierarchy as the host, and thus this data movement cost does not exist, resulting in higher performance.

Some CPU models [18] using integrated on-chip GPU could possibly reduce the data movement cost. However, they have typically 10× smaller compute resources than our baseline server GPU (Titan Xp), while taking more than the half of CPU die area. *Duality Cache* is clearly distinguishable from them by the ability to provide the orders of magnitude higher compute resources with only 3.5% of area cost.

B. Massively Parallel Execution: Compute-intensive kernels enjoy *Duality Cache* 's massive parallelism. Figure 9 shows the average number of active Control Blocks (CBs) in the kernels. A CB has 1024 threads and can map several CTAs. Since CBs are independent of each other, this chart indicates the available parallelism of the applications. Each Xeon socket can execute 280 CBs (yellow dash line), thus we have 560 CBs (dark orange line) in total in the baseline dual-socket system. Our GPU has 30 SMs, each can have up to 2 CTAs (Note that this is a register size based calculation; threads in CTAs use GPU cores in a time-multiplexed way).

We can see kernels with a high level of parallelism (e.g. backprop, b+tree, nn, gaussian, gausblur, etc.) significantly reduces execution time in Figure 6 as they can harness *Duality Cache* resources. On the other hand, other benchmarks such as `lud`, `nw` and `streamcluster` have limited parallelism available, resulting in large critical compute time in the kernel

performance. In Section 5.4 we discuss several optimizations we applied to enhance the parallelism beyond the original CUDA programs.

C. Compute / Memory Access Overlap: Some applications show a large compute time portion in the kernel performance, despite enough parallelism (e.g. `lavaMD`). These kernels can successfully hide memory latency with computation. Note few benchmarks show a slowdown (`hotspot3D` and `streamcluster`) with *Duality Cache* because they are memory bandwidth bound. For those, newer memory technology (GDDR5) could help improving performance, as explained shortly (Figure 7),

D. Flexible Cache Allocation: While GPU may underutilize / overutilize its memory bandwidth, *Duality Cache* can adjust parallelism and cache allocation size to balance memory bandwidth (Section 3.5). Many of the evaluated applications benefit from the cache partitioning. By default, we assign the unused Control Block units as cache, but we changed the allocation size based on the applications' behavior. We will discuss it in Section 5.5.

5.3 Performance without Host-Device Transfer

Figure 7 presents kernel execution time for *Duality Cache* and GPU. This experimental setup *eliminates* memcpy time from GPU and provides a GDDR5 memory to both *Duality Cache* and GPU. The goal is to compare the raw compute power of both architectures in a bandwidth neutral fashion. The execution time is normalized to that of GPU. We observe a 1.92× average speedup for Rodinia kernels and 2.39× speedup for OpenACC kernels. This speedup comes at a fraction of area cost of the CPU (3.5%), while the GPU server adds a new die of size 471 mm^2.

5.4 Deep Dive of Applications

Harnessing Full Potential of *Duality Cache* : Applications can fully exploit *Duality Cache* by exposing large parallelism and reusing data. We notice that many CUDA applications are optimized to GPU architectures, being aware of warp size and CTA size. Generally, programmers write a *tiled* program where each tile owns its sub-problem assigned to a CTA. Internally they use for-loops to iterate over the data for the sub-problem, often incrementing induction variable of a thread by warp size (32) to make warps sweep on the data. This creates a dependency between iterations, despite the absence of actual dependency. This is also driven by the fact that CTA size is limited to 1,024 threads due to the maximum register size of an SM. Although our CB can own 1,024 threads, we can expand CTA size beyond this limit provided there is no local communication between threads, as we discussed in section 3. Eliminating local communication is trivial by using atomic operations etc., so we modify some of the source code to unroll the outer for-loops and/or increase the CTA size, as shown in Table 3. OpenACC programs can also easily change the CTA size by setting the vector and worker size option in pragma. This optimization provides significant improvement in performance (e.g. 8.5× for leukocyte and 10.2× for heartwall).

Another important factor is data reuse. Given enough threads to fill CBs and existence of shared data, it is recommended to load the data and reuse it using fixed-length for-loops after launching CTAs to fill CBs. By this, we can avoid multiple fetches of shared data by different CTAs, and also take advantage of thread independent variable isolation.

Fine Interleaving of Serial and Parallel Code Using CPU: Since host-device communication cost is non-trivial for GPUs, CUDA programs tend to incorporate serial or nearly serial code with parallel code. Figure 11(left) illustrates kernel launch patterns of `bfs` and `lud` by showing number of launched CTAs (x-axis) vs. time (y-axis, advances from top to bottom). Ideal truly parallel kernels have a pattern similar to `bfs`, however, as can be seen, `lud` iteratively launches three kinds of kernels, one of which only contains 32 threads. Taking advantage of *Duality Cache* 's tight integration with CPU, we optimize `lud` to execute these small kernels on the host CPU using OpenMP. Figure 11(right) shows the execution time breakdown (normalized to the original version). We observe the optimized version of `lud` achieves a 2.26× speedup. Since the single operation latency of *Duality Cache* is much higher than CPU, we study CPU is more efficient to execute those small kernels. The same idea applies to OpenACC benchmarks as well.

5.5 Impact of Optimizations

Arithmetic Operation Latency: Figure 12 shows average arithmetic operation latency before (base) and after (opt) optimizations we present in Section 3.1. The operation latency is measured using Rodinia benchmarks. Integer multiplication observes the highest reduction in latency (13× better than the baseline). This is because, in many practical cases, integer multiplication is used to calculate address or some variables based on induction variables, and thus contains many leading zeros which we can skip by our optimization.

Floating point addition in many applications has a small dynamic range. The number of unique ediff found usually has its peak at 1 in the distribution (Section 3.1). Overall, optimized fpadd is 6.1× faster than the baseline. The proposed optimizations are not as effective for floating point multiplication and division compared to the correspondent integer operations. This is because the floating point is normalized and has implicit leading 1, which disables the leading zero optimization. However, *Duality Cache* still benefits from skipping iterations of bit 0s.

Cache Allocation: By default, we use unassigned CBs as cache. However, depending on the workload, allocating more cache can improve performance despite sacrificing parallelism. We analyze the source code through static analysis and identify some kernels that can possibly benefit from

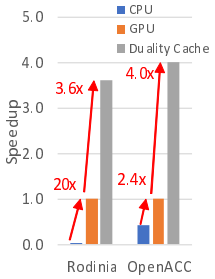
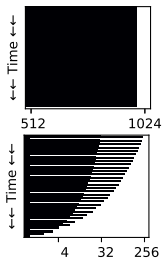


Figure 10. Average speedup.



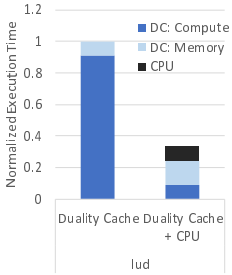


Figure 11. Kernel launch patterns (time vs. #CTAs) of `bfs`(top) and `lud`(bottom) and CPU Hvbrid execution.

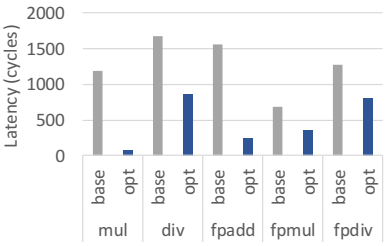


Figure 12. Average operation latency.

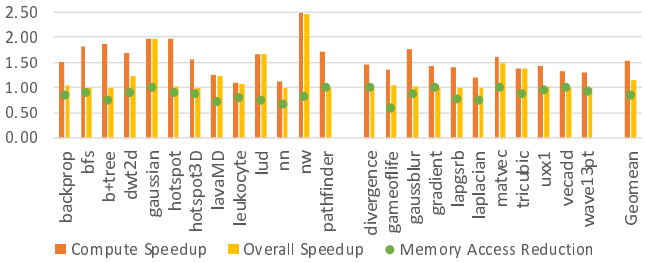


Figure 13. Effect of compiler optimizations.

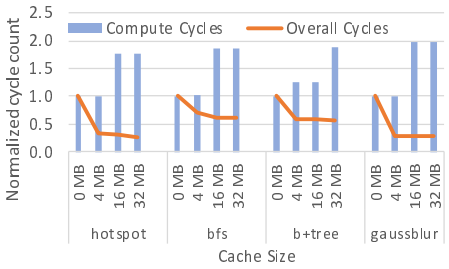


Figure 14. Effect of different cache allocation size.

larger cache size, and adjust the cache allocation. Figure 14 illustrates the system performance of different cache allocation size for some representative applications. As in Figure 9, these applications have a high level of parallelism and can fill more than half of total CBs. The blue bars show compute cycles, and the orange lines present overall performance including memory access. The largest cache size we allocate is 32MB, which is equivalent to the half of the total CBs in our 2-socket baseline. We normalize the cycle count to that of 0-cache configuration. Although augmented cache allocation roughly doubles the computation due to reduced compute units, overall execution cycles decrease substantially because of improved memory performance contributed by the large caches. This optimization provides 3.54× performance improvement on average for applications with a high level of parallelism (CB utilization ≥ 512).

Compiler Optimization: To assess the effect of our compiler optimization, we compare the execution time of applications using two compilers: our compiler and a simple ISA translator. The simple ISA translator replaces PTX with DC-PTX without any scheduling and PTX optimizations. Since kernels cannot use multiple arrays without appropriate handling of operands between arrays, simple ISA translator uses one array per TB. This, on the other hand, provides 4× more available threads, thus each CB maintains 4K threads, each with 8 32-bit registers. For both compilers, we apply our arithmetic operation latency optimizations.

Figure 13 presents the application speedup of our compiler. Compute speedup shows the speedup of the critical computation path, and overall speedup includes memory access latency. The green dots show the reduction in the number of memory access saved by the parallel instruction scheduling. PTX optimizations and our instruction scheduler's efforts to maximize parallelism and to reduce register spilling achieve 1.52× faster computation and 14.3% less memory access. This translates into 1.14× better overall performance.

5.6 Energy

Figure 8 shows the energy breakdown of the benchmarks. This is a system-to-system comparison; GPU includes energy for both memcpy and kernel. *Duality Cache* energy is normalized to the GPU energy and has a breakdown of CPU+DRAM (including memory controller), load/store instructions, and computation in *Duality Cache* . Because of the reduced execution time, we achieve 5.85× energy efficiency compared to GPU system. One core is active during execution to serve instructions. This makes CPU and DRAM access dominant in energy consumption. The only exception is tricubic, one of compute-intensive kernels, where compute energy accounts for 30.7% of total energy consumption.

6 Related Work

To the best of our knowledge, this is the first work that demonstrates the feasibility of in-cache general purpose SIMT computing, leveraging an existing parallel programming framework. In this section, we discuss some of the closely related work.

For decades, processing-in-memory (PIM) has been an attractive idea that has the potential to break the memory wall. PIM solutions move compute near memory [3, 5, 12, 14, 21, 29, 31–33, 41, 42] , and thereby reduce the gap between memory and compute. In contrast, in-memory computing architectures can morph themselves into compute engines

by exploiting the physical properties of the memory array, which makes them intrinsically more efficient then PIM.

In-SRAM computing has been envisioned to provide disruptive technology that can enhance commodity processors with massively parallel compute engines for almost free of cost [2, 10, 19]. On the other hand, in-place DRAM computation faces several challenges to attain true in-place operations. First, since DRAM performs destructive access, in-place computation inevitably corrupts stored data, and thus needs to clone it paying its cost. Also, the small margin for sensing DRAM capacitor makes analog domain computation error prone. Despite several approaches proposed for better DRAM cells, it comes at the cost of non-trivial area overhead (2-3×). Further, the peripheral logic needed to accomplish in-place arithmetic operation is difficult to be integrated with DRAM because of the technology difference. Likewise, various scrambling approaches in address and data employed in commodity DRAM make it challenging to re-purpose it for an in-memory computing device.

Recent works have leveraged compute capability of Non-Volatile Memories (NVMs) such as memristors to perform in-place bit-line computing for domain-specific acceleration [7, 34, 35]. They leverage the dot-product analog computation capability of memristors mainly for machine learning workloads. IMP [15] explores general purpose computing in memristors and proposes a compute stack using TensorFlow frontend. *Duality Cache* supports CUDA programming model, which is widely used for data-parallel applications, is more expressive than TensorFlow, and can be used for applications with irregular memory access patterns.

Floating point is difficult for memristive analog in-memory computing [7, 13, 15, 34], because the resolution of memory cells and ADC is quite limited. It is not realistic to represent a FP value using a single cell. In addition, exponent normalization takes many cycles, which is disadvantageous for memristors with low frequency and low durability. Prior work [13] thus supports fixed-point arithmetic that has the equivalent precision to floating point. They express 64 bit double numbers using 128 memristor cells (1 bit/cell). The extra 64 padding bits are included to normalize and to align numbers with different exponents with respect to a common exponent value of the array. The results are converted to floating points outside the array. In comparison, our bit-serial digital in-SRAM computing supports in-place floating point computation.

While computing in memristors is promising, they remain an emerging technology waiting for large scale production. They are also significantly slower than SRAM and are encumbered with limited endurance.

7 Conclusion

The *Duality Cache* system stack that runs general purpose GPU programs on caches is presented. Enabling in-situ floating point and transcendental functions brings computation capability that can execute SIMT programs. Our compiler introduces optimizations to enhance parallelism and efficiency within the constraints of in-cache computation, and compiles CUDA and OpenACC programs for *Duality Cache* . Our experimental results show the *Duality Cache* architecture improves performance of GPU benchmarks by 3.6× and OpenACC benchmarks by 4.0× over a server class GPU. Repurposing existing caches provides 72.6× better performance for CPU with only 3.5% of area cost.

Acknowledgments

We thank members of M-Bits research group and the anonymous reviewers for their feedback. This work was supported in part by the NSF under the CAREER-1652294 award, the XPS-1628991 award, the SHF-1763918 award and Applications Driving Architectures (ADA) Research Center, a JUMP Center co-sponsored by SRC and DARPA.

References

[1] N. Abeyratne, R. Das, Q. Li, K. Sewell, B. Giridhar, R. G. Dreslinski, D. Blaauw, and T. Mudge. 2013. Scaling towards kilo-core processors with asymmetric high-radix topologies. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. 496–507. https://doi.org/10.1109/HPCA.2013.6522344

[2] S. Aga, S. Jeloka, A. Subramaniyan, S. Narayanasamy, D. Blaauw, and R. Das. 2017. Compute Caches. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 481–492. https://doi.org/10.1109/HPCA.2017.21

[3] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. PIM-enabled Instructions: A Low-overhead, Locality-aware Processing-in-memory Architecture. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*.

[4] Ray Andraka. 1998. A survey of CORDIC algorithms for FPGA based computers. In *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*. ACM, 191–200.

[5] Jay B. Brockman, Shyamkumar Thoziyoor, Shannon K. Kuntz, and Peter M. Kogge. 2004. A Low Cost, Multithreaded Processing-in-memory System. In *Proceedings of the 3rd Workshop on Memory Performance Issues: In Conjunction with the 31st International Symposium on Computer Architecture (WMPI '04)*. ACM, New York, NY, USA, 16–22. https://doi.org/10.1145/1054943.1054946

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[7] Ping Chi, Shuangchen Li, and Cong Xu. 2016. PRIME : A Novel Processing-in-memory Architecture for Neural Network Computation in ReRAM-based Main Memory. In *IEEE International Symposium on Computer Architecture*. IEEE, 27–39. https://doi.org/10.1109/ISCA.2016.13

[8] ROSE compiler infrastructure. 2018. Rose Compiler. http://rosecompiler.org/.

[9] Gregory Frederick Diamos, Andrew Robert Kerr, Sudhakar Yalamanchili, and Nathan Clark. 2010. Ocelot: A Dynamic Optimization Framework for Bulk-synchronous Applications in Heterogeneous Systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT '10)*. ACM, New York, NY, USA, 353–364. https://doi.org/10.1145/1854273.1854318

[10] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaaauw, and R. Das. 2018. Neural Cache: Bit-Serial In-Cache Acceleration of Deep Neural Networks. In *2018 ACM/IEEE 45th Annual*

International Symposium on Computer Architecture (ISCA). 383–396. https://doi.org/10.1109/ISCA.2018.00040

[11] John R. Ellis. 1986. *Bulldog: A Compiler for VLSI Architectures*. MIT Press, Cambridge, MA, USA.

[12] A. Farmahini-Farahani, Jung Ho Ahn, K. Morrow, and Nam Sung Kim. 2015. NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*.

[13] Ben Feinberg, Uday Kumar Reddy Vengalam, Nathan Whitehair, Shibo Wang, and Engin Ipek. 2018. Enabling scientific computing on memristive accelerators. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 367–382.

[14] Basilio B. Fraguela, Jose Renau, Paul Feautrier, David Padua, and Josep Torrellas. 2003. Programming the FlexRAM Parallel Intelligent Memory System. *SIGPLAN Not.* 38, 10 (June 2003), 49–60. https://doi.org/10.1145/966049.781505

[15] Daichi Fujiki, Scott Mahlke, and Reetuparna Das. 2018. In-Memory Data Parallel Processor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/3173162.3173171

[16] Min Huang, Moty Mehalel, Ramesh Arvapalli, and Songnian He. 2013. An Energy Efficient 32-nm 20-MB Shared On-Die L3 Cache for Intel® Xeon® Processor E5 Family. *J. Solid-State Circuits* (2013).

[17] Intel. 2008. x87 and SSE Floating Point Assists in IA-32: Flush-To-Zero (FTZ) and Denormals-Are-Zero (DAZ). https://software.intel.com/en-us/articles/x87-and-sse-floating-point-assists-in-ia-32-flush-to-zero-ftz-and-denormals-are-zero-daz/.

[18] Intel. 2018. Intel Processor Graphics. https://software.intel.com/en-us/articles/intel-graphics-developers-guides.

[19] S. Jeloka, N. B. Akesh, D. Sylvester, and D. Blaauw. 2016. A 28 nm Configurable Memory (TCAM/BCAM/SRAM) Using Push-Rule 6T Bit Cell Enabling Logic-in-Memory. *IEEE Journal of Solid-State Circuits* 51, 4 (April 2016), 1009–1021. https://doi.org/10.1109/JSSC.2016.2515510

[20] Wenhao Jia, Kelly A Shaw, and Margaret Martonosi. 2012. Characterizing and improving the use of demand-fetched caches in GPUs. In *Proceedings of the 26th ACM international conference on Supercomputing*. ACM, 15–24.

[21] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. 2016. Neurocube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory. In *Proceedings of ISCA*, Vol. 43.

[22] Yoongu Kim, Weikun Yang, and Onur Mutlu. 2016. Ramulator: A Fast and Extensible DRAM Simulator. *Computer Architecture Letters* 15, 1 (2016), 45–49.

[23] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. 2010. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. *SIGARCH Comput. Archit. News* 38, 3 (June 2010), 451–460. https://doi.org/10.1145/1816038.1816021

[24] MathWorks. 2018. Compute Square Root Using CORDIC. https://www.mathworks.com/help/fixedpoint/examples/compute-square-root-using-cordic.html.

[25] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. 2009. CACTI 6.0: A tool to model large caches. *HP laboratories* (2009), 22–31.

[26] NVIDIA. 2018. CUDA Toolkit. https://developer.nvidia.com/cuda-toolkit.

[27] NVIDIA. 2018. Parallel Thread Execution ISA. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[28] NVIDIA. 2018. PGI Compilers & Tools. https://www.pgroup.com/.

[29] Mark Oskin, Frederic T Chong, Timothy Sherwood, Mark Oskin, Frederic T Chong, and Timothy Sherwood. 1998. Active Pages: A Computation Model for Intelligent Memory. *ACM SIGARCH Computer Architecture News* 26, 3 (1998), 192–203. https://doi.org/10.1145/279358.279387

[30] PathScale. 2013. Performance test suite for openacc compiler, intel mic, patus and single-core cpu. https://github.com/pathscale/OpenACC-benchmarks.

[31] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. 1997. A case for intelligent RAM. *Micro, IEEE* (1997).

[32] S.H. Pugsley, J. Jestes, Huihui Zhang, R. Balasubramonian, V. Srinivasan, A. Buyuktosunoglu, A. Davis, and Feifei Li. 2014. NDC: Analyzing the impact of 3D-stacked memory+logic devices on MapReduce workloads. In *Performance Analysis of Systems and Software (ISPASS), 2014 IEEE International Symposium on*.

[33] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. [n.d.]. RowClone: Fast and Energy-efficient in-DRAM Bulk Data Copy and Initialization. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*.

[34] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, and Rajeev Balasubramonian. 2016. ISAAC : A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars. *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)* (jun 2016), 14–26. https://doi.org/10.1109/ISCA.2016.12

[35] Linghao Song, Xuehai Qian, Hai Li, and Yiran Chen. 2017. PipeLayer: A Pipelined ReRAM-Based Accelerator for Deep Learning. In *Proceedings - International Symposium on High-Performance Computer Architecture*. 541–552. https://doi.org/10.1109/HPCA.2017.55

[36] Akihiro Tabuchi, Masahiro Nakao, and Mitsuhisa Sato. 2014. A Source-to-Source OpenACC Compiler for CUDA. In *Euro-Par 2013: Parallel Processing Workshops*, Dieter an Mey, Michael Alexander, Paolo Bientinesi, Mario Cannataro, Carsten Clauss, Alexandru Costan, Gabor Kecskemeti, Christine Morin, Laura Ricci, Julio Sahuquillo, Martin Schulz, Vittorio Scarano, Stephen L Scott, and Josef Weidendorfer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 178–187.

[37] J. E. Volder. 1959. The CORDIC Trigonometric Computing Technique. *IRE Transactions on Electronic Computers* EC-8, 3 (Sept 1959), 330–334. https://doi.org/10.1109/TEC.1959.5222693

[38] John S Walther. 1971. A unified algorithm for elementary functions. In *Proceedings of the May 18-20, 1971, spring joint computer conference*. ACM, 379–385.

[39] Xiaolong Xie, Yun Liang, Guangyu Sun, and Deming Chen. 2013. An efficient compiler framework for cache bypassing on GPUs. In *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 516–523. https://doi.org/10.1109/ICCAD.2013.6691165

[40] Q. Xu, H. Jeon, and M. Annavaram. 2014. Graph processing on GPUs: Where are the bottlenecks?. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*. 140–149. https://doi.org/10.1109/IISWC.2014.6983053

[41] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L. Greathouse, Lifan Xu, and Michael Ignatowski. 2014. TOP-PIM: Throughput-oriented Programmable Processing in Memory. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing (HPDC '14)*.

[42] Qiuling Zhu, B. Akin, H.E. Sumbul, F. Sadi, J.C. Hoe, L. Pileggi, and F. Franchetti. 2013. A 3D-stacked logic-in-memory accelerator for application-specific data intensive computing. In *3D Systems Integration Conference (3DIC), 2013 IEEE International*.