Enabling Portable Energy Efficiency with Memory Accelerated Library

Qi Guo, Tze-Meng Low, Nikolaos Alachiotis,
Berkin Akin, Larry Pileggi, James C. Hoe, Franz Franchetti
Carnegie Mellon University
{qguo1, lowt, nalachio, bakin, pileggi, jhoe, franzf}@andrew.cmu.edu

ABSTRACT

Over the last decade, the looming power wall has spurred a flurry of interest in developing heterogeneous systems with hardware accelerators. The questions, then, are what and how accelerators should be designed, and what software support is required. Our accelerator design approach stems from the observation that many efficient and portable software implementations rely on high performance software libraries with well-established application programming interfaces (APIs). We propose the integration of hardware accelerators on 3D-stacked memory that explicitly targets the memory-bounded operations within high performance libraries. The fixed APIs with limited configurability simplify the design of the accelerators, while ensuring that the accelerators have wide applicability. With our software support that automatically converts library APIs to accelerator invocations, an additional advantage of our approach is that library-based legacy code automatically gains the benefit of memory-side accelerators without requiring a reimplementation. On average, the legacy code using our proposed MEmory Accelerated Library (MEALib) improves performance and energy efficiency for individual operations in Intel's Math Kernel Library (MKL) by 38x and 75x, respectively. For a real-world signal processing application that employs Intel MKL, MEALib attains more than 10x better energy efficiency.

Categories and Subject Descriptors

B.7 [**INTEGRATED CIRCUITS**]: Types and Design Styles - *Algorithms implemented in a hardware*; C.1 [**PROCESSOR ARCHITECTURES**]: Other Architecture Styles - *Heterogeneous (hybrid) systems*

Keywords

Energy efficiency, Accelerator, Library, 3D DRAM

MICRO-48, December 05 - 09, 2015, Waikiki, HI, USA
 ACM 978-1-4503-4034-2/15/12...$15.00
DOI: http://dx.doi.org/10.1145/2830772.2830788.

1. INTRODUCTION

The power wall has made hardware accelerators the topic *du jour*. Accelerators, varying from application-specific [1, 2], and domain-specific [3, 4, 5, 6, 7] to general purpose [8, 9], have been proposed to improve energy efficiency. Herein lies a difficult choice an accelerator designer is faced with. On one hand, application-specific accelerators are highly efficient but have limited applicability, whereas general purpose accelerators have greater applicability but may not be efficient. Domain-specific accelerators tread the middle ground but the choice of domain(s) to accelerate for wide-spread applicability still remains.

Our approach toward domain-specific accelerator design is to leverage the fact that high performance software is often written in terms of subroutine calls to highly optimized black-box libraries (e.g., the Basic Linear Algebra Subprograms (BLAS) [10] and Fastest Fourier Transforms in the West (FFTW) [11]) with well-established application programming interfaces (APIs) that represent commonly used operations in their respective domains. By using these APIs, the resulting application code is portable across different platforms, as long as an efficient implementation of the library on the desired platform exists. To demonstrate the benefits gained through the use of such high performance libraries, Figure 1 shows performance gains for benchmarks from R statistical software package [12], PNNL PERFECT [13], and PARSEC [14]. Compared with the original code, the library-based code achieves up to 42x speedup on commodity machines. In a nutshell, the advantages of designing accelerators to these library APIs are 1) the fixed APIs restrict the functionality of the accelerators, resulting in a smaller design space that needs to be explored, 2) as long as widely-used library APIs are chosen, the accelerators automatically gain wide-spread applicability, and 3) legacy code that is implemented using the library APIs automatically benefits from the accelerators.

Although the use of high performance library APIs significantly reduces the accelerator design space, the choice of which library routines are the best candidate for acceleration remains. Generally, high performance library routines can be divided into two sets, i.e., compute-bounded and memory-bounded operations. Compute-bounded operations such as matrix-

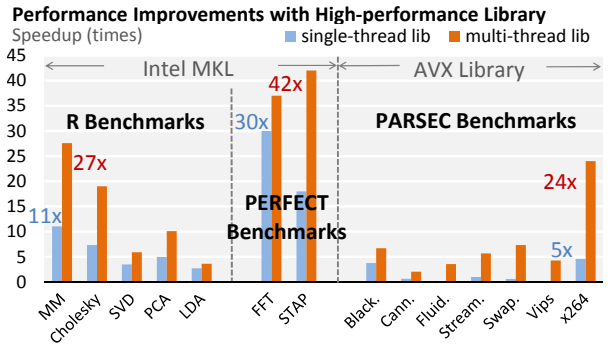


Figure 1: Attained performance gains by using high performance libraries. Intel MKL (Math Kernel Library) [15] is used for accelerating R benchmarks and PNNL PERFECT benchmarks. An AVX-based library [16] is used for accelerating PARSEC benchmarks. The library-based code is up to 27x, 42x and 24x faster for R, PERFECT and PARSEC respectively.

matrix multiply are good candidates for acceleration, as non-computational overheads (e.g., state control and memory accesses) can be greatly reduced with an efficient accelerator. However, memory-bounded operations (e.g., scalar product, general matrix vector multiply, and fast Fourier transform) benefit to a lesser degree on accelerators[1]. The reason is that the latency of a memory access is usually an order (or more) of magnitude larger than that for a floating point computation. Hence, the compute primitives are often stalled, waiting for data to arrive from memory. This is also known as the "memory wall" problem.

Recent technological breakthroughs of heterogeneous 3D die stacking, such as the Hybrid Memory Cube [17], offer high memory bandwidth to address the memory wall. In addition, it allows one to integrate accelerators into the DRAM. This architecture is ideal for accelerating memory-bounded operations, where time and energy are spent performing inefficient data transfer and unnecessary round-trips between the computation units and the memory. As memory-bounded operations are directly mapped to 3D-stacked memory-side accelerators that preserve the library APIs, legacy code can automatically benefit from the increased energy efficiency as long as the necessary software support exists.

To improve the energy efficiency of memory-bounded operations within legacy code, we propose a hardware-software co-design approach. From the hardware perspective, we introduce accelerators on the 3D-stacked memory that address three of the seven domains that are deemed to be important for science and engineering [18]. Our choice of accelerators is further restricted to operations within libraries that are known to be memory-bounded. Each of these accelerators compute one of the memory-bounded operations in the chosen libraries. In addition, by working in tandem with other accelerators, more complicated memory-bounded operations within the chosen libraries are accelerated. From the software perspective, we introduce a source-to-source compiler that converts legacy library calls to underlying runtime routines that invoke accelerators to make our vision of library-based legacy code automatically benefiting from 3D-stacked memory-side accelerators a reality.

[1]For a Haswell system with 112 GFLOPS peak performance (at 3.5 GHz), an integrated "ideal" scalar product accelerator can only achieve 6.4 GFLOPS, since the system only has 25.6 GB/s memory bandwidth.

Contribution. The work in this paper makes the following contributions:

- *Library-driven accelerator design.* We present an accelerator design approach targeting memory-bounded operations in widely used libraries. These accelerators need to be configurable (though in limited) ways to address flexibility built into the library APIs. In addition, these accelerators need to have the capability of being rewired in different configurations so that more complicated operations can be accelerated.
- *Memory accelerated library.* Driven by the proposed approach, we integrate various accelerators into the 3D-stacked DRAM to obtain an energy efficient MEmory Accelerated Library (MEALib), for memory-bounded operations. MEALib has the same interface as the target library operations enabled by the configurable infrastructure and software support.
- *Software support for portability.* To seamlessly allow legacy library-based code to benefit from MEALib, we provide the required software support for the translation of the original library calls to hardware accelerator configuration and control.

Experimental results demonstrate that, on average, legacy code using MEALib attains 38x and 75x better performance and energy efficiency than Intel MKL, respectively. Moreover, for a real-world signal processing application that is highly parallelized and optimized with various MKL routines, MEALib attains more than 10x better energy efficiency than the Intel Haswell.

2. THE MEALIB ARCHITECTURE

Vertical 3D die-stacking with short, fast, and dense Through Silicon Vias (TSVs) allows the stacking of multiple memory dies directly on top of the logic die to achieve high memory bandwidth [19, 20, 21]. Several industrial 3D memory products, including those from Samsung [22] and Micron [17], have emerged. For instance, in the Micron's Hybrid Memory Cube (HMC), multiple DRAM dies are stacked atop a *logic base*, and the logic base contains the *vault controllers* to access vertical banks (vaults), the *link controllers* to communicate with the processor and other stacks, and the *interconnect* between different controllers. It is advantageous to integrate the computation units (e.g., CPUs, GPUs, and accelerators) onto the logic base to exploit the extremely high internal bandwidth, and also reduce the round-trip data transfers between the computation units and the memory.

Due to area and power constraints of the logic base, we introduce a new accelerator layer to the existing HMC system to exploit the high internal bandwidth for accelerating memory-bounded library operations. The

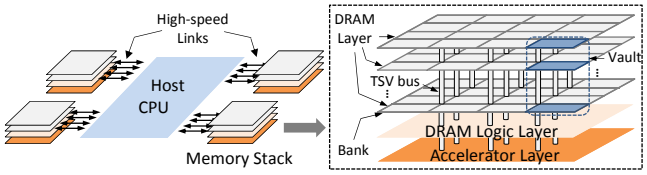


Figure 2: The overall architecture of MEALib hardware.

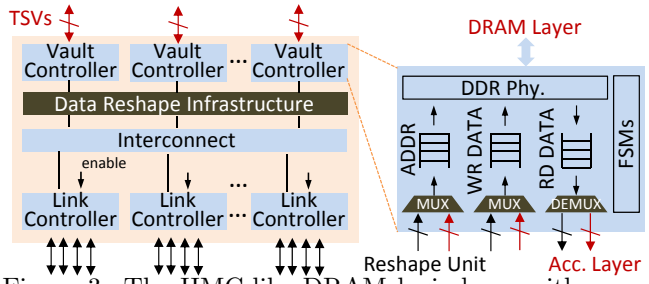


Figure 3: The HMC-like DRAM logic layer with augmented vault/link controllers and data reshape infrastructure.

overall hardware architecture of MEALib is illustrated in Figure 2. The entire system consists of a central host processor and multiple memory stacks, connected via high-speed links used in the HMC system. Each memory stack is comprised of multiple DRAM dies, on top of a traditional DRAM logic layer (i.e., HMC logic base). An accelerator layer is attached below the DRAM logic base, so that it can be easily powered by external power sources.

2.1 DRAM Logic Layer

The DRAM logic layer of MEALib hardware relies on the logic base of the HMC system to access existing infrastructure such as the vault controller, link controller, and interconnect. We augment the vault controller and link controller to incorporate the new accelerator layer. A data reshape infrastructure [23] is also introduced on the DRAM logic layer. Details of the DRAM logic layer are shown in Figure 3.

Vault controller. The vault controller is a memory controller for accessing the associated vault. It contains several queues, which are, the address, write and read queue. For each queue, we add (de)multiplexers to distinguish data access to/from the data reshape infrastructure and the accelerators on the accelerator layer. The vault controller receives requests from the accelerator layer through TSVs.

Link controller. The link controller arbitrates ownership of the DRAM between the CPU and the memory-side accelerators. We assume that the CPU and memory-side accelerators do not operate on the DRAM simultaneously. This simplifies both the hardware and software design spaces. Hence, when the data is processed by accelerators, the accesses from the CPU are blocked by the link controller.

Data reshape infrastructure. Data reshape infrastructure is used for accelerating data layout transforms. Data layout transform (e.g., linear-to-blocked, row-major to column-major, etc.) is crucial for many accelerators, as high performance accelerators often have special requirements with regards to the data layout. For example, the FFT accelerator proposed in [24] requires blocked data in memory. Many general-purpose applications also benefit from dynamic data migration [25, 23]. Since the data reshape infrastructure is a special accelerator that can be employed by both the CPU and the accelerators, we place it on the DRAM logic layer rather than the accelerator layer.

Functions	Description	Accelerator
`cblas_saxpy()`	vector scaling and add	AXPY
`cblas_sdot()`	dot product	DOT
`cblas_sgemv()`	general matrix vector multiply	GEMV
`mkl_scsrgemv()`	sparse matrix vector multiply	SPMV
`dfsInterpolate1D()`	data resampling	RESMP
`fftwf_execute()`	fast Fourier transform	FFT
`mkl_simatcopy()`	matrix transpose	RESHP

Table 1: Illustrative examples on accelerating Intel MKL's memory-bounded operations.

2.2 Accelerator Layer

Hardware accelerators. MEALib focuses on accelerating memory-bounded operations defined in widely used libraries. In our implementation, we target routines within Intel's Math Kernel Library (MKL). The accelerated memory-bounded functions are listed in Table 1. In addition to the Level-1 BLAS[2] (e.g., `AXPY`, `DOT`) and Level-2 BLAS (e.g., `GEMV`) operations, a sparse BLAS operation, i.e., sparse matrix-vector multiplication (`SPMV`), is also accelerated. Data reshape operation (`RESHP`) for matrix transpose, data resampling (`RESMP`), and fast Fourier transform (`FFT`), which are three additional routines in MKL, HPC benchmark suites such as HPC Challenges [26], and the PERFECT Benchmark Suite [13], were also selected for acceleration. Such accelerator cores can be designed manually or generated automatically. While we have listed our choice of accelerators, accelerators for other memory-bounded operations could easily be incorporated using the proposed approach, given available area and power budgets, which are the main determinants of the number of integrated accelerators.

To adapt each accelerator to "distributed" vault controllers on the DRAM logic layer, we propose a tiled architecture where each tile (containing several accelerator cores) is directly connected to the local vault controller. The reasons of placing a group of accelerator cores to a vault controller are: 1) multiple accelerator cores may better utilize the high memory bandwidth, and 2) existing high-performance distributed algorithms can be utilized for designing accelerators. In Figure 4, each vault corresponds to one tile, and these tiles are organized as a traditional mesh network. This network is different from the interconnect at the logic layer, which is mainly used for communication between vault controllers and link controllers. In each tile, there are three

[2]Level-1 BLAS indicates the vector-vector operations, and Level-2 BLAS indicates matrix-vector operations.

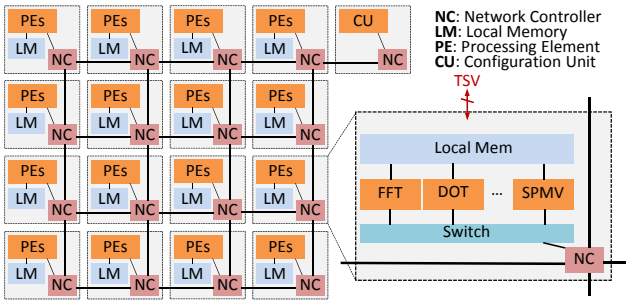


Figure 4: The tiled accelerator layer with configuration infrastructure.

main components, i.e., Local Memory (LM), Network Controller (NC), and Processing Elements (PEs). The LM is shared by accelerators in each tile, and stores data that are read from the DRAM. The NC is used for communication between tiles and vaults. The PEs are a cluster of accelerators (e.g., `FFT`, `DOT`, and `SPMV` etc.) connected via a switch for configuration purpose.

Configuration infrastructure. Since accelerators trade generality for efficiency, the flexibility and configurability of the accelerator is a critical design consideration. As our accelerators target libraries with well-defined APIs, the configurability of accelerators is determined by the parameters of APIs, such as, the problem size, the input/output buffer, and the access stride. Flexibility is required for the chaining of accelerators for the computation of more complicated operations. For instance, a Synthetic Aperture Radar (SAR) image formation algorithm requires both `FFT` and `RESMP` accelerators [27]. Both flexibility and configurability are offered by a carefully designed configuration infrastructure. Shown in Figure 4, the configuration infrastructure consists of distributed switch networks on each tile and a centralized configuration unit (CU). The CU also communicates with other tiles through the NC.

The detailed architecture of CU is shown in Figure 5. It comprises the Fetch Unit (FU), Instruction Memory (IMEM), and Decode Unit (DU). The host processor starts by storing the *accelerator descriptor* (which is essentially the hardware/software interface and will be detailed in Section 2.3) in a pre-allocated memory space. When this happens, the FU transfers the entire descriptor to the IMEM and activates the DU. The DU parses the descriptor sequentially until the end of a pass (which contains a series of processing instructions that describe a datapath representing a complete processing operation) is detected. For every instruction in the pass, the DU activates the corresponding accelerator and appropriately configures the switch logic of each tile at the input and output ports. When all accelerators in a pass are activated, the DU enables an accelerator initialization process during which each accelerator retrieves accelerator-specific configuration data from the main memory. When all accelerators in the current pass are configured, the DU initiates the processing for all tiles. During processing, the first accelerator in the pass fetches input data from the main memory while the last accelerator stores the output data back to the main memory. The DU monitors the status of the last accelerator in the pass to detect when the pass processing is over in order to proceed with the next pass in the descriptor.

Figure 5: Details of the configuration unit.

2.3 Accelerator Descriptor in DRAM

The accelerator descriptor is a physically contiguous memory region in DRAM that serves to: 1) control accelerators from software, and 2) configure accelerators with the required parameters of the targeted library routine.

The accelerator descriptor is divided into three regions, i.e., *Control Region* (CR), *Instruction Region* (IR), and *Parameter Region* (PR). The CR mainly contains the control command such as `START` and the number of instructions in the IR. Each instruction in the IR is either an *accelerator* or a *control* instruction. The accelerator instruction corresponds to one accelerator invocation, while the control instruction is related to a control flow operation such as `LOOP`. The accelerator instruction has three fields, i.e., opcode, parameter size and address. The opcode specifies which accelerator to use, while the other two fields determine the size and starting address of accelerator parameters, which are determined by the targeted library APIs. All required parameters of an accelerator are stored in the PR.

3. PROGRAMMING MEALIB

The main objective of MEALib is the acceleration of memory-bounded libraries, and a key requirement is that legacy programs written with well-defined libraries and OpenMP directives directly improve performance and energy efficiency without a reimplementation.

3.1 Legacy Code using MEALib

Listing 1 shows a code section from the STAP (Space-Time Adaptive Processing) program [28], an important application in radar systems, that uses Intel's MKL and OpenMP directives to achieve high performance and energy efficiency. This example contains four main parts, i.e., data allocation (e.g., `malloc`), data copy with FFTW guru interface (e.g., `fftwf_plan_guru_dft`), a batched FFT operation (i.e., multiple parallel FFTs), and multiple MKL library calls (e.g., `cblas_cdotc_sub`) within `for` loops annotated with OpenMP directive. To execute this code using MEALib, the following challenges must be addressed.

1. A shared memory between the CPU and accelerators must be supported, to avoid costly data transfer between CPU and accelerators. In this program, both the CPU and accelerators (more precisely, libraries) should access `datacube` directly without any explicit data copy.
2. Requirements on data allocations imposed by the accelerators must be hidden from the programmer.

For instance, our FFT accelerator requires data to be stored in a physically contiguous region. This means that datacube_pulse_major_padded should be placed into a special region that the FFT accelerator can efficiently process.

3. An efficient mapping from library routine to accelerator is necessary, as more complicated library routines may require the use of more than one accelerator. In this program, although both the data copy and FFT operations use the same library call (e.g., fftwf_plan_guru_dft), they should be mapped to two different accelerators (RESHP and FFT).
4. Multiple parallel library calls (typically with OpenMP directives) must be mapped efficiently into a small number of accelerator descriptors to reduce the cost of accelerator invocations. In this example, there are in total 16M independent function calls of cblas_cdotc_sub. Dedicating an individual descriptor to a library call would result in a tremendous accelerator invocation cost, diminishing the benefits of accelerator execution.

```
// data allocation
datacube =
  malloc(sizeof(complex) * num_datacube_elements);

datacube_pulse_major_padded =
  malloc(sizeof(complex) *
  num_datacube_pulse_major_padded_elements);
... ...
// data copy
plan_ct = fftwf_plan_guru_dft(0, NULL,
            3, howmany_dims_ct,
            datacube,
            datacube_pulse_major_padded,
            FFTW_FORWARD, FFTW_WISDOM_ONLY);
// FFT operation
plan_fft = fftwf_plan_guru_dft(1, dims,
            2, howmany_dims,
            datacube_pulse_major_padded,
            datacube_doppler_major,
            FFTW_FORWARD, FFTW_WISDOM_ONLY);
fftwf_execute(plan_ct);
fftwf_execute(plan_fft);
... ...
// multiple parallel inner products
#pragma omp parallel for num_threads(4)
private(dop, block, sv, cell)
for (dop = 0; dop < N_DOP; ++dop)
  for (block = 0; block < N_BLOCKS; ++block)
    for (sv = 0; sv < N_STEERING; ++sv)
      for (cell = 0; cell < TBS; ++cell)
        cblas_cdotc_sub(TDOF*N_CHAN,
          &adaptive_weights[dop][block][sv][0], 1,
          &snapshots[dop][block][cell], TBS,
          &prods[dop][block][sv][cell]);
```

Listing 1: Code from the STAP program.

3.2 The Underlying Support

The first challenge can be partially addressed by a unified address space between the CPU and accelerators. Moreover, a shared memory management mechanism is provided to allow the CPU to directly access data processed by accelerators. For the second challenge, we designed *memory management runtime routines* for memory allocation/free in a physically contiguous memory region, to substitute the standard allocation/free functions. To support automatic substitution, a source-to-source compiler is implemented. The compiler is also used to parse the library calls and convert them to the appropriate *accelerator control runtime routines*. The compiler, along with the accelerator control runtime routines, also address the third and the fourth challenges as mentioned.

The overview of our solution is shown in Figure 6. The library calls (possibly with OpenMP directives) are first identified with our source-to-source compiler. The compiler translates them to the corresponding accelerator control runtime routines for generating the accelerator descriptor. After identifying the library calls, the declaration of their input/output data, and thus the related allocation/free functions can also be identified by the compiler. The compiler translates them to the corresponding memory management runtime routines. The entire runtime routines, including the memory management and accelerator control routines, are built on a shared memory management mechanism.

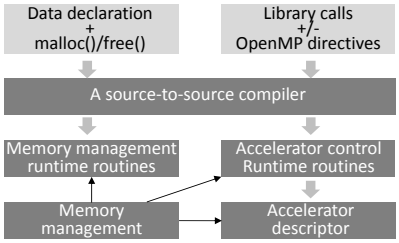


Figure 6: The overview of underlying software support to enable MEALib portability.

3.3 Shared Memory Management

Recall that our memory-side accelerators are directly integrated into the memory stack. This implies that the accelerators share the same memory with the host processor. Therefore, it is natural that both accelerators and the processor share the same memory address space, i.e., a unified physical address space.

To simplify hardware design and avoid non-trivial performance penalty for supporting virtual memory, our accelerators do not include the memory management unit (MMU), which means that accelerators only access memory via *physical addressing*. Moreover, accelerators may also require physically contiguous space for efficient processing. In contrast, legacy applications have been implemented using *virtual addressing* and are not aware of the physical memory layout. In order to allow legacy code to automatically benefit from the proposed system, this requires us to bridge this difference in memory adressing. We achieved this by mapping the reserved physically contiguous memory, which is managed by the memory management runtime routines, to virtual address space. As a result, the memory region can be either accessed by the accelerators via physical addressing or by the processor via virtual addressing.

For a given accelerator, the memory stack on which it is integrated, is viewed as the *Local Memory Stack* (LMS), while the remaining memory stacks are considered as *Remote Memory Stacks* (RMS). The data processed by the accelerator should reside in its LMS to

better utilize bandwidth. As the physical space of LMS is the natural interface between the CPU and accelerators for communication, we further divide the LMS into a *command* space and a *data* space. The command space primarily stores the *accelerator descriptor* as introduced in Section 2.3. The CR (i.e., Control Region) in the command space is monitored by the hardware. Once the `START` command is written to the CR, the configuration infrastructure is invoked to process instructions in the IR (i.e., Instruction Region). It also retrieves the parameters of each instruction that are stored in the PR (i.e., Parameter Region).

Address translation. On the CPU side, we implement a device driver to allow manipulation of the physical space. Once the device driver is installed, the command space is allocated in the physical space, and then it is mapped to the virtual space via the `mmap` system call. Thus, the associated accelerators can be directly controlled and configured by writing the accelerator descriptor to the mapped virtual space. The data space is also allocated/freed through the device driver. Specifically, the device driver provides the `ioctl` system call to process memory allocation/free requests from the runtime routines. Then, customized `mmap` is implemented in the device driver to map the allocated contiguous physical memory to virtual memory space. As accelerators use physical addresses as inputs, the CPU performs the virtual-to-physical translation when specifying the input/output addresses for each accelerator.

Figure 7 shows details of the shared memory management. After allocating memory from the mapped virtual space, the CPU initializes the allocated region with user provided data (Step 1). Hereafter, the CPU prepares the accelerator descriptor with translated physical addresses, and stores it to the mapped command space (Step 2). Finally, the accelerator accesses the data from the specified region and start processing (Step 3).

3.4 Source-to-Source Compiler

A source-to-source compiler is crucial for portable energy efficiency using MEALib. It is built to recognize library calls (possibly annotated with OpenMP directives) that can be accelerated using our memory-side accelerators. The associated memory allocation/free functions are also translated into MEALib runtime routines. At the heart of the translation is a Task Description Language, which is used to describe sequences of accelerator invocations and their configurations.

Task description language. The *accelerator descriptor* is a high level description of the computation to be performed by the accelerators, and is described in our so-called Task Description Language (TDL). The TDL consists of three basic blocks, i.e., `COMP`, `PASS`, and `LOOP`. The `COMP` block corresponds to the invocation of a single accelerator. It describes the accelerator to be invoked, and the location (file) where the parameters of the accelerator can be found. Multiple `COMP` blocks compose a `PASS` block, and each `PASS` block has its own input and output data buffers. The `LOOP` block indicates that the included `PASS` block(s) can be executed

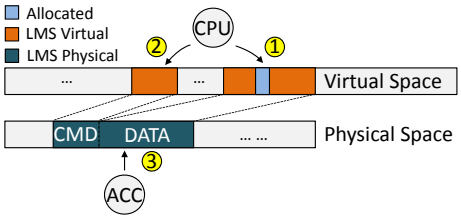


Figure 7: The unified physical address space, where the CPU uses the virtual address and the accelerators use the physical address.

multiple times. In MEALib, the TDL string and the related parameter files are automatically generated by our compiler.

The compiler works in two passes. In the first pass, it identifies the library calls, determines their input/output data, and builds a high level description of the accelerator descriptor, using the TDL. In the second pass, the related allocation/free functions are identified and translated to runtime routines, which will be discussed subsequently.

Pass 1: Library call identification. Library calls are translated to accelerator control runtime routines. The TDL string that describes the desired configuration for each invocation of the accelerator(s) is also generated during this pass. Moreover, the information of the input and output buffers is also identified by the compiler. The rest of the API parameters are stored into a parameter file, which is a part of the TDL string. An optimization is performed when an accelerated library call is immediately followed by another accelerated library call such that the input of the second is the output of the first. These two library calls are then chained together, and their TDL strings are then merged into one. In Listing 1, data copy and FFT operations can be chained. The two pairs of `fftwf_plan_guru_dft` and `fftwf_execute` are translated to a single TDL string with a single `PASS` block containing the `RESHP` and `FFT` accelerator invocation details. The parameters of the accelerators are stored in two files, `reshape.para` and `fft.para`, corresponding to the two invocations respectively.

Multiple library calls within the OpenMP `for` loop are handled by the compiler. The compiler recognizes the initial values, the end conditions, and the steps of each loop, to determine the total number of loop iterations, input stride, and output stride. Based on these information, the compiler generates a TDL string with a `LOOP` block, which translates to a single accelerator descriptor. Compared with multiple TDL strings, each of which contains only one `PASS` block, the `LOOP`-based TDL string can significantly reduce the cost of accelerator invocation. In this concrete example, more than 16M function calls of `cblas_cdotc_sub` are finally translated into only one accelerator invocation.

Pass 2: Memory allocation/free transformation. Recall that the data that are processed by the accelerator need to be in physically contiguous space for efficient processing. This implies that there is a need to identify the memory region used by the accelerators,

and to allocate them in a contiguous physical space. During this pass, `malloc` and `free` calls, with no guarantees that physical space is contiguous, are replaced with our customized memory management runtime routines, which provides such guarantees.

3.5 Runtime Routines

MEALIB has two kinds of runtime routines, memory management runtime and accelerator control runtime, to operate on the *data* and *command* space, respectively. The program using MEALIB should be linked with this runtime library to gain acceleration.

Memory management routines. The memory management routines of MEALIB are used for allocating (i.e., `mealib_mem_alloc`) and freeing (i.e., `mealib_mem_free`) data in the data space. The memory stack used for allocation can also be explicitly specified during memory allocation. Since our compiler automatically replaces standard allocation/free functions in the program with MEALIB routines, this additional parameter is determined by the compiler as well.

```
// generate the accelerator descriptor
acc_plan mealib_acc_plan(const char *tdl,
                         void *in_addr, long in_size,
                         void *out_addr, long out_size);
// invoke accelerators
void mealib_acc_execute(acc_plan p);
// destroy the accelerator descriptor
void mealib_acc_destroy(acc_plan p);
```

Listing 2: Accelerator control routines.

Accelerator control routines. The accelerator control routines are used for configuring and controlling accelerators with the accelerator descriptor. As shown in Listing 2, the routine `mealib_acc_plan` takes the TDL string, the input/output buffer addresses and sizes as inputs, to generate the `acc_plan` data structure, which is essentially the accelerator descriptor. The accelerator descriptor can be reused to invoke the same accelerator(s) with the same configuration multiple times through `mealib_acc_execute`. Finally, the accelerator descriptor is destroyed using `mealib_acc_destroy`.

In `mealib_acc_execute`, prior to invoking the accelerators, the host CPU guarantees that input data of memory-side accelerators is up-to-date. In contrast to using *nontemporal instructions* to operate on uncachable regions as in [29], the entire memory in MEALIB can still use the traditional hardware cache coherence, and the data coherence is enforced by using the `wbinvd` instruction to write back the modified cache lines to main memory before invoking the accelerators. Moreover, as the CPU and accelerators do not operate on the same memory region simultaneously, it is not necessary to modify existing memory consistency protocols.

4. EVALUATION METHODOLOGY

We propose a hybrid evaluation methodology that combines native execution and simulation to obtain an accurate approximation of the end-to-end execution on the proposed system. We evaluate MEALIB by running programs written with MKL APIs on a multicore system (e.g., Intel Haswell). The multicore CPU is treated as the central host, while one memory DIMM is used for mimicking the 3D memory stack with accelerators. The behavior of this stack is simulated and modeled by several tools. The performance and power of the remaining DIMMs and the central processor are measured through actual execution. When the processor calls the library with aforementioned accelerator descriptor, the simulation is triggered to get the power and performance estimation of the invoked accelerators. The overall performance and power are estimated by aggregating the simulated and the measured results. As shown in Figure 8, the evaluated system is used to characterize a multicore system that consists of both conventional DDR and 3D-stacked DRAM, similar to Intel Knights Landing's flat addressing mode having both "near" and "far" memory [30].

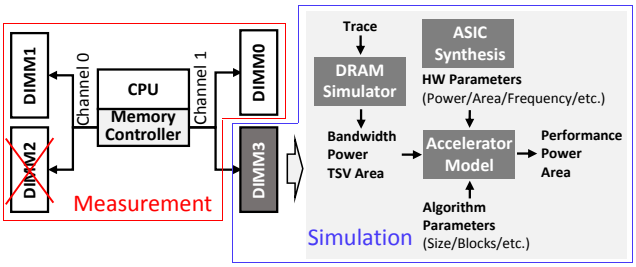


Figure 8: Overview of the evaluation methodology.

4.1 System Configuration and Measurement

A key challenge of using the DIMM to simulate a memory stack is that the channel-interleaving mode, where one physical page is equally distributed across different channels in cache-block granularity, is automatically enabled on modern systems. Thus, the system cannot guarantee that physical address is contiguous within one DIMM. For simulation purposes, we remove DIMM2 in Figure 8 from the motherboard to convert the channel-interleaving mode to the asymmetric mode [31], where the memory zone with high address is in single-channel mode. This disables the memory channel interleaving on DIMM3. In our experimental set-up, the address spaces of DIMM0 and DIMM1 are still interleaved, while the address space of DIMM3 is separate from others. Hence, DIMM3 is used to simulate the local memory stack of all accelerators.

We use PAPI [32] to read the hardware performance counters for execution time, and Running Average Power Limit (RAPL) [33] for the power consumption of the processor and DRAM. The measurement excludes the pure simulation process, but includes the overhead of memory management such as memory allocation, accelerator control, and cache flushing.

4.2 Simulation and Modeling Methodology

A single simulation or modeling tool is not able to obtain the performance/power/area of the accelerated DRAM stack. As shown in Figure 8, we first generate memory traces from accelerators, and treat them as in-

Functions	Data Set	Accelerator
`cblas_saxpy()`	256M vector (1GB)	AXPY
`cblas_sdot()`	256M vector (1GB)	DOT
`cblas_sgemv()`	16384 × 16384 matrix (1GB)	GEMV
`mkl_scsrgemv()`	*rgg_20* from UF SMC [36]	SPMV
`dfsInterpolate1D()`	16384 blocks	RESMP
`fftwf_execute()`	8192 × 8192 matrix (512MB)	FFT
`mkl_simatcopy()`	16384 × 16384 matrix (1GB)	RESHP

Table 2: Data sets of the accelerated functions.

Platforms	Cores	Bandwidth
Haswell i7-4770K	4-core @ 3.5 GHz	25.6 GB/s
Xeon Phi 5110P	60-core @ 1.0 GHz	320 GB/s
PSAS	-	25.6 GB/s
MSAS	-	102.4 GB/s
MEALib hardware	4-core @ 3.5 GHz	510 GB/s

Table 3: Hardware platforms for comparison.

puts for an in-house cycle-accurate 3D-stacked DRAM simulator (where the basic parameters of 3D-stacked DRAM are obtained from CACTI-3DD [34]) to obtain the achieved memory bandwidth and power. Meanwhile, we use Synopsis Design Compiler with 32nm library for synthesized power and area of logic and floating-point units. The results from the above tools and algorithmic parameters (e.g., problem size and tile size) are used as inputs for customized analytical models for each accelerator to produce the related performance, power, and area. Using analytical models facilitates fast design space exploration over a large number of parameters, and is inspired by the work in [35, 27, 24]. The total energy and execution time of the host processor and accelerators are accumulated to compute the overall power consumption.

4.3 Benchmarks, Platforms, and Baseline

To evaluate the efficiency of MEALib, we wrote synthesized programs with standard Intel MKL 11.2 APIs, and the related input data sets of evaluated functions are shown in Table 2. We ran the same code on different platforms, i.e., the Intel i7-4770k 4-core processor, the Intel Xeon Phi 60-core coprocessor, and our MEALib architecture to demonstrate the portability of our approach. In addition, we compare our MEALib architecture with other acceleration alternatives, including the *Processor-Side Accelerated System* (PSAS), where the accelerators share the same memory hierarchy with the central processor, and the *2D Memory-Side Accelerated System* (MSAS), where the accelerators sit atop the conventional DRAM [29]. The configurations of these platforms are shown in Table 3.

In addition to evaluating individual library functions, we compared the performance of a real-world application, i.e., STAP (Space Time Adaptive Processing) from the PNNL PERFECT suite [13], on our MEALib architecture, with its execution on the Intel i7-4770k processor. To improve the quality of our baseline, the STAP code is highly optimized with MKL library and multithreading technique with OpenMP, and the involved 5 library functions (in order) are listed in Table 4. Compared with its original implementation, the optimized baseline is 42x and 787x better in terms of performance and energy-delay product (EDP) [37], respectively.

Functions	Purpose	Type
`fftwf_execute()`	data copy, FFT	memory-bounded
`cblas_cherk()`	rank-k matrix update	compute-bounded
`cblas_ctrsm()`	triangular matrix solver	compute-bounded
`cblas_cdotc_sub()`	inner production	memory-bounded
`cblas_saxpy()`	vector scaling	memory-bounded

Table 4: Library functions used in STAP.

5. EXPERIMENTAL RESULTS

We first present overall efficiency, power, and area of MEALib. Then, we study the design space of two accelerators. After that, we evaluate efficiency of proposed configuration infrastructure. Finally, we use MEALib to accelerate the STAP application.

5.1 Overall Efficiency

Performance. We compared the performance and energy efficiency of MEALib with the other four platforms, i.e., Haswell, Xeon Phi, PSAS, and MSAS. Figure 9 shows the comparison on performance in terms of GFLOPS (giga floating-point operations per second)[3], where all performance results are normalized to MKL performance on the Haswell. The first observation is that, with the evaluated version of MKL, Xeon Phi (with 32 threads) cannot significantly outperform Haswell (with 4 threads). For `AXPY`, Xeon Phi achieves the best improvement over Haswell, i.e., 2.23x, while for `RESHP`, the performance of Xeon Phi is only 2.4% of that of the Haswell. A possible reason is that the data sets might not be large enough to exploit a large number of hardware threads. The second observation is that on average PSAS and MSAS are 2.51x and 10.32x better than the Haswell, respectively, which demonstrates the effectiveness of specialized accelerators. Finally, MEALib achieves the best performance on all the evaluated operations, and the improvements range from 11x (`SPMV`) to 88x (`RESHP`). On average, MEALib achieves 38x, 15x, and 3.7x better performance over the Haswell, PSAS, and MSAS, respectively.

Energy efficiency. As energy efficiency is an important design concern of computer systems, we further compare the energy efficiency in terms of GFLOPS/W in Figure 10. The first observation is that the energy efficiency of Xeon Phi further decreases when comparing with the Haswell. The reason is that the power consumption of Xeon Phi is typically larger than that of Haswell (e.g., 130W vs. 48W for the FFT operation). The second observation is that the energy efficiency gains of MEALib are much larger than the performance gains shown in Figure 9, since the MEALib has relatively smaller power consumption compared with other platforms. For example, when computing FFT, the

[3]Note that for `RESHP`, there is no floating point operation, and here the performance is measured by GB/s.

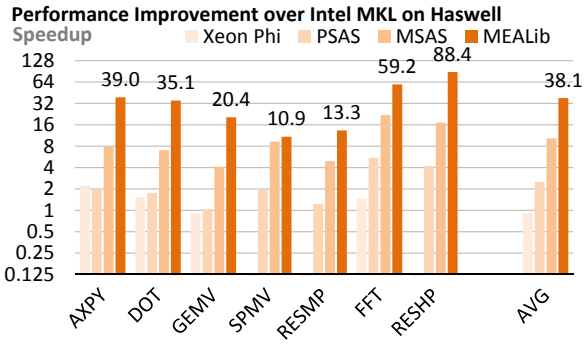


Figure 9: Performance improvement over the standard MKL on the Haswell machine.

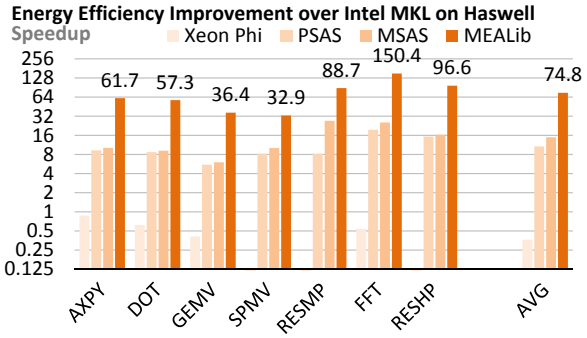


Figure 10: Energy efficiency improvement over the standard MKL on the Haswell machine.

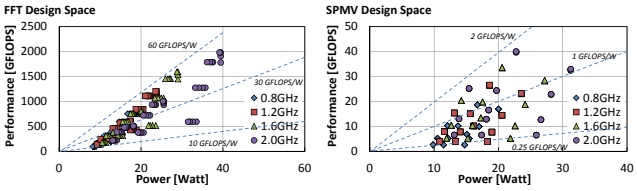
power consumption of MEALib is only 19W, while the power consumptions of Haswell, Xeon Phi, and MSAS are 48W, 130W, and 41W, respectively. On average, MEALib achieves 75x, 7x, and 5x energy efficiency gains over the Haswell, PSAS, and MSAS, respectively.

5.2 Power and Area Analysis

Table 5 lists the power consumption and area of main components (including accelerators, NoC, and TSVs) on the accelerator layer. For primitive accelerators, the power consumption includes both the accelerator and the 3D DRAM power (where the power of TSVs is also included). Since the accelerators are designed to maximally exploit the available memory bandwidth (i.e., 510GB/s), multiple primitive accelerators cannot be activated simultaneously. As the configuration infrastructure is only used before the deployment of accelerators, it does not factor in when the total power consumption is computed. We only need to consider the primitive accelerator with highest power consumption (i.e., `GEMV` with 23.75W power) and the NoC power, leading to total power consumption as 23.85W. We assume that the total area of the accelerator layer is 68mm^2, the same as the DRAM area reported for HMC 2011 [17]. The entire area of all these components is 41.77mm^2, 61% of the available area of the accelerator layer. Compared with modern server chips that are always more than 400mm^2 in size [38], the total area of proposed hardware in MEALib is relatively small, and more domain-specific, memory-bounded libraries can be accelerated with more area budget. It is notable that `RESHP` lies on

Component	Power(W)	Area(mm^2)	Area(%)
AXPY	23.56	1.38	(2.03%)
DOT	23.49	1.81	(2.66%)
GEMV	23.75	2.45	(3.60%)
SPMV	15.44	14.17	(20.84%)
RESMP	8.19	2.64	(3.88%)
FFT	18.89	16.13	(23.72%)
RESHP	22.70	-	-
NoC (router + link)	0.095	1.44	(2.12%)
TSVs	-	1.75	(2.57%)
Total	**23.85**	**41.77**	**(61.43%)**

Table 5: Estimated power and area (32nm) for components on the accelerator layer.



(a) FFT Design Space (b) SPMV Design Space

Figure 11: Design space analysis of the `FFT` and `SPMV` accelerator. The performance and energy efficiency also depend on the architectural design of accelerators.

the DRAM logic layer, and the reported power is for a matrix transpose operation that already includes the DRAM power.

The additional logic at the DRAM logic layer mainly contains the MUX and data reshape unit. The total power of these components is 0.25W, and the total area is 0.45mm^2, which is only 0.66% of the entire logic layer.

5.3 Design Space Analysis

The energy efficiency gain of MEALib is highly dependent on the available bandwidth, and the architectural design of accelerators. Power constraints may cause changes to the available bandwidth, which may result in different accelerator designs.

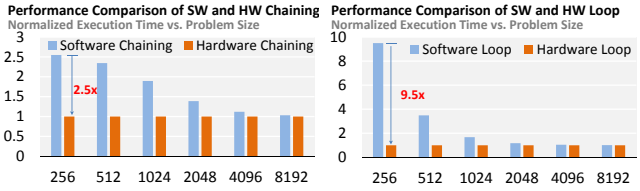
We illustrate the design space analysis with two accelerators, i.e., `FFT` and `SPMV`. Given a memory bandwidth of 510GB/s, we explored various design parameters, such as accelerator frequency, row buffer size, number of accelerator cores, and block size. Figure 11 shows the performance (GFLOPS) and power (W) of these accelerators in the explored design space. For the `FFT` accelerator, energy efficiency varies from 10 to 56 GFLOPS/W, depending on power constraints. For the `SPMV` accelerator, the energy efficiency difference is even larger for different design options, varying from 0.18 to 1.76 GFLOPS/W.

5.4 Configuration Efficiency

We use *accelerator chaining* and *accelerator loop* to demonstrate the configuration efficiency offered by the proposed configuration infrastructure. For accelerator chaining, we compare the chaining of `RESMP` and `FFT` accelerators (hardware-based chaining) for the SAR (Synthetic Aperture Radar) application [27] against sepa-

rate invocations (software-based chaining). The comparison results for different problem sizes are shown in Figure 12a. Given an input image of 256 × 256 pixels, hardware-based chaining reduces the execution time by a factor of 2.5x. As the input size increases, the performance gap between software-based and hardware-based chaining decreases. However, as there are many images to process in real-world scenarios, the aggregated performance gain of hardware-based accelerator chaining is expected to be significant.

For *accelerator loop*, we compare using the `LOOP` block of TDL to invoke the `FFT` accelerator for 128 times within one accelerator descriptor (hardware-loop) with a `FFT` invocation embedded within a `for` loop with 128 iterations (software-loop). The results are shown in Figure 12b. Hardware-based loop reduces the execution time by a factor of 9.5x for a problem size of 256×256. Although the performance gain over software-based loop decreases along with the increasing problem size, common applications (e.g., STAP) with intensive loops still greatly benefit from the hardware-based loop.

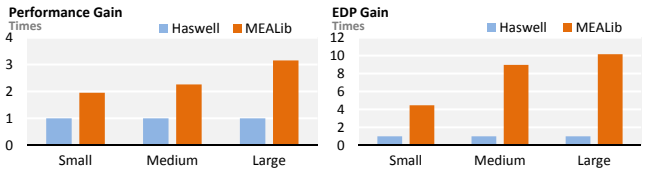


(a) Chaining (b) Loop

Figure 12: Efficiency of the proposed configuration infrastructure. (a) Comparison of software- and hardware-based accelerator chaining. (b) Comparison of software- and hardware-based accelerator loop.

5.5 Accelerating STAP

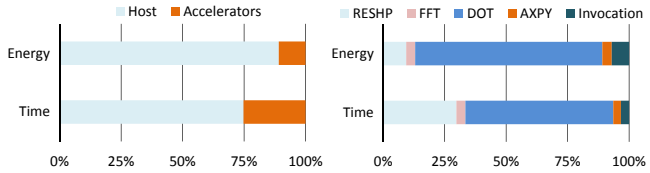
For the STAP application, as shown in Table 4, the compute-bounded functions (i.e., `cherk` and `ctrsm`) are executed by the central multicore processor, while the rest memory-bounded functions (i.e., `fftwf_execute`, `cdotc`, and `saxpy`) are executed by accelerators. After compiling the code with our source-to-source compiler, the translated code is linked with MEALib runtime routines, and eventually related accelerators including `RESHP`, `FFT`, `DOT`, and `AXPY` are invoked.



(a) Performance Gain (b) EDP Gain

Figure 13: Achieved (a) performance and (b) energy efficiency gains of STAP using MEALib over its execution on the Haswell. Three different data sets are used for evaluation.

Figure 13 shows the performance and energy efficiency (energy-delay product [37]) gains of MEALib over the Haswell on different data sets (small, medium, and large). We can see that compared against the optimized library-based multi-threaded STAP, MEALib improves the performance by a factor of 2.0x, 2.3x, and 3.2x for small, medium, and large data set, respectively. Regarding the energy efficiency, MEALib improves the EDP by a factor of 4.5x, 9.0x, and 10.2x for small, medium, and large data set, respectively. It is expected that the benefit of MEALib is more significant as the data set size continues to increase.



(a) Host and Accelerators (b) Different Accelerators

Figure 14: The execution time and energy breakdown of the host, accelerators, and invocations. (a) The breakdown of the host and accelerators. (b) The breakdown of different accelerators and invocations.

We further show the execution time and energy breakdown of MEALib in Figure 14. In Figure 14a, we can see that the execution on the host multicore processor is about 75% of the entire execution time, while the energy consumption is about 90% of the total energy, which also demonstrates the energy efficiency of deployed accelerators. Figure 14b further shows the breakdown of deployed accelerators, where the `DOT` accelerator is the dominated one. The execution time and energy of `DOT` are about 60% and 76% of the total time and energy, respectively. The `AXPY` accelerator consumes the least time and energy, which are only 3.1% and 3.8% of the total time and energy, respectively.

We also show the cost of accelerator invocation, which mainly includes the costs of cache flushing and data copy of the accelerator descriptor. Thanks to the proposed configuration infrastructure, we are able to compact in total about 17M library calls (cf. Listing 1) into only 3 accelerator descriptors. The resulting accelerator invocations only consume 3.3% and 7.1% of the total accelerator time and energy, respectively.

6. RELATED WORK

Portable libraries. There exists many efficient, portable software libraries for various computing domains. In addition to the well-known MKL, Intel also develops another library, Integrated Performance Primitives (IPP), for multimedia and data processing applications [39]. Open source portable libraries are also very active in different domains. GotoBLAS [40] is widely used in high performance computing. SPARSITY [41] is for sparse matrix computation, libFLAME [42] is for dense linear algebra, and Gunrock [43] is for graph processing. GPU-accelerated libraries, e.g., cuDNN for deep neural network and cuFFT for FFT [44], also exist.

Hardware accelerators. Hardware accelerators have been extensively studied in many research proposals. Conservation Cores [1] and QsCores [2], which are conceptually application-specific hardware accelerators, are proposed to reduce the energy consumption for a wide range of applications. Since several applica-

tions, such as signal processing, data mining, and pattern recognition, can tolerate inexact results, a neural processing unit (NPU) [45, 46] is proposed to accelerate general-purpose codes at the cost of reducing accuracy. For data-intensive applications, Lim *et al.* [47] designed SoC accelerators to efficiently process a widely-used key-value store system as Memcached. Kocberber *et al.* [48] introduced an on-chip specialized accelerator for hash index lookups of in-memory databases. Wu *et al.* [5] proposed HARP accelerator for data partitioning.

Researchers are also aware of the importance of domain-specific accelerators. Milder *et al.* [4] proposed to automatically generate highly efficient FFT cores. Perdram *et al.* [3] proposed linear algebra accelerators for Level-3 BLAS. Wu *et al.* [6] built heterogeneous accelerators for improving energy efficiency of database querying. Madhavan *et al.* [49] proposed the Race Logic for accelerating dynamic programming algorithms. Liu *et al.* [7] designed a polyvalent accelerator for various machine learning algorithms.

Programming heterogeneous systems. Directive-based programming models such as OpenACC [50] are proposed to ease the programming burden of specialized accelerators. Another source of heterogeneity is graphics coprocessors (GPUs). Well known programming models for GPUs include CUDA [51] and OpenCL [52]. Memory management is the key determinant of GPU programming model. Originally, the CPU and GPU have their own virtual memory spaces, as *separate memory space*, leading to tedious efforts to maintain two data copies. Unified Virtual Addressing (UVA) [53] is proposed to enable shared virtual space between the CPU and the GPU. Recently, shared virtual address space [54, 55] is proposed to allow the programmer to simply use the standard memory allocation functions such as `malloc` and `new`.

3D-stacked near-DRAM computing. There are a few literatures on integrating programmable computation logic to the logic layer of 3D-stacked DRAM. Pugsley *et al.* [38] integrated energy-efficient processor cores to the logic die of 3D-stacked DRAM for MapReduce workloads, and compared the advantages of various near-data computing approaches for in-memory MapReduce [56]. Zhang *et al.* [57] proposed to integrate programmable GPUs to 3D-stacked DRAM to offer high throughput. Guo *et al.* [58] built a 3D-stacked memory-side accelerated system with sufficient software support for programming. Azarkhish *et al.* [59] showed the Smart Memory Cube that employs AXI-based interconnect to link a processor-in-memory with vault controllers on the logic die of HMC. Nair *et al.* [9] proposed the Active Memory Cube that integrates SIMD-like processing elements to the logic layer of HMC to balance the programmability and energy efficiency. Farmahini-Farahani proposed a near-DRAM acceleration architecture (NDA) to stack accelerator logics atop conventional 2D DRAM [29]. In contrast to these work, MEALib focuses on accelerating memory-bounded operations that are well-defined by portable library APIs. Hence, MEALib can inherently resolve the programming dilemma for such accelerated systems to achieve portable energy efficiency. Moreover, as such library operations are widely used in many domains, MEALib is applicable for many applications.

7. CONCLUSIONS

High performance libraries are pervasively used in a broad range of applications and computation domains. Hence, we proposed to design accelerators that instantiate the APIs of these commonly used libraries in hardware. In particular, a hardware-software co-design approach is used to target memory-bounded operations within commonly used high performance libraries (e.g., BLAS and FFTW) for portable energy efficiency. The end result is Memory Accelerated Library (MEALib).

From the hardware perspective, accelerators targeting memory-bounded operations are integrated onto the 3D stacked DRAM with extremely high internal bandwidth. This allows us to overcome the memory limitation between memory and computation units. A configuration infrastructure is built to expand the application scope of MEALib. This allows accelerators to be chained in sequence to compute more complicated operations, while reducing the need to design individual accelerators for each library routine. A source-to-source compiler is used to translate the original library calls to MEALib runtime routines for accelerator control and configuration. This allows legacy code to use memory-side accelerators without reimplementation. A shared memory management is also implemented to ease the design of such runtime routines. Experimental results show that, on average, legacy code using MEALib improves performance and energy efficiency by 38x and 75x, respectively, for individual operations in Intel MKL. Finally, MEALib attains more than 10x better energy efficiency over a highly optimized real-world application using conventional software libraries.

Acknowledgements

We thank the anonymous reviewers for their insightful comments. We also thank the CMU PERFECT and SPIRAL team. This work was sponsored by the DARPA PERFECT program under agreement HR0011-13-2-0007. The content, views and conclusions presented in this document do not necessarily reflect the position or the policy of DARPA or the U.S. Government, no official endorsement should be inferred.

8. REFERENCES

[1] G. Venkatesh *et al.*, "Conservation cores: Reducing the energy of mature computations," in *ASPLOS*, 2010.

[2] G. Venkatesh *et al.*, "Qscores: Trading dark silicon for scalable energy efficiency with quasi-specific cores," in *MICRO*, 2011.

[3] A. Pedram *et al.*, "Codesign tradeoffs for high-performance, low-power linear algebra architectures," *IEEE Trans. Comput.*, 2012.

[4] P. Milder *et al.*, "Computer generation of hardware for linear digital signal processing transforms," *ACM Trans. Des. Autom. Electron. Syst.*, 2012.

[5] L. Wu *et al.*, "Navigating big data with high-throughput, energy-efficient data partitioning," in *ISCA*, 2013.

[6] L. Wu *et al.*, "Q100: The architecture and design of a database processing unit," in *ASPLOS*, 2014.

[7] D. Liu *et al.*, "Pudiannao: A polyvalent machine learning accelerator," in *ASPLOS*, 2015.

[8] W. Qadeer *et al.*, "Convolution engine: Balancing efficiency & flexibility in specialized computing," in *ISCA*, 2013.

[9] R. Nair *et al.*, "Active memory cube: A processing-in-memory architecture for exascale systems," *IBM Journal of Research and Development*, 2015.

[10] "BLAS (basic linear algebra subprograms)." `http://www.netlib.org/blas/`.

[11] M. Frigo and S. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, 2005.

[12] "The R project for statistical computing." `http://www.r-project.org/`.

[13] K. Barker *et al.*, *PERFECT (Power Efficiency Revolution For Embedded Computing Technologies) Benchmark Suite Manual*. Pacific Northwest National Laboratory and Georgia Tech Research Institute, 2013.

[14] C. Bienia *et al.*, "The PARSEC benchmark suite: Characterization and architectural implications," in *PACT*, 2008.

[15] "Intel math kernel library (MKL)." `http://software.intel.com/en-us/articles/intel-mkl/`.

[16] J. Cebrian *et al.*, "Optimized hardware for suboptimal software: The case for SIMD-aware benchmarks," in *ISPASS*, 2014.

[17] J. Jeddeloh and B. Keeth, "Hybrid memory cube new dram architecture increases density and performance," in *VLSIT*, 2012.

[18] P. Colella, "Defining software requirements for scientific computing," 2004.

[19] G. H. Loh, "3D-stacked memory architectures for multi-core processors," in *ISCA*, 2008.

[20] D. H. Woo *et al.*, "An optimized 3D-stacked memory architecture by exploiting excessive, high-density tsv bandwidth," in *HPCA*, 2010.

[21] D. H. Kim *et al.*, "3D-maps: 3d massively parallel processor with stacked memory," in *ISSCC*, 2012.

[22] Samsung, "Samsung to release 3D memory modules with 50% greater density," 2010.

[23] B. Akin *et al.*, "Data reorganization in memory using 3d-stacked dram," in *ISCA*, 2015.

[24] B. Akin *et al.*, "Understanding the design space of dram optimized hardware FFT accelerators," in *ASAP*, 2014.

[25] K. Sudan *et al.*, "Micro-pages: Increasing dram efficiency with locality-aware data placement," in *ASPLOS*, 2010.

[26] P. Luszczek *et al.*, "Introduction to the hpc challenge benchmark suite," *ICL Technical Report*, 2005.

[27] F. Sadi *et al.*, "Algorithm/hardware co-optimized sar image reconstruction with 3d-stacked logic in memory," in *HPEC*, 2014.

[28] K. Hwang *et al.*, "Benchmark evaluation of the IBM SP2 for parallel signal processing," *IEEE Trans. Parallel Distrib. Syst.*, 1996.

[29] A. Farmahini-Farahani *et al.*, "Nda: Near-dram acceleration architecture leveraging commodity dram devices and standard memory modules," in *HPCA*, 2015.

[30] "More knights landing xeon phi secrets unveiled." `http://www.theplatform.net/2015/03/25/more-knights-landing-xeon-phi-secrets-unveiled/`.

[31] "Desktop 4th generation intel core processor family, desktop intel pentium processor family, and desktop intel celeron processor family datasheet - volume 1 of 2," 2014.

[32] S. Browne *et al.*, "A portable programming interface for performance evaluation on modern processors," *Int. J. High Perform. Comput. Appl.*, 2000.

[33] "Intel 64 and IA-32 architectures software developerś." `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf`.

[34] K. Chen *et al.*, "CACTI-3DD: Architecture-level modeling for 3d die-stacked dram main memory," in *DATE*, 2012.

[35] Q. Zhu *et al.*, "Accelerating sparse matrix-matrix multiplication with 3D-stacked logic-in-memory hardware," in *HPEC*, 2013.

[36] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, 2011.

[37] R. Gonzalez and M. Horowitz, "Energy dissipation in general purpose microprocessors," *IEEE Journal of Solid-State Circuits*, 1996.

[38] S. H. Pugsley *et al.*, "NDC: analyzing the impact of 3d-stacked memory+logic devices on mapreduce workloads," in *ISPASS*, 2014.

[39] "Intel integrated performance primitives." `https://software.intel.com/en-us/intel-ipp`.

[40] "GotoBLAS2." `https://www.tacc.utexas.edu/research-development/tacc-software/gotoblas2`.

[41] E.-J. Im and K. Yelick, "Optimizing sparse matrix computations for register reuse in sparsity," in *Computational Science*, 2001.

[42] F. G. V. Zee *et al.*, "The libflame library for dense matrix computations," *Computing in Science and Engineering*, 2009.

[43] Y. Wang *et al.*, "Gunrock: A high-performance graph processing library on the gpu," in *PPoPP*, 2015.

[44] "GPU-accelerated libraries." `https://developer.nvidia.com/gpu-accelerated-libraries`.

[45] H. Esmaeilzadeh *et al.*, "Neural acceleration for general-purpose approximate programs," in *MICRO*, 2012.

[46] R. St. Amant *et al.*, "General-purpose code acceleration with limited-precision analog computation," in *ISCA*, 2014.

[47] K. Lim *et al.*, "Thin servers with smart pipes: Designing soc accelerators for memcached," in *ISCA*, 2013.

[48] O. Kocberber *et al.*, "Meet the walkers: Accelerating index traversals for in-memory databases," in *MICRO*, 2013.

[49] A. Madhavan *et al.*, "Race logic: A hardware acceleration for dynamic programming algorithms," in *ISCA*, 2014.

[50] "OpenACC: Directives for accelerators." `http://www.openacc-standard.org/`.

[51] "CUDA C programming guide." `http://docs.nvidia.com/cuda/cuda-c-programming-guide`.

[52] "OpenCL: the open standard for paralle programming of heterogeneous systems." `https://www.khronos.org/opencl/`.

[53] "Nvidia's next generation cuda compute architecture: Fermi." `http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf`.

[54] J. Power *et al.*, "Supporting x86-64 address translation for 100s of gpu lanes," in *HPCA*, 2014.

[55] B. Pichai *et al.*, "Architectural support for address translation on gpus: Designing memory management units for cpu/gpus with unified address spaces," in *ASPLOS*, 2014.

[56] S. Pugsley *et al.*, "Comparing implementations of near-data computing with in-memory mapreduce workloads," *IEEE Micro*, 2014.

[57] D. Zhang *et al.*, "Top-pim: Throughput-oriented programmable processing in memory," in *HPDC*, 2014.

[58] Q. Guo *et al.*, "3D-stacked memory-side acceleration: Accelerator and system design," in *WoNDP*, 2014.

[59] E. Azarkhish *et al.*, "High performance AXI-4.0 based interconnect for extensible smart memory cubes," in *DATE*, 2015.