A Low-Power Hybrid CPU-GPU Sort

Lawrence Tan

Mar 2014
CMU-CS-14-105

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
David G. Andersen, chair
Garth A. Gibson

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science

This research was sponsored by the National Science Foundation under grant numbers CNS-0619525 and CNS-0716287. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: CUDA, low power, Joulesort

Abstract

This thesis analyses the energy efficiency of a low-power CPU-GPU hybrid architecture. We evaluate the NVIDIA Ion architecture, which couples an Intel Atom low power processor with an integrated GPU that has an order of magnitude fewer processors compared to traditional discrete GPUs. We attempt to create a system that balances computation and I/O capabilities by attaching flash storage that allows sequential access to data with very high throughput.

To evaluate this architecture, we implemented a Joulesort candidate that can sort in excess of 18000 records per Joule. We discuss the techniques used to ensure that the work is distributed between the CPU and the GPU so as to fully utilize system resources. We also analyse the different components in this system and attempt to identify the bottlenecks, which will help guide future work using such an architecture.

We conclude that a balanced architecture with sufficient I/O to saturate available compute capacity is significantly more energy efficient compared to traditional machines. We also find that the CPU-GPU hybrid sort is marginally more efficient than a CPU-only sort. However, due to the limited I/O capacity of our evaluation platform, further work is required to determine the extent of the advantage the hybrid sort has over the CPU-only sort.

Contents

Chapter 1

Introduction

This thesis evaluates a novel low power CPU-GPU hybrid architecture as an energy efficient platform for data-intensive workloads. In our design and evaluation, the metric we target is reducing and measuring the amount of energy consumer per record sorted. We focus on sort because of its importance as a primitive in a variety of data-intensive workloads. The Joulesort metric measures the number of 100-byte records a system can sort per Joule of energy [19]. We build a Joulesort candidate and show that our candidate is almost twice as energy efficient as previous candidates.

For comparison, we evaluate a sort with a similar design that runs solely on the CPU, and evaluate the optimizations made to achieve power efficiency.

1.1 The Importance of Power

The total cost of ownership of computer datacenters is becoming increasingly dominated by power costs. Up to 50% of the three-year total cost of ownership of a computer is the cost of power and cooling. In addition, the number of machines that can be housed in a datacenter is bound by the amount of power and cooling that can be supplied to them. The power required is estimated at 10–20 kW of power per rack and up to 10–20 MW per datacenter [13]. The EPA estimates that in 2011, given current energy trends, datacenters will consume more than 100 billion kWh, costing roughly US$7.4 billion annually [24].

Previous work [2, 5, 16, 8, 22, 12, 14] has shown that even for CPU-bound workloads, using slower CPUs achieves higher energy efficiency even though the work may take longer to complete. The reason for

this is twofold. First, the frequency at which a CPU can be run is proportional to its input voltage. The power dissipated by the CPU in turn scales quadratically with the input voltage. Hence, a CPU that runs twice as fast dissipates four times as much power.

In addition, most modern high-speed processors dedicate a significant proportion of transistors to implement techniques such as branch prediction, speculative execution and increasing the amount of caching. These techniques do not increase the execution speed of basic instructions, but the increased transistor count increases the power consumption of these processors.

This is even more apparent in workloads that are IO-bound, where the CPU is not fully utilized. Power-proportionality is difficult to achieve even with techniques such as dynamic voltage and frequency scaling (DVFS) because of fixed power costs such as transistor leakage currents [6], and non-CPU components which consume a relatively constant amount of power [4]. These factors make it such that running a system at 20% load may still consume more than 50% of its peak power [23].

In the FAWN project we built a cluster that "couples low-power, efficient embedded CPUs with flash storage to provide efficient, fast, and cost-effective access to large, random-access data." [2] As a proof of concept, we also designed and built the FAWN-KV key-value store that ran on the FAWN cluster. This key-value store provides similar services as those used in industry today [7, 18, 15]. FAWN-KV maintained roughly the same level of performance while consuming an order of magnitude less energy.

This project extends this prior work in two ways. First, the workload targetted by FAWN-KV is a seek-based workload, where a large amount of data is stored on disk, and requests are made for data at random offsets within the data store. We examine a different workload where data is read sequentially off the disk and permuted in memory. We discuss optimizations for the workload to ensure maximum CPU utilization throughout the sort. Secondly, the FAWN project examines clusters built using low-power CPUs. We extend the work done in this regard by considering a novel CPU-GPU hybrid architecture.

The GPU is particularly interesting in this regard, because it devotes more transistors to data processing than a traditional CPU does as shown in Figure 1.1. It is suited for workloads where the same program is executed on many data elements—this reduces the amount of sophisticated control flow required, and also allows us to hide memory latency behind calculations for other elements instead of using large caches. A GPU is thus likely to be more energy efficient for the same amount of work, if the workload is amenable to its highly data-parallel processing model.

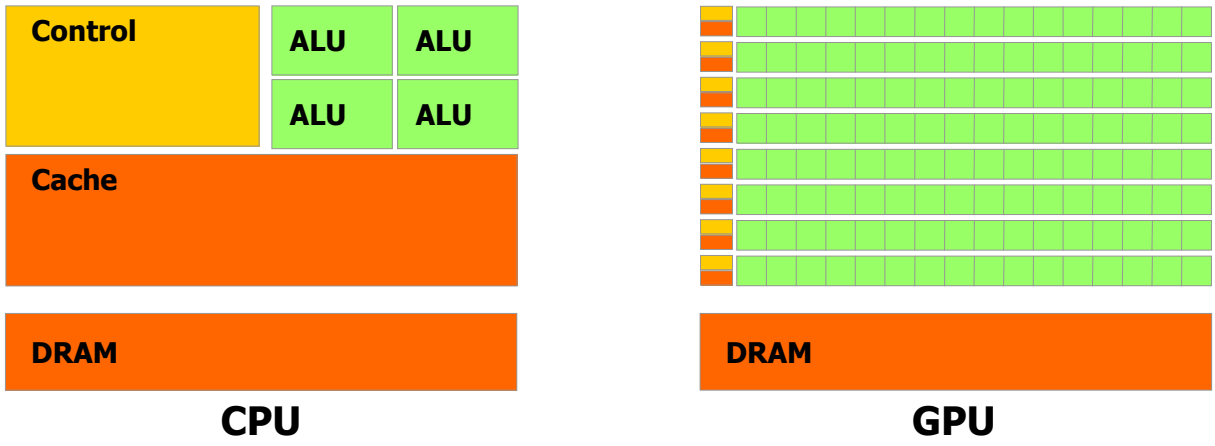

Figure 1.1: The GPU has more transistors devoted to data processing

1.2 Sorting As A Workload

Sorting is a popular workload in many benchmarks, from sort-specific benchmarks such as GraySort and PennySort [10, 11] to general-purpose benchmarks such as the NAS Parallel benchmark [3]. Reasons commonly cited are that the workload is easy to define, and that it stresses multiple components of a system. Sort is easily understood, and can be implemented on almost any computer architecture, so it provides us a basis by which to compare disparate systems. Sorting is also a very common primitive in data-intensive systems such as commercial databases. It is thus somewhat representative of workloads that access large amounts of data sequentially.

The Joulesort benchmark [19] is a well known benchmark that uses the number of sorted records per Joule to measure the black-box energy efficiency of a system. We use this benchmark to analyse our CPU-GPU hybrid architecture because it is easily understood and well known. In addition, the basic premise behind the benchmark—being able to perform sequential-I/O based workloads using as little energy as possible—reflects the type of workload we are trying to profile.

1.3 Main Results

We implemented a Joulesort candidate and benchmarked it against a dataset with 10^8 records. The candidate was almost twice as energy efficient as any other candidate previously submitted for the Joulesort

competition in 2007 - no newer candidates had been submitted as of the initial writing of this thesis.

Chapter 2

Background and Related Work

This section first provides an overview of work that has been done to build energy efficient systems. Next, we briefly explore approaches taken by previous Joulesort winners. Subsequently, we provide an overview of programming using the CUDA programming model. Last, we discuss previous work that has been done in implementing parallel sorts to run on GPUs (not necessarily using CUDA).

2.1 Building Energy Efficient Systems

The Joulesort paper proposed a concrete benchmark to allow researchers to compare the energy efficiency of disparate systems. The authors of this paper also built a balanced system comprising an array of 12 laptop disks and a low power (34 W) processor. This system achieved 3863 sorted records per Joule. Subsequent Joulesort entries have achieved only marginal improvements over the original Joulesort entry.

More recently, there were 2 papers that each proposed an architecture for energy efficient systems. The first such paper is the Gordon paper, which proposed pairing an array of flash chips and DRAM with low power CPUs for energy efficient, data-intensive computing. This system tries to achieve a balance between I/O and compute by using fast flash chips and DRAM with slower, more energy efficient CPUs. One of the primary focuses of this work was to develop a flash translation layer (FTL) that allows one CPU to work with multiple raw flash chips.

The FAWN paper also proposed using fast flash memory paired with low power CPUs to build a balanced system. The authors then designed a key-value store to run on a cluster of FAWN nodes. Experimental

evaluation showed that this key-value store could achieve the same level of performance (in terms of latency and throughput) as a traditional system using a tenth as much energy.

The workload studied in the FAWN paper was a seek-based workload, which flash memory is particularly suited to. In this paper, we extend the work done in the FAWN project by studying a different workload – sort – and extending the FAWN architecture by building a CPU-GPU hybrid machine.

2.2 Previous Joulesort Winners

2.2.1 CoolSort

The first Joulesort winner was CoolSort [19] in 2007. The focus on this paper was the choice of hardware to build an energy-efficient sorting system.

The authors first evaluated several "unbalanced" systems under a sort workload, which included 2 server-class machines and a modern laptop. In all 3 cases, they showed that the CPUs were highly underutilized and thus concluded that a system with more I/O would provide better energy efficiency.

They then built a system that paired a modern processor with a large number of regular disks in a disk array configured using LVM2. The number of disks used was chosen such that CPU utilization was maximized and I/O was no longer the bottleneck. By varying the I/O capacity of the system, the authors showed that a system that with sufficient I/O to saturate the CPU is the most power-efficient.

2.2.2 OzSort

The next Joulesort winner was OzSort [21] in 2009. This entry was different from CoolSort in that the authors both built the machine and wrote the sorting software that ran on it.

Similar to CoolSort, the OzSort authors found that it was important to have sufficient I/O to fully utilize available CPU cycles. They built a system with 7 harddisks because they found that fewer harddisks "could not offer enough bandwidth to be competitive". In addition, they ensured maximum bandwidth from the disk drives by placing data on the outer-rim of the disk and placing the 7 disks in a RAID0 configuration.

OzSort differentiated itself from CoolSort by optimizing their sort software to fully utilize the capabilities of their machine. OzSort is implemented as a 2-phase merge sort, which is divided into the Run phase and the Merge phase. In the Run phase, the input file is partitioned into individual "runs" such that each

"run" fits entirely in an input buffer that is about half the size of total RAM. The majority of the remaining memory is used for a prefetch buffer, with small output buffers and bookkeeping data structures occupying the remaining space.

Fitting an entire run in memory allowed for sorting to happen most efficiently since the sorting threads would only ever be blocked by the memory subsystem, which is orders of magnitude faster than physical disks. The prefetch and output buffers allow for disk I/O to be overlapped with in-memory processing. For instance, while a run is being sorted in memory, data belonging to the next run is read into the prefetch buffer.

In the merge phase, each sorted run is divided into several microruns (a fraction of a run), and the majority of main memory is filled by the next microrun from each run. A run prediction heap is used to store the last record of each microrun and the address on disk of that record so that the next few microruns that will likely be needed can be prefetched from the disk.

The design of the sort and the algorithms used are chosen such that data that will be needed by the sort is read into memory before the CPU would block on it. This is done to maximize the utilization of the CPU and thus to maximize the sort's power efficiency.

2.3 The CUDA Programming Model

CUDA [17] is NVIDIA's parallel computing architecture designed to run on NVIDIA's graphics processing units(GPUs). GPUs are specialized for compute-intensive, highly parallel computation, and are designed with more transistors devoted to data processing rather than caching or control flow, as was shown in Figure 1.1.

GPUs are particularly suited to problems where the solution requires executing the same piece of code on a large number of elements of data. A number of sorting algorithms can be adapted to run efficiently on a GPU, and we discuss briefly some work that has been done in implementing parallel sorts using GPUs.

2.3.1 Mapping CUDA To Hardware

A graphics processing unit consists of a number of "multiprocessors", and each multiprocessor consists in turn of 8 "stream processors" and a number of special function units. Each multiprocessor works in the

Single Program Multiple Data (SPMD) model where all 8 stream processors execute the same instruction on different pieces of data. Each multiprocessor can execute multiple threads simultaneously.

Logically, threads are divided into thread blocks which can be assigned to a specific multiprocessor. A thread block is required to execute independently of all other blocks, which means that a block can be executed by any multiprocessor in any order. This allows for easy load balancing where a block can be assigned to any idle multiprocessor in the system.

Threads within a block are executed warp by warp, where each warp contains 32 threads of a particular block. All threads in a warp execute a common instruction simultaneously. Full performance is thus realised if all threads in the same warp take the same branches during execution. If one or more threads diverge in a branch, the warp serially executes all possible paths, and only reverts back to full performance when the paths converge.

Synchronization of threads in a thread block is achieved using a barrier primitive.

An illustration of the CUDA programming model is shown in Figure 2.1.

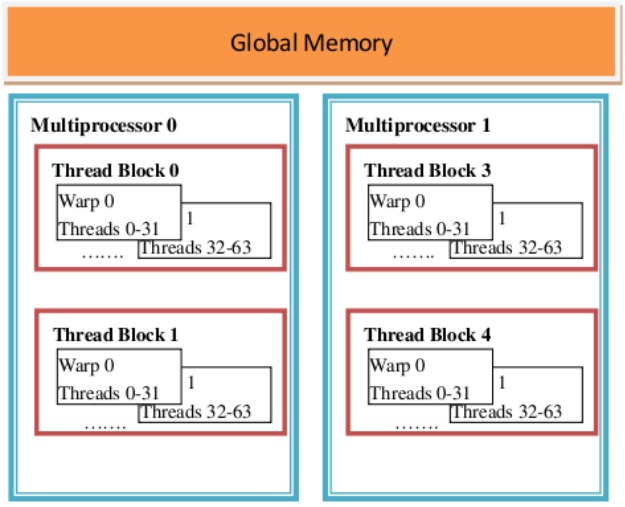

Figure 2.1: CUDA Programming Model

2.4 Parallel Sorting on GPUs

There has been recent interest in sorting algorithms optimized to run on GPUs. A large amount of this interest has been generated since the introduction of general-purpose computing on graphics processing

units (GPGPU) frameworks such as CUDA and the open industry standard Open Computing Language (OpenCL) framework. These frameworks define language extensions and APIs that allow programmers to write code using an abstract multithreaded machine model that can be compiled to run on several GPU and CPU architectures.

2.4.1 GPUTeraSort

GPUTeraSort [9] is a sort algorithm developed in 2005 that uses a GPU as a co-processor to accelerate the sorting of billion-record wide-key databases. This work was done before the introduction of CUDA, and hence it was necessary for them to map GPU processing primitives to sorting operations while writing their sort. Although this was not the first paper to propose GPU-based sorting algorithms, previous algorithms described were limited to datasets that fit entirely within GPU memory.

This article observed that external memory sort performance (where the dataset is too large to fit into system memory) is limited by the traditional Von Neumann-style architecture of the CPU. The CPU and main memory bottleneck is traditionally accounted for by using several level of caches, but CPU-based sorting algorithms incur significant cache misses on large datasets. The authors further note that GPUs have significantly higher memory bandwidth and also achieve an order of magnitude more operations per second by exploiting data parallelism.

In their work, they developed a hybrid sorting architecture that "offloads compute-intensive and memory-intensive tasks to the GPU". The sorting architecture maps a bitonic sorting network to GPU rasterization operations. As mentioned earlier, this mapping is necessary because the authors were not able to use a GPGPU framework when working with the GPU.

2.4.2 Radix Sort on GPUs

A newer paper co-authored by NVIDIA engineers described the design of radix sort and merge sort for GPUs using CUDA [20]. Our implementation of radix sort is based upon this implementation of radix sort, which is bundled as an example application in the CUDA SDK.

This paper introduced efficiency considerations when designing algorithms for GPUs. In their words, these efficiency considerations are "much like cache line sizes on traditional CPUs", which can be ignored for correctness but must be carefully considered when designing for performance. For instance, a warp of

threads is executed with all threads executing the same instruction simultaneously. If threads within the same warp take different execution paths because of a branch, the different paths are executed serially, and all threads begin executing simultaneously again only when these paths converge. Hence, to maximize performance, threads within the same warp should avoid execution divergence. Another example is that threads are permitted to load from and store to any valid address. However, when threads within the same warp access consecutive words in memory, the hardware coalesces these accesses into aggregate transactions, resulting in much higher memory throughput.

The paper then describes an implementation of radix sort, and explains how the efficiency considerations mentioned above are accounted for in their implementation.

Chapter 3

Design

Because our dataset is too large to fit in memory, we adopt a 2-pass approach similar to the approach described in OzSort [21]. The 2-pass CPU-GPU hybrid sort uses the first pass to divide the input data into buckets that each fit entirely in GPU memory. The second pass then reads each bucket in and sorts them independently of all other buckets. In the second pass, we use a modified version of the radix sort provided with the CUDA SDK, and a detailed description of this sort can be found in the appendix.

3.1 Design Goals and Requirements

We discuss our design goals and requirements in conjunction with the limitations of our hardware. By doing so, we can more clearly see the design decisions made later on so as to achieve maximum performance. We provide brief details of the hardware used in Table 3.1.

Component	**Hardware**
CPU	Intel Atom N330 (1.6GHz, dual core)
Chipset/GPU	NVIDIA Ion (512MB shared memory)
RAM	4GB DDR2-800 (3GB addressable by CPU)
Disk	4 $\times$ Intel X25-M SSD (250MB/s read, 70MB/s write)

Table 3.1: Hardware description

The NVIDIA Ion GPU does not have discrete memory. Instead it shares a configurable amount of main memory that ranges from 128MB-512MB. A beneficial side effect is that it becomes possible for

both the CPU and the GPU to address the same region of memory. This allows data to be accessed by both without time-consuming copying or being bound by system bus throughput. However, the amount of mapped memory that can be allocated is bound by the amount of memory the GPU can address.

Given that power draw is not proportional to load, we aim to operate all components at peak load to achieve maximum energy efficiency. In addition, our choice of hardware imposes certain constraints on our design. For instance, to achieve maximum throughput from our SSDs, we must write sequentially to the drives, preferably in chunks that are a multiple of the erase block size. Our design should achieve the following goals:

- Minimize the amount of time waiting on I/O
- Keep the GPU as busy as possible
- Maximize I/O throughput - since the dataset does not fit entirely in main memory, it is necessary to perform a 2-pass sort. The effective throughput of a 2-pass sort is $\frac{1}{4}$ the aggregate I/O throughput of the system.

3.2 2-pass External Sort

As mentioned earlier, it is necessary to perform a 2-pass sort given the size of our dataset. In our sort, the first pass buckets the data into a set of files that each fit entirely in GPU memory. These files are strictly ordered, such that all keys in the first file are strictly smaller than keys in the second file, which are in turn strictly smaller than keys in the third file and so on. The second pass of the sort then uses the radix sort described in the appendix to sort each bucket independently.

3.2.1 Pass One

In the first pass, we bucket all input key-value pairs by the value of the first n bytes of the key. These buckets are written back to disk as they are filled up. We choose n such that all buckets fit entirely in GPU memory. In our experiment, with 10^8 records consisting of 10-byte keys coupled with 90-byte values, using $n = 1$ produces 95 buckets, each approximately 100MB large. This is illustrated in Figure 3.1.

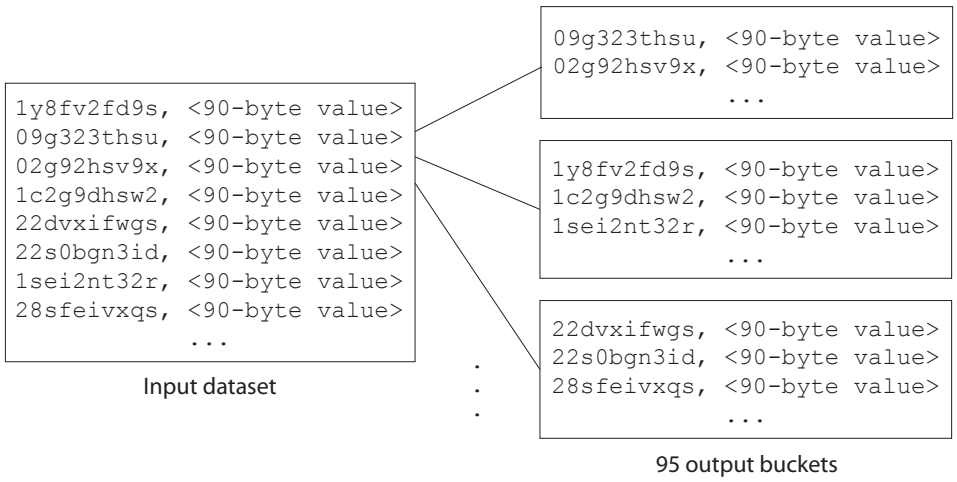

Figure 3.1: Pass 1

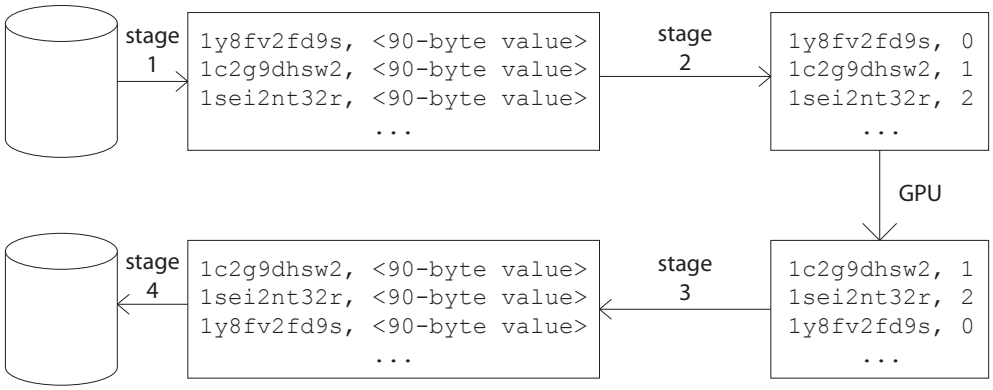

Figure 3.2: The CPU-GPU hybrid sort pipeline

3.2.2 Pass Two

The second pass reads the buckets produced in the previous step one at a time, sorts the buckets and writes the sorted bucket back to disk. To achieve maximum throughput, we execute the second pass in a 4-stage pipeline so that sorting can occur simultaneously as buckets are being read and written to and from disk. Each stage of the pipeline is executed by a thread, and the output of each stage of the pipeline except the last is buffered in a queue. Our 4-stage pipeline is illustrated in Figure 3.2. The stages in the pipeline are:

1. **Input:** The input stage reads the contents of an unsorted bucket from disk to main memory.

2. **Sorting:** In the sorting stage, we first generate a list of (key, index) pairs. This avoids the need to

send the large 90-byte value along with the much smaller 10-byte key to the GPU. Instead, we send a 4-byte index that later allows us to construct a sorted list of key-value pairs. This allows us to conserve usage of scarce mapped memory.

The list of (key, index) pairs is stored in mapped memory that both the CPU and the GPU can access with no copying required. We sort the list of (key, index) pairs using our GPU radix sort mentioned above. Note that the implementation of the radix sort used only allows us to sort keys that are a multiple of 4-bytes long.

The indices in the sorted list of (key, index) pairs tells us how to permute the original list of key-value pairs such that the keys are in sorted order.

3. **Permutation:** This stage uses the permutation generated in the previous stage that is stored in mapped memory, and copies key-value pairs out of the original input in the correct order to construct a sorted list of key-value pairs. This speed of this operation is likely to be bound by memory bandwidth, because key-value pairs in the original input are accessed in a random order, and caches are unlikely to help.

4. **Output:** This stage writes the sorted list of key-value pairs generated in the permutation stage out to disk.

3.3 Implementing the Pipeline

To implement the pipeline for the second pass described in the previous section in a simple and efficient way, we used the Intel Threading Building Blocks (TBB) library. This library allows us to create a pipeline by implementing a series of pipeline filters which process data in some way. The filters are then chained together to form our pipeline[1].

In our implementation, we created 4 separate filters corresponding to the 4 stages of the pipeline. All stages except for the sorting stage can be parallelized—multiple threads can read and write to disk at the same time, and buckets can be permuted independent of each other. The only stage that has to be executed in series is the sorting stage because the GPU can only sort one bucket at a time.

3.4 Optimizing I/O Performance

A significant technical challenge encountered in this project was fully exploiting the I/O capabilities of the system while working around the idiosyncracies of flash memory.

GPUTeraSort [9] provides an analysis of the effective throughput of a p-pass sort as a function of the aggregate disk throughput of the entire system. In particular, a two-pass sort such as the one we perform can achieve at most $\frac{1}{4}$ the aggregate disk throughput of the entire system because every item must be read twice from disk, and written twice to disk.

Instead of using multiple threads to try to read and write files to and from multiple disks at the same time, we built 2 RAID 0 arrays, each containing 2 of the 4 SSDs attached to our system. In the first pass, we read the input data from one RAID array and write the intermediate output to the other RAID array. The second pass then reads that data back in and writes the final output to the first RAID array.

When we bucket the records in the first phase, it is important first of all that each bucket fits entirely in main memory so that the data can be sorted without ever being blocked on disk accesses. In addition, we theorize that changing the bucket size will affect the efficiency of our sort even if all bucket sizes are smaller than the amount of available memory. For instance, if our buckets are unnecessarily large, we pay a penalty at the start of the sort where we are blocked waiting for at least one bucket to be fully loaded into memory.

Chapter 4

Experimental Evaluation

4.1 Hardware Setup

The hardware specification of our NVIDIA Ion based machine is listed below:

- **CPU:** Intel Atom 330 (1.6GHz Dual Core, 45nm, 8W TDP, released in 2008)
- **GPU:** NVIDIA MCP7A-ION
- **RAM:** 4GB DDR2-800
- **Main Storage:** 3 $\times$ 80GB Intel X25-M SSD, 1 $\times$ 160GB Intel X25-M SSD

The peak power consumed by the system as measured at the wall plug at any time in our experiments is 32W, and the system typically draws about 23W when idle. Power consumed was measured using a "Watts Up?" electricity consumption meter. This meter displays the instantaneous power consumption. We sampled the value displayed every second to calculate the total energy draw of each run.

4.2 Experimental Evaluation

We used the Joulesort benchmark described in the preceding section of measure the energy efficiency of our architecture. Joulesort benchmark rules require that the sorted output file is created as part of the sort and is written to secondary storage (in our case, the SSDs). The 2-pass algorithm we used also requires us to write

out temporary files, each containing one sorted bucket of data, to the SSDs. Since Joulesort rules require that each record is 100 bytes large and the limited amount of storage available, we only ran our Joulesort benchmark on an input set with 10^8 records.

In each run, we measured the total running time and recorded the power draw every second using a watt meter. We assume that the power draw displayed on the watt meter represents the average power drawn in the past second. Note that we do not record the total energy consumed directly on the watt meter because of its limited granularity (energy consumed is measured in tenths of a kWh).

In our experimental evaluation, we ran the sort benchmark on our dataset of 10^8 records 3 times, and calculated the average amount of time taken to sort the records. We also sum up all readings recorded from the watt meter, and we average the total number of Joules across the 3 runs.

3 runs of our sort benchmark took an average of 143.8s per run to complete. Each run consumed an average of 4600J. This works out to an energy efficiency of approximately 21700 sorted records per Joule.

According to Intel's datasheets, our SSDs have a maximum sequential read bandwidth of 250MB/s, but a maximum sequential write bandwidth of only 70MB/s. As mentioned above, in a 2-pass sort, the theoretical upper bound of the sort throughput due to I/O limitations is $\frac{1}{4}$ the aggregate disk bandwidth. Since our system has an aggregate write bandwidth of 280MB/s, the theoretical upper bound of our sort is 70MB/s. In our experiments, we find that our sort has an effective throughput of approximately 66MB/s which is very close to the theoretical upper bound. We thus believe that our sort is I/O-bound. In addition, the CPU is never fully utilized at any point in the sort. The motherboard used in this experiment only has 4 SATA ports so we were unable to connect additional SSDs to increase the aggregate bandwidth of our system.

We show in Figure 4.1 the average amount of time it took to complete each stage of the pipeline in the second pass.

4.2.1 Choice of Sorting Algorithm

To convince ourselves that radix sort was in fact the correct choice for our application, we ran a simple benchmark to compare our GPU radix sort, a GPU quick sort and radix sort on the CPU. The results of this benchmark are summarized in Table 4.1. We can see that our implementation of radix sort for the GPU runs faster than both our implementation of quicksort for the GPU and radix sort for the CPU. This is expected since the Joulesort workload consists of short (10 byte) keys that are randomly distributed through-

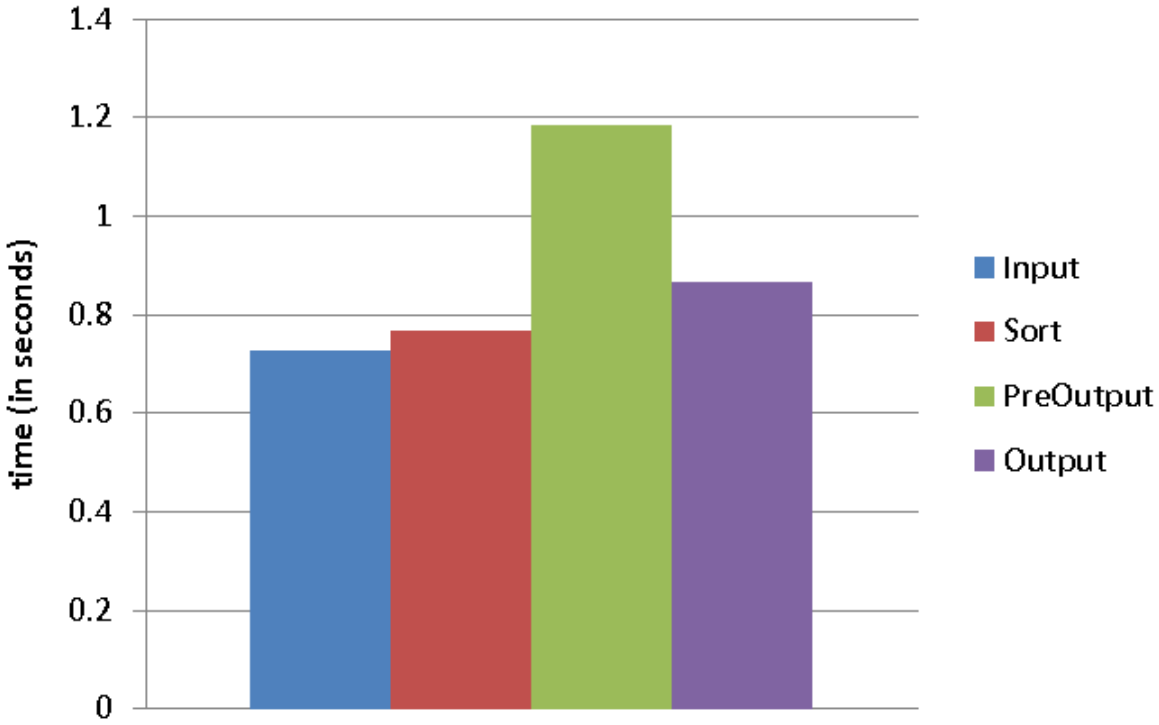

Figure 4.1: The performance of each stage in the pipeline of the second pass

out the entire keyspace. Radixsort has worst case performance $O(kN)$, where k is the length of the key. Comparison-based sorts such as quicksort run in $O(NlogN)$ time in the worst case. Since k does not grow as N grows, radixsort will always eventually perform better than quicksort.

No. of elem	GPU Radix-sort	GPU Quick-sort	CPU Radix-sort
2^{20}	151	205	160
2^{21}	305	400	332
2^{22}	670	789	659
2^{23}	1213	1614	1314

Table 4.1: Comparison of different sorts

4.3 Evaluation of CPU-only sort

We ran a sort on a comparable platform[1], but using radix sort on the CPU for the second phase. In this sort, we retained the same 2-pass approach, but we replaced the sorting and the permutation steps with a CPU-based radix sort.

4.3.1 Hardware and Software Setup

The hardware specification of our NVIDIA Ion based machine used for this test is listed below:

- **CPU:** Intel Atom D525 (1.80GHz dual-core, 45nm, 13W TDP, released 2010)
- **Chipset:** Intel NM10
- **GPU:** NVIDIA MCP7A-ION
- **RAM:** 2GB DDR2-667
- **Main Storage:** 1 $\times$ 160GB Intel 320 SSD, 1 $\times$ 120GB Intel 320 SSD

[1]The CPU-only sort was evaluated more than a year after the initial work was done, and an identical platform was no longer available for this test.

We used the radix sort included with Intel's Integrated Performance Primitives (IPP); specifically, we used the ippsSortRadixIndexAscend_32s function twice in our sort. We used the same watt meter as before, and the same methodology to measure the power draw and the energy consumed in each run.

4.3.2 Evaluation

The CPU-only sort took an average of 157s per run to complete, and consumed an average of 4730J per run. This works out to an energy efficiency of approximately 21140 sorted records per Joule.

We also repeated the CPU-GPU hybrid sort on this platform. The hybrid sort took an average of 146s per run to complete, and consumed an average of 4530J per run. This works out to an energy efficiency of approximately 22075 sorted records per Joule, which is just over 4% better than the CPU-only sort.

4.3.3 Discussion

We note that the performance of the CPU-GPU hybrid sort was not significantly better than that of the CPU-only sort. If we calculate the throughput of the 2 sorts, the hybrid sort has a throughput of approximately 65M/s and the CPU-only sort has a throughput of approximately 60M/s.

The aggregate write bandwidth of the 2 SSDs is 295MB/s. The theoretical upper-bound on our 2-pass sort throughput is $\frac{1}{4}$ of that, or 73.75M/s. Both the hybrid sort and the CPU-only sort are close to the theoretical maximum throughput of a system with brand-new drives, and it is likely that the drives in the test system which have some wear on them have slightly lower throughput.

In addition, the CPU never shows 100% utilization during either run, and this is a good indication that both tests are I/O bound. It is likely that using more disks or faster disks would allow us to see a larger difference between the two sorts.

Chapter 5

Conclusions and future work

Our hybrid CPU-GPU sort achieves almost twice the energy efficiency of even the most recent winning Joulesort entry. We believe that this shows that it is important to ensure a good balance between compute and I/O when designing a computer architecture. Traditional computers have significantly more compute than I/O capacity with extremely powerful CPUs paired with large but slow mechanical disks. Balancing compute and I/O can be achieved by using low-power processors paired with sufficient SSDs to saturate them.

The use of low power processors (both for the CPU and the GPU) contributes to the high power efficiency of the system even though the sort may take a longer time to complete on our system compared to a traditional machine.

In our comparison of the hybrid CPU-GPU sort against the CPU-only sort, we found that the hybrid sort was marginally more efficient than the CPU-only sort. Although this result is encouraging, both the hybrid CPU-GPU sort and the CPU only sort are I/O bound. An evaluation platform with more I/O will be needed to determine the extent by which the hybrid sort is more efficient than the CPU only sort.

In the future, we hope to replace our MLC X25-M disks which offer 250MB/s read throughput and 70MB/s write throughput with SLC X25-E drives, which offer the same read throughput, but 170MB/s write throughput. With these faster drives, we believe that we may be able to cause our workload to be bottlenecked by the amount of compute available as opposed to the limited I/O throughput we achieve with the current system. Another limitation of our current system is the limited number of drives we can connect

to the machine. If it turns out that we are unable to saturate the available compute with 4 X25-E drives, we will look for an alternative machine that allows for more drives to increase throughput.

Lastly, with the trend of disk sizes increasing we hope to be able to process 10^9 and 10^{10} records using our current architecture and be able to give a more complete evaluation of our system.

Appendix A

Radix Sorting on GPUs

Unlike GPUTeraSort [9], we use an implementation of radix sort instead of quicksort on the GPU because our sort is built specifically for short keys. Given n input records with keys of size k, radix sort takes only $O(nk)$ time, whereas quicksort takes $O(nlogn)$ time.

We based our implementation on that discussed in [20], which is provided as part of the NVIDIA CUDA SDK, and modified it for our purposes.

The sequential version of the radix sort is illustrated in Figure A.1; the radix sort consists of two primitives:

1. mapping keys into buckets (computing the array "count")

2. prefix summing (computing the array "offset")

To fit the CUDA programming model, we divide the array of key-index pairs into tiles that will be assigned to p thread blocks. Suppose the length of the key is d. We divide the radix sort into d/b passes such that we sort b-bits of the key in each pass. Each block has an independent array of 2^b-entry buckets.

A rough description of the parallel radix sort algorithm is given below:

- Each block loads the tile assigned to it and sorts it using 1-bit radix sort. Use each entries position in the block to construct the 2^b-entry digit histogram.

- Perform a prefix sum over $p \times 2^b$ histogram table to compute a global offset.

```
d <- number of bits in key
b <- number of bits sorted in each pass
For i From d To 0 Step by -b do
  Set count to zero
  Set offset to zero
  For every index[j] do
    w <- key[index[j]][i-b+1...i]
    count[w] <- count[w]+1
  End for
  For k From 1 To 2^b do
    offset[k]<-offset[k-1]+count[k-1]
  End for
  For every index[j] do
    w <- key[index[j]][i-b+1...i]
    index'[offset[w]] <- index[j]
    offset[w] <- offset[w]+1
  End for
  index <- index'
End for
```

Figure A.1: Serialized radix sorting

- Each block rewrites the index to its correct output position according to the global offsets.

A.1 Handling Wide Keys

The time complexity of radix sort increases linearly with the key width. With sufficiently wide keys, it may be more efficient to use comparison based sorts such as quick sort since these run in $O(nlogn)$ time. We have determined experimentally that for our keys, which are effectively at most 9 bytes long, our implementation of radix sort is more efficient than our quick sort implementation.

Bibliography

[1] Working on the assembly line: pipeline. `http://software.intel.com/sites/products/documentation/doclib/tbb_sa/help/tbb_userguide/Working_on_the_Assembly_Line_pipeline.htm`. URL retrieved Sep 2013.

[2] D. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, Oct. 2009.

[3] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L.Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS Parallel Benchmarks. Technical Report RNR-94-007, National Aeronautics and Space Administration, Mar. 1994.

[4] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, 2007.

[5] A. M. Caulfield, L. M. Grupp, and S. Swanson. Gordon: Using flash memory to build fast, power-efficient clusters for data-intensive applications. In *14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*, Mar. 2009.

[6] P. de Langen and B. Juurlink. Trade-offs between voltage scaling and processor shutdown for low-energy embedded multiprocessors. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, 2007.

[7] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. 21st ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2007.

[8] Dell XS11-VX8. Dell fortuna. http://www1.euro.dell.com/content/topics/topic.aspx/emea/corporate/pressoffice/2009/uk/en/2009_05_20_brk_000, 2009.

[9] N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: High Performance Graphics Coprocessor Sorting for Large Database Management. In *Proceedings of the ACM SIGMOD Intl. Conference on Management of Data (SIGMOD)*, 2006.

[10] J. Gray. Sort benchmark home page, Oct. 2006. http://research.microsoft.com/barc/SortBenchmark.

[11] J. Gray, J. Coates, and C.Nyberg. Performance / price sort and pennysort. Technical Report MS-TR-98-45, Microsoft, Aug. 1998.

[12] J. Hamilton. Cooperative expendable micro-slice servers (CEMS): Low cost, low power servers for Internet scale services. http://mvdirona.com/jrh/TalksAndPapers/JamesHamilton_CEMS.pdf, 2009.

[13] R. H. Katz. Tech titans building boom. IEEE Spectrum, Feb. 2009.

[14] K. Lim, P. Ranganathan, J. Chang, C. Patel, T. Mudge, and S. Reinhardt. Understanding and designing new server architectures for emerging warehouse-computing environments. In *International Symposium on Computer Architecture (ISCA)*, June 2008.

[15] Memcached. A distributed memory object caching system. http://www.danga.com/memcached/.

[16] Microsoft Marlowe. Peering into future of cloud computing. http://research.microsoft.com/en-us/news/features/ccf-022409.aspx, 2009.

[17] NVIDIA. NVIDIA CUDA Programming Guide. http://www.nvidia.com/object/cuda_get.html.

[18] Project Voldemort. A distributed key-value storage system. http://project-voldemort.com.

[19] S. Rivoire, M. A. Shah, P. Ranganathan, and C. Kozyrakis. JouleSort: A balanced energy-efficient benchmark. In *Proc. ACM SIGMOD*, June 2007.

[20] N. Satish, M. Harris, and M. Garland. Designing Efficient Sorting Algorithms for Manycore GPUs. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2009.

[21] R. Sinha and N. Askitis. Ozsort: Sorting 100gb for less than 87kjoules. http://sortbenchmark.

org/OzJoule2009.pdf, 2009.

[22] A. Szalay, G. Bell, A. Terzis, A. White, and J. Vandenberg. Low power Amdahl blades for data intensive computing, 2009.

[23] N. Tolia, Z. Wang, M. Marwah, C. Bash, P. Ranganathan, and X. Zhu. Delivering energy proportionality with non energy-proportional systems – optimizing the ensemble. In *Proc. HotPower*, Dec. 2008.

[24] U.S. Environmental Protection Agency. Report to Congress on Server and Data Center Energy Efficiency. http://www.energystar.gov/ia/partners/prod_development/downloads/EPA_Datacenter_Report_Congress_Final1.pdf.