SAND2014-20215

SANDIA REPORT

SAND2014
Unlimited Release
Printed 2014

Performance and Energy Implications for Heterogeneous Computing Systems: A MiniFE Case Study

Li Tang[†], X. Sharon Hu[†], and Richard F. Barrett[‡]

Department of Computer Science and Engineering[†]
University of Notre Dame
Notre Dame, IN 46556, USA

Center for Computing Research[‡]
Sandia National Laboratories
Albuquerque, NM 87123, USA

Sandia National Laboratories

SAND2014
Unlimited Release
Printed 2014

Performance and Energy Implications for Heterogeneous Computing Systems: A MiniFE Case Study

Abstract

Heterogeneous computing systems, which employ a mix of general-purpose (GP) processors and accelerators such as graphics processing units (GPUs) or Field Programmable Gate Arrays (FPGAs), have the potential to offer much higher performance and lower energy usage than homogeneous systems. However, designing heterogeneous computing systems to achieve high performance and low energy usage is a challenging task. Designs that offer higher performance do not necessarily lead to lower energy consumption. Furthermore, mapping of applications to different computing devices can play a key role in performance and energy tradeoff. In this report, we present a detailed performance and energy study of executing a specific mini-application on different heterogeneous systems. The results show that hardware choices, application implementations, and mapping of applications to hardware can all significantly impact system performance and energy consumption and that the impact on performance and energy can be quite different. This study forms a basis for modeling the interdependencies of program structures and hardware execution units, which could be used to guide design space exploration.

Acknowledgment

The authors thank Sandia National Laboratories for funding this research.

Contents

Summary **9**

1 Introduction **11**

2 Background **13**

Hardware architectures 13

Programming models 14

MiniFE data assembly 16

3 Methodology for evaluating DA energy consumption **17**

4 Implementations **21**

Legacy serial 21

Multi-threading 22

SIMD 22

GPU 23

Heterogeneous 23

5 Evaluation **27**

CPU 27

Single-threaded performance and energy 27

Architectures 27

Compilers 28

Multi-threaded performance and energy 29

Architectures 30

Programming models and compilers 30

SIMD performance and energy 32

GPU 32

FPGA 35

Heterogeneous implementations 38

6 Summary **41**

References **42**

List of Figures

1 Schematic of the power supply for the CPU+GPU system 18

2 Workflow of the DA stage in miniFE 22

3 Using different memory resources for data communication between the STC and SMA kernels. 24

4 Scheduling of STC and SMA on the GPU and CPU. The blue and green rounded rectangles denote the STC kernel on the GPU and SMA functions on the CPU, respectively. The arrows represent data transfer operations. Once the STC kernel finishes computing a set of stiffness matrices, these matrices are transferred to the host memory. When this data transfer is done, a new set of input data are transferred to the GPU's global memory, and the STC kernel and SMA function starts concurrently. This operation repeats until all the sub-domains are processed. 25

5 Performance and energy of using one CPU thread. 28

6 Performance and energy of using the Intel and GCC compilers on the i7 CPU. 29

7 Simulation results of DA on SST. 30

8 Performance and energy of using all available CPU threads. 31

9 Performance and energy of using different multi-threaded programming models with the GCC and Intel compilers. 33

10 Performance and energy of using SIMD. 34

11 Performance and energy of using different memory resources for data communication between the STC and SMA kernels. 36

12 Performance and energy of using GPUs for different kernels. 37

13 Performance and energy of using FPGA with different offloading code. 39

14 Performance and energy of DP and CP implementations of DA on different heterogeneous systems. 40

List of Tables

1 Computing devices 17

Summary

Building high-performance and energy efficient heterogeneous systems is not an easy task. There are many factors can impact the performance and energy efficiency of heterogeneous systems. In this report, we examine the performance and energy impacts of multiple factors covering hardware selection, application design and the interplay between the two. We compare the real performance and energy of executing a specific mini-application on a variety of heterogeneous systems at the single-node level. The comparisons form a basis for modeling the interdependencies of hardware computing devices and application structures, which could be used to guide the design space exploration for heterogeneous computing.

Chapter 1

Introduction

Energy consumption has become a major concern in building future exascale supercomputers [?]. It is widely accepted that heterogeneous computing systems, which employ a mix of GP processors and accelerators such as GPUs or FPGAs, have the potential to offer lower energy solutions without degrading system performance. In recent years, such systems have been gaining popularity in the high-performance computing (HPC) community and leading to increased performance under affordable power budget for HPC applications [?]. However, designing heterogeneous computing systems to achieve high performance and low energy usage is a challenging task.

Many factors impact performance and energy of a heterogeneous system. On the highest level, they can be divided into three categories: hardware choices, application implementations, and the interplay between the two. There are the usual suspects including the complexity of the processor architecture, processor supply voltage and speed, degree of parallelism in the computing device and in the application, actual application implementation, etc. Other factors such as compilers, matching between the algorithms and the processor architectures, and programming models also play a major role in performance and energy.

Furthermore, the impact on performance and that on energy can be rather different and vary greatly from one implementation to another. For example, a system consisting of a less powerful GP processor and a more powerful GPU may be more desirable in terms of performance while a more powerful GP processor together with a less powerful but integrated GPU may be more desirable in terms of energy. Different partitioning approaches such as data partitioning (referred to as DP, where each computing device solves the entire problem but for input datasets) and code partitioning (referred to as CP, where each computing device solves a part of the problem) may result in significantly different performance and energy trends.

To better manage the design complexity of heterogeneous HPC systems, the concept of co-design [?] is being adopted by the HPC community for designing future supercomputers. Co-design methodologies promotes systematic collaboration between the architecture and application developers. To facilitate co-design of architectures and applications, it is important to understand performance and energy implications of different design choices and eventually build models capturing the interdependencies of program structures and hardware execution units. Such models can then be used to support design space exploration.

This report summarizes our effort in using the data assembly stage (referred to as DA) in Finite Element Method (FEM) as a case study to investigate performance and energy implications in heterogeneous systems, FEM is a numerical technique widely used in finding approximate solutions for many scientific and engineering problems, such as simulation of fluid dynamics and particle transport. FEM is roughly decomposed into the DA and solver stages. Depending on the sizes of the particular problems, DA execution can take up to 50% of FEM's total execution time [**?**]. DA is responsible for generating an equation system based on the input object and mainly possesses multiple different computation and memory behaviors. To reduce the development cost and make the study closer to real HPC FEM applications, we use miniFE [**?**], a proxy FEM application, as our target application. Using DA in miniFE instead of simple benchmarks allows us to investigate a number of different design strategies on heterogeneous systems.

In this work, we implemented DA in different ways on single-socket heterogeneous systems at the single-node level. In addition to the hardware and application design considerations, we also consider using different multi-threaded programming models with the GCC and Intel compilers. We compare different designs by using performance and energy efficiency. General energy efficiency is defined as 'using less energy to provide the same service' [**?**]. Hence, we use energy consumption of an identical problem size of DA on heterogeneous systems to represent the energy efficiency. We use direct performance and power measurement to gather the relevant data. Our results show that there is no single implementation of our target application can fit to all heterogeneous systems for achieving the best performance or energy efficiency.

Chapter 2

Background

In this chapter, we provide some details about our targeted hardware architectures and considered multi-threaded programming models. We also give a brief introduction to the DA stage in miniFE.

Hardware architectures

To conduct an in-depth study of the impact of hardware architectures on performance and energy, we examined a number of different computing devices such as CPUs, GPUs and FPGA. Below, we present some details about each of these devices.

Intel's i7 2600K is a 3.4GHz quad-core processor with 32nm technology node. Each core is based on the Sandy Bridge microarchitecture and has 64 KB L1 data cache and 256 KB L2 cache. The Sandy Bridge microarchitecture supports 256 bit wide SIMD operations and AVX2 instruction set. The i7 2600K CPU has 8MB Intel smart cache and features turbo scheme that can dynamically increase the CPU frequency to 3.8GHz. Hyper-Threading Technology (HTT) is utilized to enable the use of two logical processors on one single physical core. The main difference between i7 2600K and i3 2100T is that the i3 2100T CPU has only two cores, 3MB smart cache and 2.5GHz base frequency.

Intel's Atom 330 is a 1.60 GHz dual-core processor with 24 KB L1 data cache and 512 KB L2 cache in each core. The Atom 330 core uses 45nm technology node and supports HTT. Two discrete physical dies are integrated on the same substrate and communicate with each other through a 533MHz Front-side Bus (FSB). The Atom 330 CPU uses a relatively old microarchitecture (was released in 2008) comparing with the Sandy Bridge microarchitecture. The thermal design power (TDP) of Atom 330 is only 8 watts. The Atom 330 CPU is mounted with the 945GSE chip-set and 2GB of DDR2-533 memory.

GPU architecture is a cluster of many SIMD-like processing elements (PEs), where a single instruction can be issued and executed over multiple PEs per cycle. NVIDIA's Kepler [?] GPU architecture integrates 192 GPU cores onto a single streaming multiprocessor (SMX). Each SMX is equipped with 255 register files for private fast data storage. Also, a block of L1 cache is used to cache global memory operations for each SMX. Shared memory and L1 cache can be configured with three modes in Kepler: 16KB+48KB, 32KB+32KB and 48KB+16KB. To store large data sets, GPUs are equipped with off-chip global memory which is available to all SM cores.

AMD's Trinity Accelerated Processing Units (APU) is a heterogeneous processor which integrates a quad-core Piledriver processor and a Radeon HD 7660D GPU on a single die. A Radeon HD 7660D GPU has 384 GPU cores which are organized as six SIMD units. Each SIMD unit is equipped with 64 GPU cores and forms a 16 four-way Very Long Instruction Word (VLIW) Thread Processors (TP) sharing 32KB local data share (LDS). TP and LDS correspond to NVIDIA's GPU cores and shared memory. The core feature of AMD Trinity APU is the shared memory controller and multiple integrated GPU (iGPU) memory accessing paths. The Radeon Memory Bus (RMB) [**?**] manages a region of system memory allocated to iGPU as its global memory and features full iGPU memory bandwidth. The GPU part of A10-5800K can be allocated with up to 2GB RAM of system memory as its dedicated GPU global memory. The Fusion Compute Link (FCL) connects the graphics memory controller to the unified northbridge (UNB). The UNB also manipulates the CPU memory requests, which means iGPU could also access the cached system memory. The tight coupling of integrated CPU (iCPU) and iGPU and the sharing of system memory provide different ways for iCPU/iGPU communication and reduces the cost of data transfer between iCPU and iGPU.

The Freescale i.MX6 processor is a heterogeneous processor which integrates a quad-Core ARM Cortex A9 processor and a Vivante GC2000 GPU. The ARM Cortex A9 core runs at up to 1.2GHz and features dual-issue superscalar and out-of-order microarchitecture. Each core has 64KB four way associative L1 data caches and 4MB of L2 cache. The i.MX6 processor is equipped with 1GB of 64-bit wide DDR3 RAM at 532MHz. The integrated GPU part of i.MX6 possesses 16 GFLOPS peak performance and supports OpenCL standard.

Altera's Stratix V FPGA is based on 28nm technology node. Stratix V possesses some transceivers with up to 28 Gbps speed and a unique array of special function intellectual property (IP) blocks. The Nallatech P385-D5 FPGA card mounts an Altera Stratix V GS D5 FPGA with the support of Altera's OpenCL SDK. The Stratix V GS D5 FPGA contains 457K equivalent Logic Elements (LEs) and 1590 variable precision DSP blocks. Each variable precision DSP block is equivalent to two 18x18 multipliers with 32bit resolution. The TDP of the P385-D5 FPGA card is only about 30W.

Programming models

In this section, we provide a brief summary of the programming models have been used to program on our targeted computing devices.

Message Passing Interface (MPI) is the de-facto standard for parallel programming on distributed memory computing systems. The MPI implementation provides a scheme to define parallel systems in terms of processes. A group of processes communicate with each other by sending and receiving messages under a protocol of communication. MPI is supported in multiple languages such as C/C++ and FORTRAN. The MPI programs can also run on shared memory multi-core CPU systems.

OpenMP is an interface that has been widely used for multi-threaded programming on shared memory computing systems. The OpenMP implementation consists of a set of directives defining parallelism construct and exploiting data parallelism. The OpenMP directives guide the C/C++ or FORTRAN compilers to generate binaries using multiple concurrent threads on multi-core CPUs.

Threading Building Blocks (TBB) is a C++ library for multi-threaded programming on multi-core CPU systems. TBB defines a program as fine-grained tasks and maps the tasks to CPU threads. The TBB library provides some schemes to automatically schedule the fine-grained tasks based on the program's parallel processing pattern.

Compute Unified Device Architecture (CUDA) [**?**] is only used for programming NVIDIA's GPUs. CUDA can be viewed as an extension of the C language facilitating the use of GPUs for computation. The extension includes language-level constructs for defining kernels, execution threads, thread blocks and grids. A CUDA kernel is a program function to be executed on the physical GPU. A thread block contains multiple threads executing the same kernel, and one thread is mapped to one GPU core that can access its private register files and shared memory. The actual construction of threads and kernels for an application greatly impacts the available parallelism and memory bandwidth usage, and hence leads to drastically different performance/energy values.

Open Computing Language (OpenCL) [**?**] is a C-based programming language that becomes a de-facto standard for writing parallel applications on a variety of computing devices (including CPUs, GPUs and FPGAs). A simple OpenCL application typically includes one ANSI C-based host file and one OpenCL C kernel file. The host file uses OpenCL API to communicate with the OpenCL kernel running on hardware accelerator and can be compiled by standard C compiler. OpenCL adopts just-in-time (JIT) compilation which means that a kernel file can be compiled during execution. This dynamic compilation scheme offers strong portability and flexility, but may increase the program execution time.

The Altera OpenCL SDK 13.1 implements Khronos OpenCL 1.0 standard and allows easy compilation of regular OpenCL kernels onto Altera FPGAs. The Altera OpenCL SDK maps an OpenCL kernel onto a compute unit by using pipeline architecture for optimal throughput. Depending on the available FPGA resource, multiple compute units can be generated. The OpenCL run-time scheduler breaks the data-parallel tasks into chunks and sends them to the generated compute units. By using the Altera OpenCL SDK, a single OpenCL kernel file is translated into a single bitstream FPGA configuration file which also includes the implementations of PCI-Express (PCIE)/memory controllers and OpenCL run-time scheduler.

MiniFE data assembly

To explore different approaches of implementing DA on heterogeneous systems and study their performance and energy, we use miniFE, a proxy FEM application, as our target application. MiniFE is one mini-application of the Mantevo Suite [?] which is developed by Sandia National Labs for helping researchers contribute to HPC application development. The project has created multiple self-contained mini-applications possessing essential performance characteristics of real HPC applications. Mini-applications include core functions and operations of real HPC applications, thus can be used to predict the performance trends [?] of the associated HPC applications. MiniFE simulates FEM on a 3D mesh object with a configurable size in each dimension. MiniFE can be mainly decomposed into two stages: (i) DA and (ii) CG solver. A discrete linear system of equations for the input problem is generated by DA and then solved by the CG solver which has already been well studied on using GPUs [?], [?].

In the legacy serial implementation of DA in miniFE, there are two main functions: (i) stiffness matrix computation (STC) and (ii) stiffness matrix assembly (SMA). STC computes and generates an 8×8 local stiffness matrix for a single cube element in the input 3D mesh by using the generated 3D coordinates. A data item in a local matrix describes the physical relationship between two nodes in the associated element and is computed by using the node information. Comparing with STC's high compute intensity, SMA is mainly composed of memory operations. SMA uses binary search to find the destinations in the compressed global matrix for each data item in the local stiffness matrices for accumulating the data items.

Chapter 3

Methodology for evaluating DA energy consumption

To study the performance and energy of DA on different systems, we consider eight different computing devices. These computing devices cover low-power/high-performance CPUs, two performance grades of Kepler GPUs, integrated APU, ARM processor and FPGA. Table 1 shows the details. All the desktop heterogeneous systems use the same CORSAIR CX600 power supply unit, and run the same CentOS 6.3 64-bit operating system. The ARM board uses a 5V DC power supply. The GCC 4.3 compiler is used to produce optimized host-side binaries at the optimization level of `-O3`. The NVCC compiler of CUDA 5.0 is used to generate the kernel binaries on the NVIDIA GPUs.

In this work, we use the energy consumption of CPU or CPU plus GPU (or FPGA) to explore the energy behaviors of different DA implementations. The reason of not using system energy in our study is that some components in the system like disk do not use energy for computation. For collecting energy data, we have built a power measurement system. Figure 1 illustrates the system schematic using a discrete CPU+GPU system as an example, where the CPU motherboard connects with the GPU card through the PCIE interface.

The desktop CPU motherboards have a dedicated 12V DC input for CPU power supply and a voltage regulator module (VRM) for dynamically converting the 12V DC voltage to

Table 1. Computing devices

Name	Detail	# of Cores	Clock Speed
i7	Intel Core i7 2600K CPU	4	3.4GHz
i3	Intel Core i3 2100T CPU	2	2.5GHZ
ATOM	Intel Atom 330 CPU	2	1.6GHz
iCPU	AMD A10 5800K's iCPU	4	3.8GHz
iGPU	AMD A10 5800K's iGPU	384	0.8GHz
ARM	Freescale i.MX6 CPU	4	1GHz
Titan	NVIDIA GeForce GTX Titan GPU	2688	0.84GHz
GTX750	NVIDIA GeForce GTX750 GPU	512	1.02GHz
FPGA	Nallatech P385 D5 FPGA	Varies	Varies

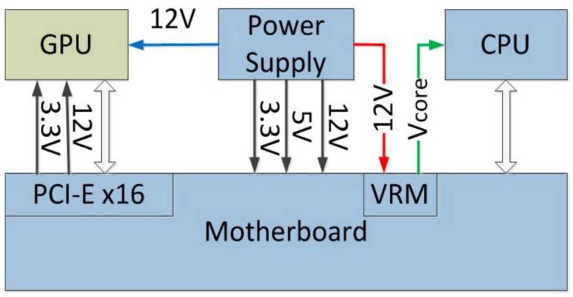


Figure 1. Schematic of the power supply for the CPU+GPU system

the actual CPU operating voltage. Although the 12V supply includes the energy overhead of the VRM, such inclusion is reasonable when deriving the GPU or FPGA card energy because the card energy includes the energy loss of the onboard VRM. Since there is no dedicated power input for the Atom processor, we insert a wire (the V_{core} arrow in Figure 1) between the VRM and the CPU in order to deliver the V_{core} DC input to ATOM. For the ARM energy consumption, we use the whole board energy as the CPU energy since most of the board energy is used by the ARM processor.

To ensure a fair comparison, we add an additional 20% VRM energy loss (similar to that used in other papers, e.g., [?]) to the measured Atom energy. The GPU power supply is roughly decomposed into a 12V auxiliary rail (the AUX arrow in Figure 1) on top of the GPU card and the power lane from the PCI-Express interface which can be further classified into the 3.3V and 12V lanes.

We use measured DC currents to derive energy usage. Four FLUKE 80i-110s clamps are employed to continuously capture the current values on different power lanes. A PCI-Express riser card is inserted between the PCI-Express interface and the GPU or FPGA card to separate the power pins of the card. An NI USB 6126 data acquisition system is used to collect the readings of clamps and deliver the data to a computer for record keeping. The sampling rate is 10,000 samples-per-second.

To obtain energy data, we use the measured current values and leverage the fact that the observed supply voltages fluctuate up to 5% of the voltage specifications. The energy usage is thus calculated by

$$Energy = \sum_i Fixed\_supply\_voltage * Current_i * \Delta_t, \tag{3.1}$$

where $\Delta_t = 0.0001$ seconds (the inverse of the sampling rate), and i represents the i_{th} sampling period. We use this equation to compute the energy of both CPU and GPU or FPGA during the execution, but we use the total energy of CPU and GPU or FPGA (if equipped with GPU or FPGA) as the energy consumption in the following sections. For performance data, we obtain the execution time of a program by using timers inserted directly into the program.

Chapter 4

Implementations

In this chapter, we talk about different implementations of DA on our heterogeneous systems.

Legacy serial

We use the original serial version of DA in miniFE as the single-threaded implementation of DA. We refer this implementation to as `Serial`. The workflow of the serial implementation of DA is illustrated in Figure 2. DA outputs a large global matrix to form the discrete linear system of equations. The global matrix is sparse and symmetric, and is stored in the format of compressed sparse row (CSR) which uses row starting indices and column indices to locate solely non-zero (NZ) data items in the global matrix.

STC computes and generates a local stiffness matrix for each element in the input 3D mesh to reflect the geometric and material properties of that element. For the serial implementation of STC, the input and output for one single element in the input 3D mesh are 24 3D coordinates of eight hexahedron nodes and an 8×8 local stiffness matrix. The main body of the serial version of STC is a three level nested loop for computing the physical relationships of any two nodes of the associated element. This also explains why the DA stage in miniFE is insensitive to memory bandwidth.

SMA is responsible for assembling all the local matrices into the global matrix by accumulating data items in local matrices to the corresponding positions in the global matrix. Hence, SMA is mainly composed of memory operations. The input of SMA is the global CSR matrix and all local stiffness matrices with their associated element coordinates. The output of SMA is the global CSR matrix after accumulating the data items in those input local stiffness matrices. SMA iteratively searches the positions in the global matrix for each data item in local stiffness matrices by using binary search on the CSR structure of the global matrix. There is no data dependency between any two elements for STC and SMA, which makes the DA stage suitable to be highly parallelized.

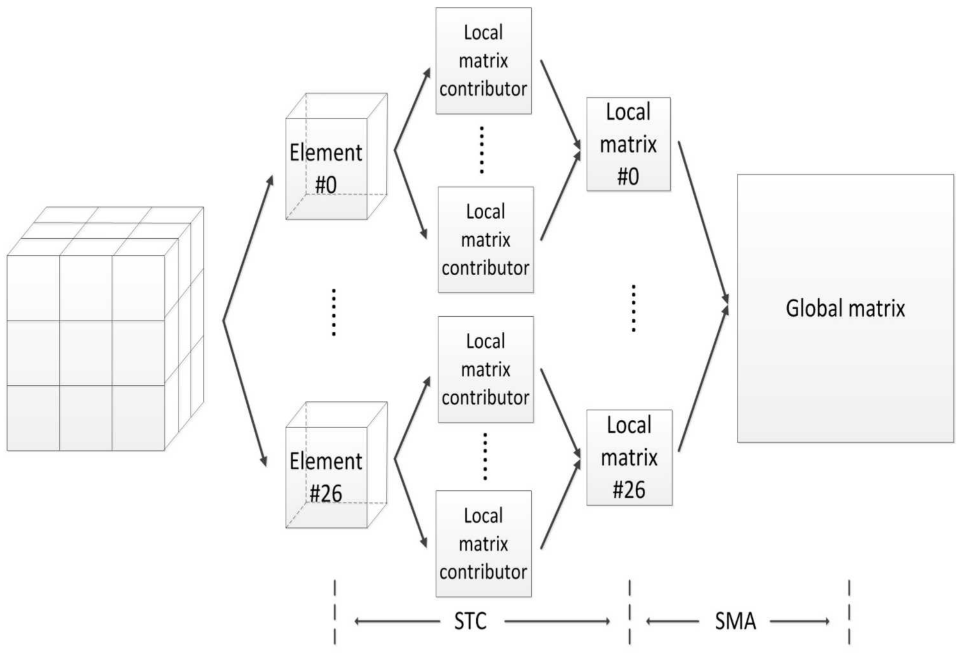


Figure 2. Workflow of the DA stage in miniFE

Multi-threading

In addition to the original TBB and MPI multi-threaded implementations of DA (refer to as `TBB` and `MPI`), we also use OpenMP (refer to as `OMP`) to parallelize DA on multi-core CPUs. OpenMP is a directive-based API that provides multi-threaded programming ability on shared memory computing systems. We straightforwardly distribute the workload of DA onto each CPU thread evenly by using the OpenMP static loop scheduling scheme. The basic workload unit to be distributed is the work associated with one cube element in DA. This simple element-wise scheme is used due to that the computation of any two elements in DA are data independent and possess similar workload.

SIMD

To further exploit the power of CPUs with SIMD support, we employ Advanced Vector Extensions 2 (AVX2) intrinsics to use SIMD on i7, i3 and iCPU. The Sandy Bridge and Piledriver microarchitectures support AVX2 and possess 256-bit wide SIMD registers and execution units. Namely, when fully using the 256-bit register, the ideal throughput is 8X higher than not using SIMD for single precision floating point operations. Based on the OpenMP implementation of DA, we assign the workload of eight elements to one thread for utilizing wider execution units. To implement the binary search in SMA, we have two implementations. One is a pure AVX2 implementation, and the other one is a mixed implementation of AVX2 intrinsics and GCC built_ins (refer to as `AVX` and `MIXED`). `AVX` uses concurrent comparison in wide registers to brutally search the data item locations in the CSR structure. `MIXED` uses GCC built_ins, such as _popcount and _popcount, for parallelizing the comparisons in binary search.

GPU

In this section, we discuss the details of porting CPU implementations of STC and SMA to GPU. The serial implementations of DA process each cube element in the 3D mesh independently. Multi-threaded implementations of DA follow the element-wise approach and simply assigned one thread to the workload of one element. To increase the parallelism of the multi-threaded implementations of STC on GPU, we break a single thread into eight threads. Then each thread is responsible for eight elements and handles the computation of one row of each 8×8 local matrix (hence different threads handle different rows of the 8×8 matrices). That is, each thread computes the same row of the 8 local matrix contributors and accumulates them before writing the results to the local matrix.

Unlike the STC function which is dominated by computation, the SMA function is memory-bound. The performance bottleneck for the SMA's GPU implementation lies in the exploitation of memory bandwidth. To better utilize the GPU memory bandwidth, one thread is responsible for accumulating the 8 data items in one row of the 8×8 local matrix to the global matrix. Since the eight threads of each local matrix synchronously access eight data items in a row of the local matrix, these accesses have close global memory addresses and may be coalesced into a single memory request [?].

Based on the developed GPU implementations of STC and SMA, we present two different implementations of DA, `GmemDA` and `SmemDA`, on GPU. Figure 3 illustrates the main ideas of the two design strategies. The main difference is that the global memory and shared memory are used for data communication between STC and SMA. `GmemDA` uses separate kernels for STC and SMA and uses the global memory for data communication between the two kernels. `SmemDA`, on the other hand, fuses the STC and SMA kernels into one single kernel through the use of shared memory for data communication.

Heterogeneous

This section focuses on design strategies of using both the CPU and GPU for the whole DA on heterogeneous systems. Current related studies focus on selecting the appropriate device of heterogeneous systems for computation [?] and balancing the workload distribution on CPU and GPU through the DP method [?]. DP divides the input data sets into two

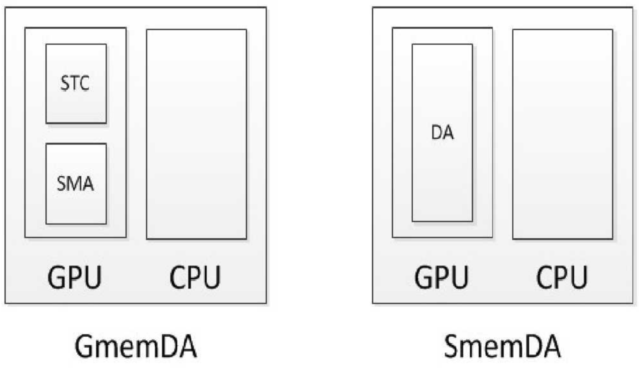


Figure 3. Using different memory resources for data communication between the STC and SMA kernels.

subsets to be handled by the CPU and GPU separately[1]. Though DP can help balance workload, it cannot fully exploit the distinct capabilities of the CPU and GPU. To address this issue, we employ CP to separate the original target code and only offload the appropriate code onto the GPU.

We use `OMP` and `SmemDA` to form a DP implementation of DA for fully stressing the CPU and GPU. Since `OMP` and `SmemDA` are complete DA implementations, we partition the workload (the input elements in 3D mesh) into two parts and distribute them onto the CPU and GPU. To balance the CPU/GPU workload distribution, we calculate a rough workload partitioning ratio by using the performance of `OMP` and `SmemDA` on the targeted heterogeneous systems. We then manually tune the ratio to make sure that the CPU and GPU can start and terminate at the same time due to the synchronization and non-overlapped data transfer overhead.

The idea of CP approaches is to only offload the appropriate code of a whole application onto the GPU for better using GPU. The CP1 implementation of DA is a straightforward implementation of DA as it only offloads STC onto the GPU. The main work of SMA is binary searching and data accumulation, which means SMA is memory intensive and contains a lot of irregular memory operations. While the work of an element of STC is iteratively accumulating an 8×8 local stiffness matrix by computing a new 8×8 local stiffness matrix contributor. Intuitively, SMA and STC is appropriate for the CPU and GPU, respectively. Hence, we only offload STC onto the GPU in CP1. Local matrices are transferred from the GPU to the main memory between the SMA and STC kernels. We also use a simple software pipelining scheme to overlap the SMA and STC execution. Figure 4 depicts the scheme. Since data size for each stage could not be too large or too small, we set each CPU and GPU stage process an input subdomain of size 25^3. This size can keep the GPU running with enough workload and also makes most of the CPU and GPU stages overlapped.

[1]Some papers refer to input data partitioning as workload partitioning.

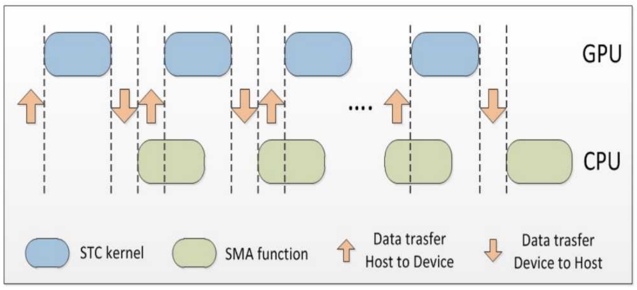


Figure 4. Scheduling of STC and SMA on the GPU and CPU. The blue and green rounded rectangles denote the STC kernel on the GPU and SMA functions on the CPU, respectively. The arrows represent data transfer operations. Once the STC kernel finishes computing a set of stiffness matrices, these matrices are transferred to the host memory. When this data transfer is done, a new set of input data are transferred to the GPU's global memory, and the STC kernel and SMA function starts concurrently. This operation repeats until all the sub-domains are processed.

The CP2 implementation of DA further removes some GPU unfriendly code out of STC and offloads the rest code onto the GPU. In STC, there are some division and branching operations that may degrade GPU performance. Therefore, in CP2, we only offload the key computation part of STC onto the GPU. The key advantages of CP2 over CP1 are higher GPU usage and lower GPU register pressure. We use the same software pipelining scheme of CP1 to schedule the CPU and GPU stages in CP2.

Chapter 5

Evaluation

In this chapter, we present our study results of running DA on the different heterogeneous systems discussed earlier. Impacts of workload distribution, programming models and compilers are also analyzed. All results are based on the DA problem size of $100 \times 100 \times 100$ with the single precision floating point representation.

CPU

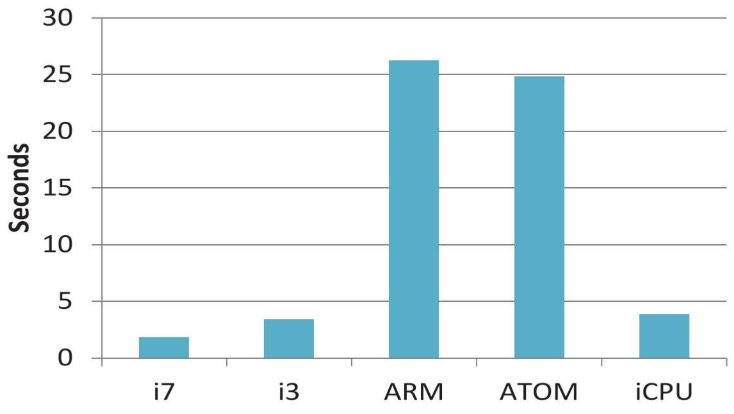
Comparing the performance and energy of CPUs is not an easy task since there are multiple design considerations of using a CPU. In our study, we consider running DA on different CPU parallelism levels such as single-threaded, multi-threaded and SIMD. We also consider using different multi-threaded programming models and compilers for DA. Our results show that the parallelism level significantly impacts the CPU energy.

Single-threaded performance and energy

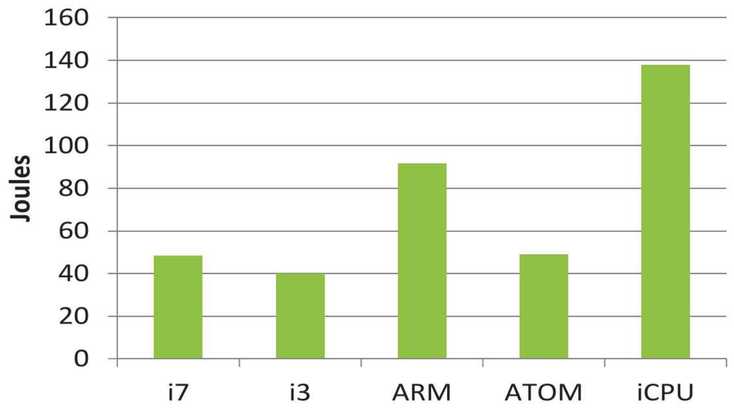
We first look at the performance and energy of the serial implementation of DA on a single thread of CPUs. We also use the GCC and Intel compilers to examine the performance and energy impact of using different compilers.

Architectures

We select Intel i7 2600K, Intel i3 2100T, Intel Atom 330, Freescale i.MX6 and AMD APU A10-5800K as the target CPUs. The executable binary is generated by using the GCC compiler. The single-threaded performance and energy results are shown in Figure 5. We can see the i7 CPU has the best single-threaded performance and is 14X faster than the slowest ARM A9 CPU. ARM A9 has similar (6% slower) single-threaded performance with the low power ATOM CPU. The i7 and i3 CPUs have the same Sandy Bridge core architecture, but the i7 CPU has larger L1 cache size and higher frequency than i3. This difference makes the i7 CPU has 47% higher single-threaded performance than the i3 CPU.



(a) Performance



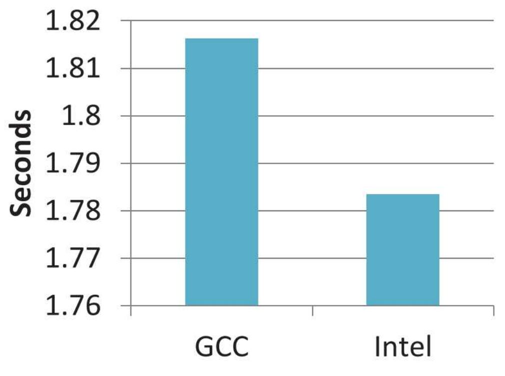
(b) Energy

Figure 5. Performance and energy of using one CPU thread.

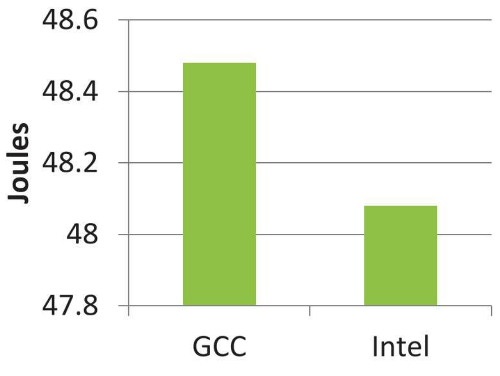
For energy, the i3 CPU has the lowest single-threaded energy usage. The i7 and ATOM CPUs have similar energy consumption which is 21% higher than the i3 CPU. The i.MX6 ARM uses about 128% more energy than the i3 CPU and 35% less energy than APU which is the least energy efficient CPU. The energy results indicate heavyweight X86 CPUs with modest L1 cache size, such as i3, can achieve low single-threaded energy use. Two reasons contribute to this observation. One is that the heavy DA data reuse significantly limits the benefits from having large L1 cache. The second reason is that the complicated X86 CPUs usually have lower uncore energy [?] than simple core CPUs due their higher performance.

Compilers

For the highest performance i7 CPU, we examine the performance and energy impact of using two different compilers: GCC 4.4 and Intel 13.0. The performance and energy results



(a) Performance



(b) Energy

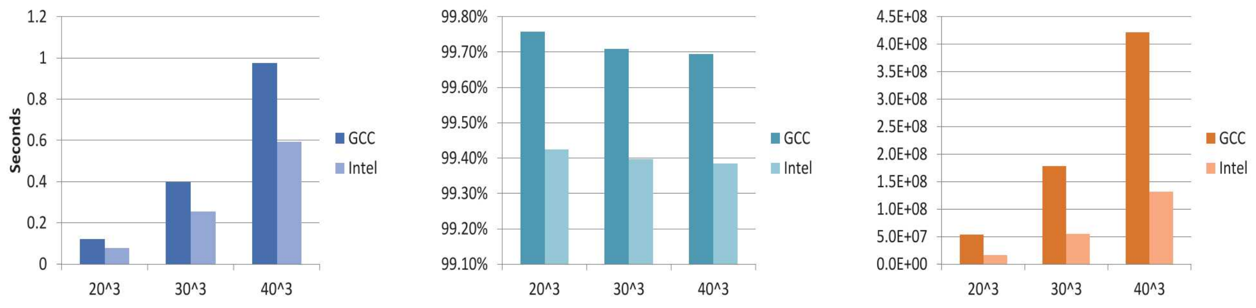
Figure 6. Performance and energy of using the Intel and GCC compilers on the i7 CPU.

are shown in Figure 6. By using the Intel 13.0 compiler, the i7 CPU performs 5% faster than by using the GCC compiler. However, the two compilers have very similar energy consumption. We hypothesize that the Intel compiler generates a binary that pushes the CPU towards higher performance and results in higher instant CPU power than the GCC compiler.

To better understand the performance difference of generated binaries by the two compilers, we simulated the memory behaviors of DA by using SST 4.0.0 with the Ariel memory simulation support. The target simulated core architecture is a simple in-order core with 3 levels of data cache: 32KB L1, 256KB L2 and 2MB L3. Due to the low simulation speed, we only simulated the problem sizes of $20 \times 20 \times 20$, $30 \times 30 \times 30$ and $40 \times 40 \times 40$. The binaries are generated by using the same local machine. Figure 7 shows the simulated performance, numbers of generated memory requests and L1 data cache hit ratios. We can see that the trend of simulated performance agrees with the real performance. The Intel compiler can generate binaries of DA with about 38% higher performance than the GCC compiler. The L1 data cache hit ratios of the Intel compiler binaries are much lower than the GCC compiler executables. However, the Intel compiler can generate more optimized binary with about 69% less memory requests than the GCC compiler. This is because the data structure used in DA can be easily vectorized and can take advantages of the Intel compiler for generating faster binaries.

Multi-threaded performance and energy

A fair performance and energy comparison of multi-core CPUs should be based on using all the available cores of CPUs. An interesting comparison would be the energy of two CPUs based on the same architecture but with different amounts of computing resource (e.g., cores and cache size). We also consider the performance and energy impact of using three different



(a) Estimated performance (b) L1 data cache hit ratios (c) Total memory requests

Figure 7. Simulation results of DA on SST.

multi-threaded programming models: MPI, TBB and OpenMP.

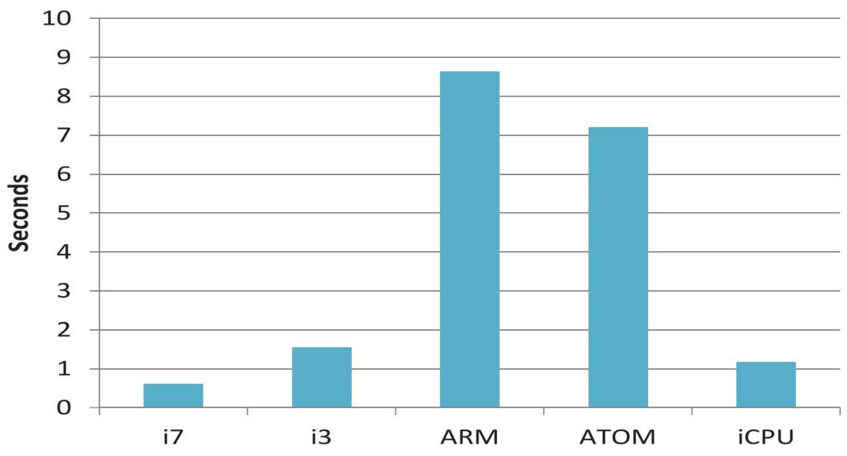
Architectures

To compare the multi-threaded performance and energy of our selected CPUs, we use OpenMP to launch the maximum available number of threads of each CPU in experiments. The Intel i7 and i3 CPUs feature hyperthreading which allows two concurrent threads to share the execution units in one physical core. Figure 8 shows the performance and energy results. The multi-threaded performance trend is similar to the single-threaded performance trend except that iCPU has higher performance than i3. The reason behind this performance difference is that iCPU has 4 cores, which doubles the number of cores of the i3 CPU.

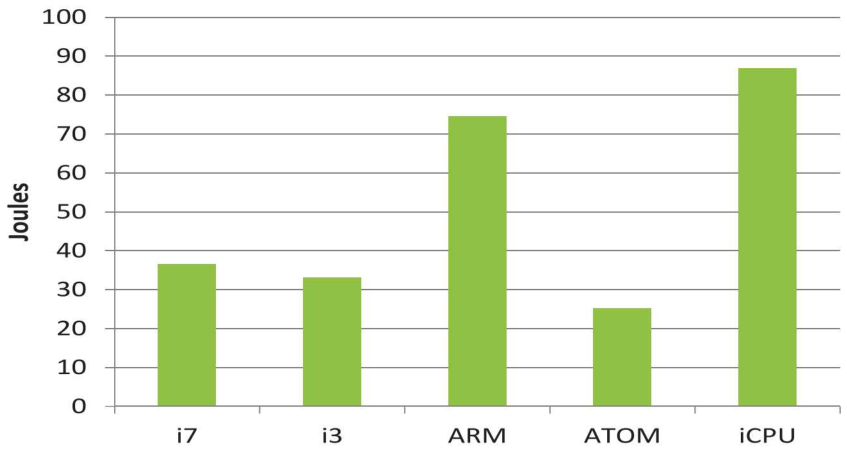
For energy, ATOM has the lowest energy usage even though it is made by using the old 45nm technology node. This fact shows the energy is not dominated by the CPU lithography. As a powerful mobile CPU, ARM uses a different instruction set and 32bit addressing, which makes it consume almost 3X the energy of the ATOM CPU. Based on our energy results, an energy efficient CPU design would feature new technology node, lightweight cores and powerful Dynamic Voltage and Frequency Scaling (DVFS) scheme with more power states. Another conclusion we can draw is that the number of cores with the same architecture on one chip is not strongly related to the energy consumption when all available cores are used. The i3 CPU has 17% and 9% lower energy consumption than the i7 CPU for single- and multi-threaded implementations, respectively. The reason is that the energy impact of off-core components has been weakened when using multiple cores on a same chip.

Programming models and compilers

There are many choices for multi-threaded programming on CPUs. In this section, we consider using the GCC and Intel compilers with three multi-threaded programming models: OpenMP, TBB and MPI. All the related experiments are based on the i7 CPU. The



(a) Performance



(b) Energy

Figure 8. Performance and energy of using all available CPU threads.

performance and energy results are illustrated in Figure 9. The Intel compiler works better than the GCC compiler with all three multi-threaded programming models for generating binaries with higher performance. This trend can also be found in single-threaded mode 5. For the TBB, MPI and OpenMP implementations of DA, binaries generated by using the Intel compiler perform 5%, 13% and 2% faster than using the GCC compiler. With the GCC compiler, the OpenMP implementation of DA performs 24% and 21% slower than TBB and MPI. This is due to the use of static scheduling scheme in the OpenMP implementation of DA, which makes the workload of each thread is not well balanced.

For energy, the MPI implementation with the Intel compiler uses the lowest energy and saves about 33% energy of the OpenMP implementation with GCC. The Intel compiler generates more energy efficient binaries than the GCC compiler for all the multi-threaded programming models. However, the Intel compiler with OpenMP only saves less than 1% energy than the GCC compiler with OpenMP. This small energy saving can also be found in the single-threaded energy comparison of the two compilers 5.

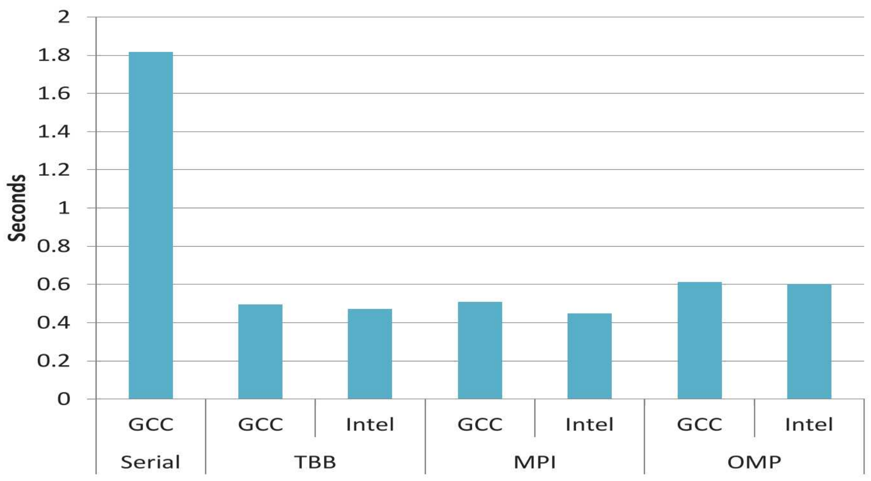
SIMD performance and energy

The i7, i3 and APU CPUs feature wide SIMD registers and execution units. In this section, we examine the performance and energy of using SIMD on these CPUs. The Advanced Vector Extensions 2 (AVX2) instructions are supported by these CPUs and used with OpenMP to fully exploit the CPU computation power. Figures 10 shows the performance and energy results of our two SIMD implementations with the GCC and Intel compilers. It can be seen from the data that the SIMD implementations on i7 have about 3x better performance and energy efficiency over the OpenMP implementations of DA. For iCPU, SIMD performs 2.3x faster and 1.7x more energy efficient than OpenMP. This indicates that using wider register and execution units does not significantly increase the instant power. Namely, using SIMD can save the total energy usage by reducing the execution time and slightly increasing the instant power. When using different compilers, the pure and mixed AVX2 implementations of DA run about 8% faster on average with the Intel compiler than the GCC compiler.

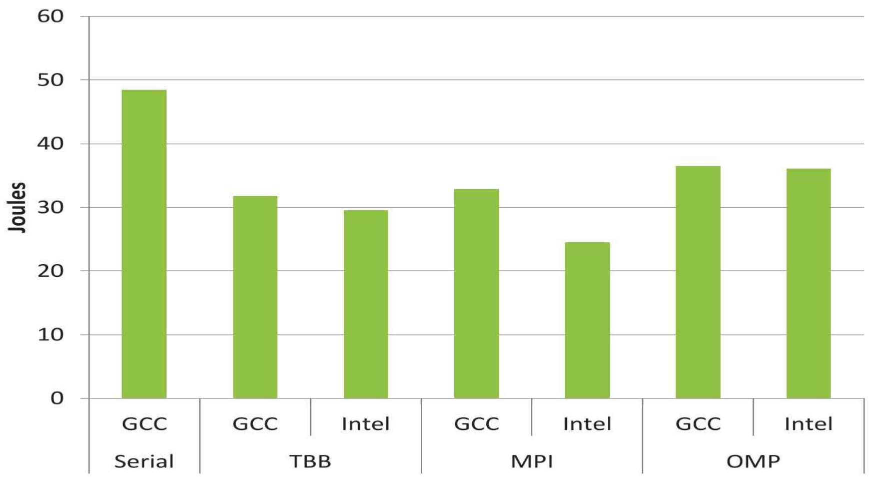
For energy, the pure AVX2 implementation with the GCC compiler consumes 6% less energy than the mixed AVX2 with the Intel compiler.

GPU

In this section, we focus on examining the performance and energy of running different kernels on GPUs. For different GPUs, we tune the number of launched threads and blocks to achieve the best performance. The execution time and energy include the impact of data communication between host and GPU. We first examine two implementations of DA, `GmemDA` which utilizes the global memory for communication between the STC and SMA

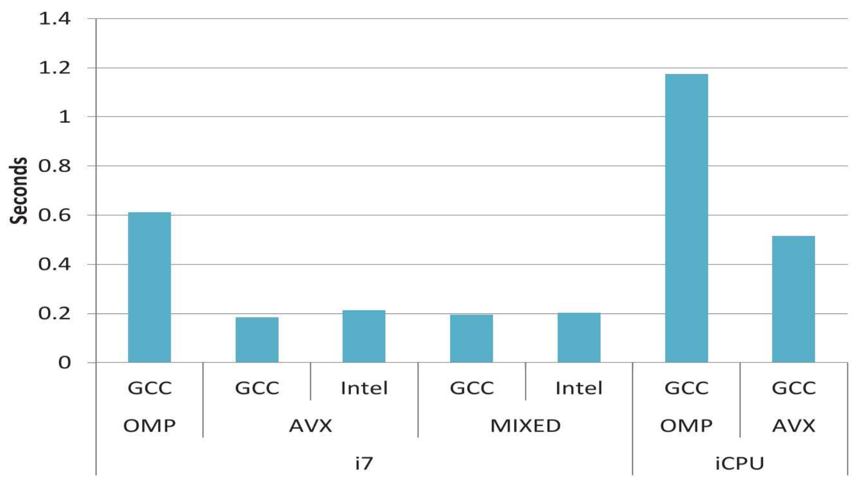


(a) Performance

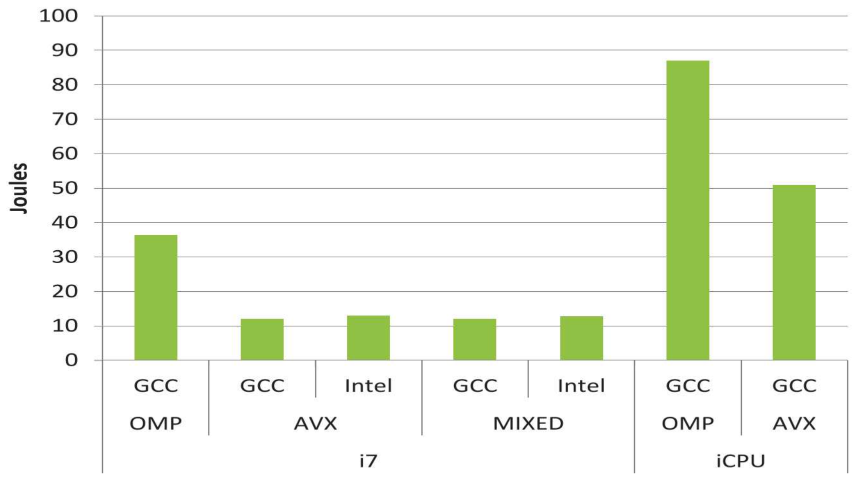


(b) Energy

Figure 9. Performance and energy of using different multi-threaded programming models with the GCC and Intel compilers.



(a) Performance



(b) Energy

Figure 10. Performance and energy of using SIMD.

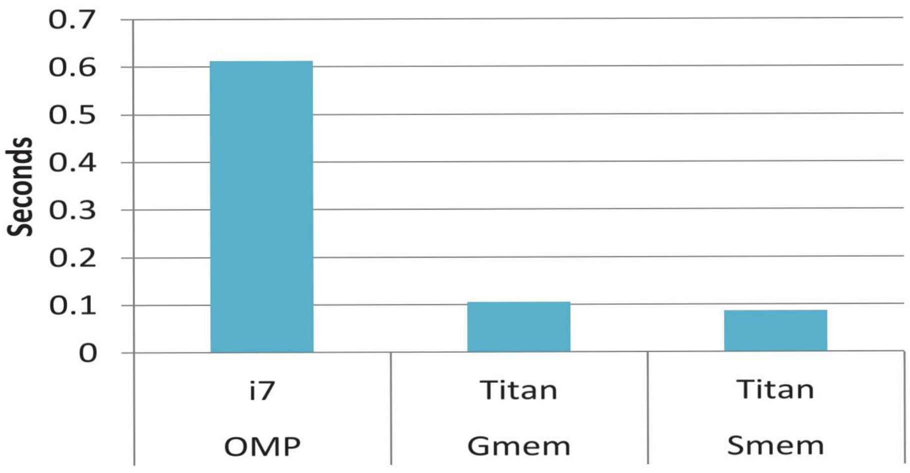
kernels, and `SmemDA` which leverages the shared memory to fuse the STC and SMA kernels into a single kernel. The performance and energy results of `GmemDA` and `SmemDA` are shown in Figure 11. Both `GmemDA` and `SmemDA` can run about 6X faster and use 58% less energy than the OpenMP implementation of DA on i7. `SmemDA` removes global memory accesses between STC and SMA, and hence leads to better performance speedup and less energy usage than `GmemDA`. When using the global memory, `GmemDA` is 18% faster and with 16% less energy consumption than `GmemDA`. This observation indicates that the global memory is slow and power hungry and the large communication overhead between STC and SMA is a key issue for improving performance and reducing energy.

To explore the GPU performance and energy impacts of different kinds of kernels, we also run the compute intensive STC and memory intensive SMA kernels on GPUs. Figure 12 shows the performance and energy data. The Titan GPU runs about 2X faster than the GTX750 GPU for all the three kernels. The DA kernel is composed of the STC and SMA kernels and shared memory is used for communication between STC and SMA for reducing global memory communication overhead. The lower memory bandwidth requirement of DA makes it runs relatively faster than the STC and SMA kernels on GTX750 since it has 72% lower memory bandwidth than Titan. This trend comes more obvious on iGPU. The Titan GPU runs about 10.9X, 27.3X and 6.8X faster than iGPU for the STC, SMA and DA kernels. The iGPU in APU uses host DDR3 memory as its global memory, which suffers about 91% lower memory bandwidth than Titan and suffers the absence of data racing optimization. The SMA kernel possesses a lot of uncoalesced accessing and atomic memory operations, which makes the SMA kernel very slow on the GTX750 GPU.

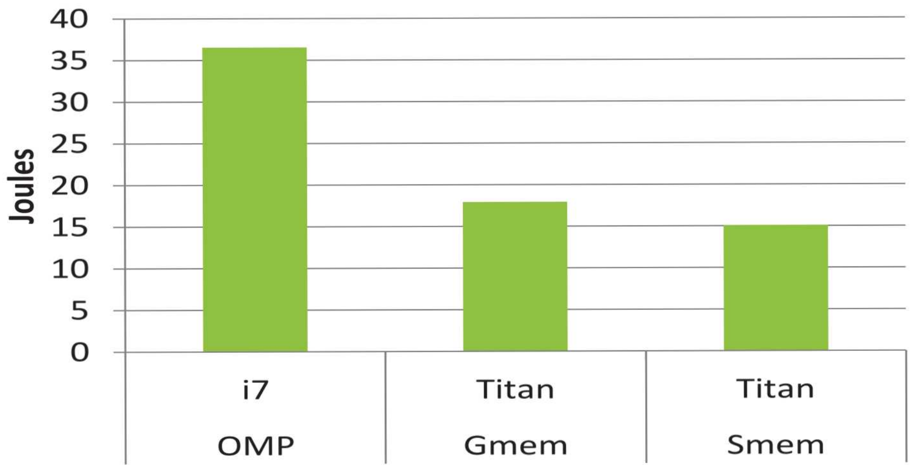
For energy, the Titan GPU uses 1.4X, 1.1X and 1.2X more energy than GTX750 for the STC, SMA and DA kernels. Similar trends can be found in the comparison between Titan and iGPU. Titan uses 49%, 85% and 74% less energy than iGPU for the STC, SMA and DA kernels. Since all the three kernels do not have enough arithmetic intensity that can fully use all GPU cores, the GPU with less cores like GTX750 would have lower energy usage and benefit more if the kernel has lower memory bandwidth requirement (e.g., the DA kernel) than Titan which has more GPU cores.

FPGA

We use FPGA as an alternative accelerator solution to GPU due to its low power features. The Nallatech FPGA card is equipped with a Stratix V D5 FPGA and mounted on the Intel i7 2600K system. The Altera OpenCL 13.1 SDK has been used to compile the OpenCL implementation of `SmemDA` and generate the FPGA configuration bitstream file. We also implemented STC, SMA and `Loop` (a compute intensive nested loop in STC) on the FPGA. The overhead of data transfer between host and FPGA is considered. Figure 13 shows the performance and energy data of FPGA in comparison with the Titan GPU. The FPGA implementation of DA is 281X slower and uses 76X more energy than DA on Titan. Because of the limited logic resource of the FPGA chip, only a small number of execution units of

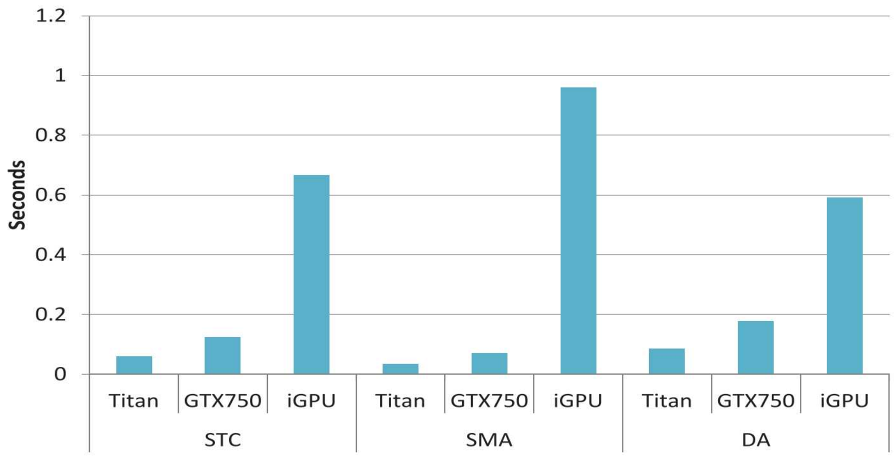


(a) Performance

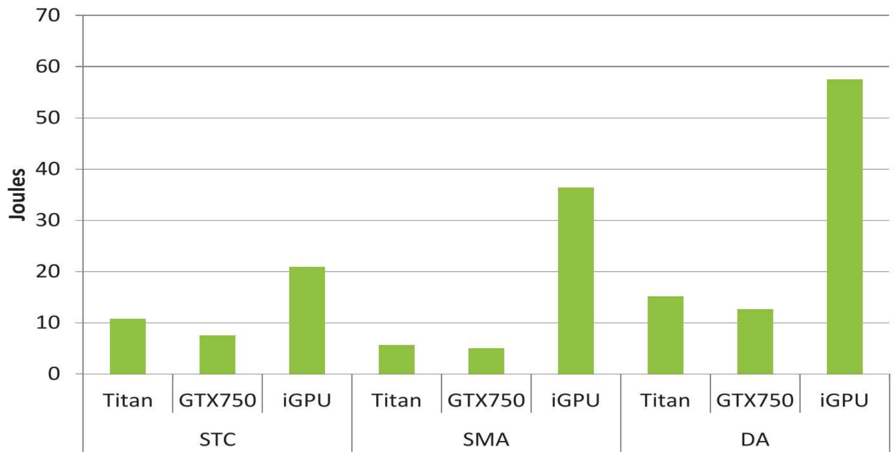


(b) Energy

Figure 11. Performance and energy of using different memory resources for data communication between the STC and SMA kernels.



(a) Performance



(b) Energy

Figure 12. Performance and energy of using GPUs for different kernels.

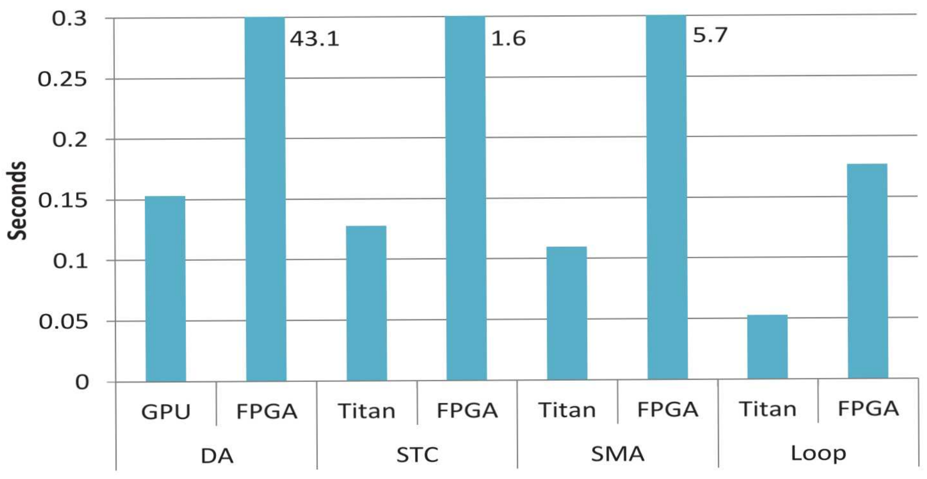
DA can be generated. A lot of logic resource has been used for generating the multipliers to perform real floating point multiplication. Also, PCIE controller and thread scheduler use a large amount of logic and memory resource. With less logic resource available to form basic execution units, the STC and SMA implementations run 12X and 52X slower than Titan, respectively. Though SMA uses less logic resource to form one execution unit than STC, the low FPGA memory bandwidth (on board DDR3) and the absence of atomic operation optimization in memory controller make SMA slower than STC on FPGA. `Loop` is only a small piece of code and uses much less logic resource for one execution unit. It can be seen that `Loop` performs 3.3X slower but with 15% less energy consumption than Titan.

Heterogeneous implementations

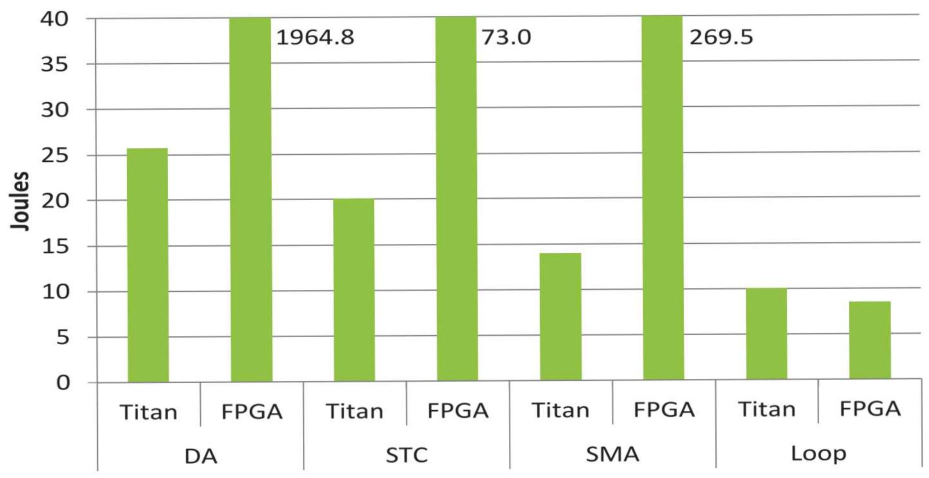
We now examine the performance and energy of using both CPU and GPU for DA. Specifically, we run one DP and two CP implementations (CP1 and CP2) of DA discussed before 4 on five heterogenous systems: i7+Titan, i7+GTX750, i3+Titan, i3+GTX750 and iCPU+iGPU (AMD A10-5800K APU). For DP, we divide the input data set into two subsets and distribute the two subsets to the CPU and GPU. We tune the CPU/GPU workload ratios on each system to make sure the CPU and GPU can start and terminate at the same time. The performance and energy results are shown in Figure 14. There is no clear performance winner among the three heterogeneous implementations on all the five heterogeneous systems. DP has the highest performance on i7+Titan, i3+Titan and i3+TX750 and only runs 9% and 4% slower than the fastest one on i7+GTX750 and iCPU+iGPU, respectively. Since DP has a balanced CPU/GPU workload distribution and both the CPU and GPU can be fully used, DP performs better than CP1 and CP2 on most systems.

Although CP1 and CP2 offload GPU friendly code onto the GPU, the system performance usually suffers from the unbalanced CPU/GPU workload distribution due to the fact that the CP workload distribution is unchangeable. Because of that i7 and GTX750 have smaller performance difference than the other systems and CP1 has heavier GPU workload distribution than CP2, CP1 performs the best on i7+GTX750. CP2 has higher data communication overhead between host and GPU than DP and CP1 due to the use of software pipelining and the specific implementation. Hence, the high iCPU+iGPU data communication speed makes CP2 the fastest implementation of DA on iCPU+iGPU.

For energy, it can be observed that DP consistently delivers the highest energy efficiency on all the discrete heterogeneous systems. This trend agrees with the performance trend except the i7+GTX750 system. DP runs 9% slower and saves 2% energy than CP1 on i7+GTX750. This performance and energy tradeoff comes from the wasted idle energy which is caused by the unbalanced CPU/GPU workload distribution of CP1 on i7+GTX750. On iCPU+iGPU, DP uses 20% and 9% more energy than CP1 and CP2, respectively. Two factors contribute to the low energy use of CP1 and CP2: (i) the integrated iGPU+iGPU has faster CPU and GPU communication speed; (ii) iGPU is much less powerful than discrete GPUs and benefits more from only running GPU friendly code.

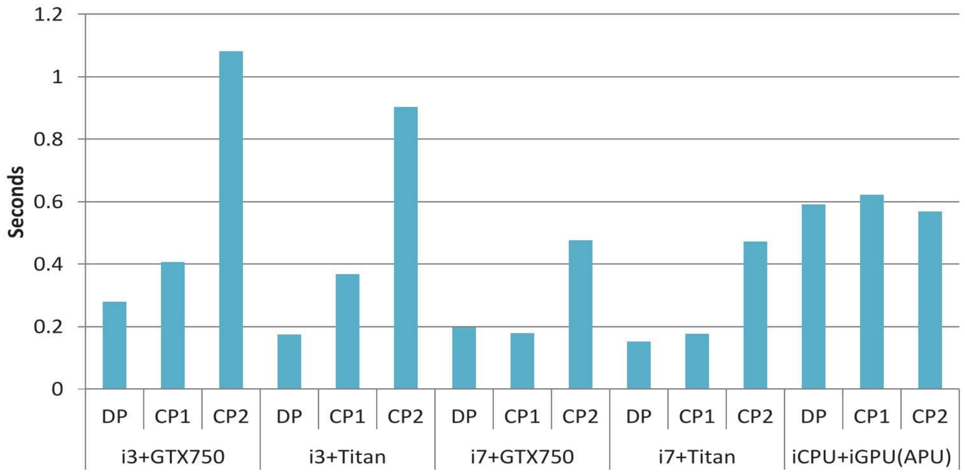


(a) Performance

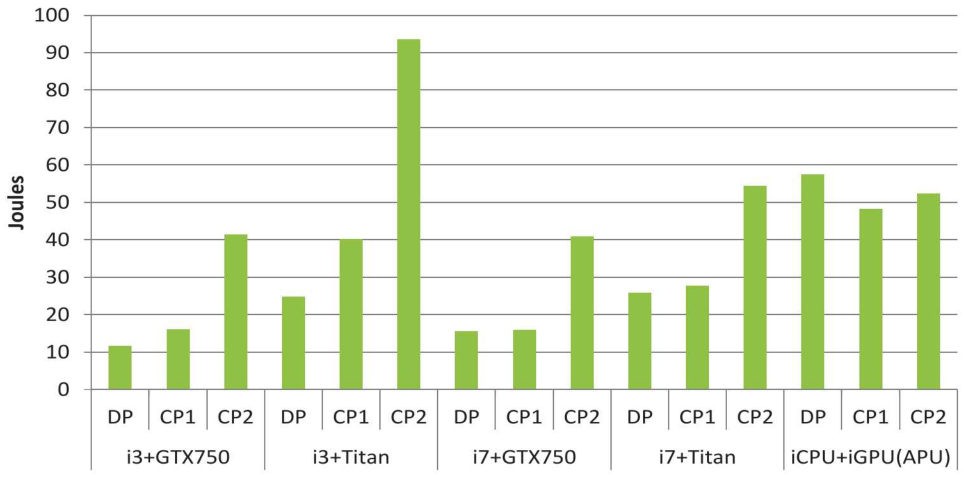


(b) Energy

Figure 13. Performance and energy of using FPGA with different offloading code.



(a) Performance



(b) Energy

Figure 14. Performance and energy of DP and CP implementations of DA on different heterogeneous systems.

Chapter 6

Summary

In this report, we studied the performance and energy impacts of hardware, application, and the interplay between the two for heterogeneous systems by using DA. We selected a wide range of CPUs and accelerators to form different heterogeneous systems. To use those heterogeneous systems, we used different approaches to implement DA on using CPU or GPU alone or using both. We also considered employing different programming models with the GCC and Intel compilers. The actual performance and energy results were obtained on real machines by using our developed energy measurement scheme. The results show that hardware choices, application implementations, and mapping of applications to hardware all heavily impact the performance and energy usage. Our data also indicates that there is no single mapping scheme of applications to hardware that can achieve best performance and energy efficiency on all heterogeneous systems.

DISTRIBUTION:

1	MS 0899	Technical Library, 9536 (electronic copy)

Sandia National Laboratories