GPU-Accelerated Document Filtering

Gordon Reid - 1002536r

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Level 4 Project — March 28, 2014

Abstract

Document filtering is a problem of growing importance. The growth of the Internet has resulted in a massive surge in the transfer and storage of data and information. A prime example is email where hundreds of billions of messages are sent every day. Emails, and other kinds of documents, can sometimes need to be filtered. This problem is highly data parallel in nature as the classification of one document does not affect the classification of another. This report investigates the viability of using GPUs rather than CPUs for the classification. GPUs are designed with parallel problems in mind and thus have the potential to allow for significantly higher throughput compared with CPUs.

The results show that a CPU and single GPU system can classify documents from a solid state drive or a 15Gbit/s Ethernet network feed in real-time. In future work, investigations of performance scaling with multiple consumer class GPUs will be conducted, with the aim of being able to parse and score documents from a 100Gbit/s Ethernet link.

Acknowledgements

I would like to thank my project supervisor, Dr. Wim Vanderbauwhede, for his extensive guidance, knowledge, and support throughout the project.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: Gordon Reid Signature: Gordon Reid

Contents

1 Introduction **1**

1.1 Introduction 1

1.2 Problem Description 1

1.3 Motivation 2

1.4 Related Research 2

1.5 Report Structure 2

2 Background **3**

2.1 OpenCL 3

2.1.1 Language Overview 3

2.1.2 Models 3

2.2 Device comparisons 6

2.2.1 CPU and GPU 7

2.2.2 Bridging the gap - APUs 8

2.2.3 Intel Xeon Phi 9

2.3 Document Classification 10

2.3.1 Parser 10

2.3.2 Scorer 11

3 System Implementation **14**

3.1 Base Implementation 14

3.1.1 Both Implementations 15

3.1.2 Scoring Only Implementation 16

3.1.3 Parsing and Scoring Implementation . . . 18

3.2 Optimisations . . . 19

3.2.1 Local Size . . . 19

3.2.2 Mapped Buffers . . . 22

4 Experiments **23**

4.1 Devices Used . . . 23

4.1.1 Mac Pro System . . . 23

4.1.2 Intel Xeon Phi System . . . 23

4.1.3 AMD System . . . 24

4.2 Experiment Set up and Input Files . . . 24

4.2.1 Data Transfer . . . 24

4.2.2 Document Classification . . . 25

4.3 Data Transfer Results . . . 26

4.4 Scoring Only Results . . . 26

4.5 Parsing and Scoring Results . . . 28

4.6 Discussion of Results . . . 30

4.6.1 Data Transfer . . . 30

4.6.2 C++ versus OpenCL CPU throughput . . . 31

4.6.3 Effect of Bloom Filters . . . 31

4.6.4 Overall . . . 33

5 Conclusion **34**

5.1 Summary of Results . . . 34

5.2 Future Work . . . 34

Appendices **36**

A Example TREC Document **37**

Bibliography **39**

Chapter 1

Introduction

1.1 Introduction

With traditional CPUs experiencing slowdowns in improvements to clock speed due to thermal and power limitations for a single core, the computing industry is looking for an alternative hardware and/or software paradigm to take us into the next era of computing performance. This has led to the increase in the number of cores on a single CPU die, with most consumer devices containing four cores. This allows four individual threads of control to run simultaneously on a single chip. In addition to this, the growth of computational power available in GPUs has led to the idea of using GPUs for general purpose computing tasks (GPGPUs). This is known as Heterogeneous Computing and refers to any system that uses more than one kind of processor, for instance a CPU and a GPU. In addition to this, there are growing numbers of devices with a Heterogeneous System Architecture (HSA) where multiple processor types (such as CPUs and GPUs) exist on the same die.

1.2 Problem Description

The primary task of the project was to take existing document filtering code running on a CPU and investigate the use of OpenCL to allow the code to be executed on multiple architectures, for instance a GPU, to improve overall performance of the system.

Document filtering can place documents into two categories, with the canonical example being spam or not spam. Plain text documents are given to a parser which generates a collection of terms for each document. This collection is given to a scorer which works out each document's score. For each term in a document, it is looked up in a profile which contains the significant terms. Terms in a document that are typically detected in a spam document will have a score assigned to them in the profile, it is this score which is collected and added to the document's overall score. At the end of scoring, if the document has a high enough score, it is classed as spam, otherwise it is not spam. This works for any two classification possibilities. For example, using the Text Retrieval Conference (TREC [1]) data collection, the *not spam* class could be Entertainment articles, and the *spam* class could be Financial articles.

[1] `http://trec.nist.gov`

1.3 Motivation

Document filtering is problem of growing importance. The advent of the Internet has resulted in a massive surge in the transfer and storage of data and information. A prime example is email where hundreds of billions of emails are sent every day. Emails, and other kinds of documents, sometimes need to be filtered. This problem is highly data parallel in nature as the classification of one document does not affect the classification of another. With GPU manufacturers concentrating on parallelism in their chip designs for pixel colour value generation, these devices have a suitable architecture for other highly parallel tasks.

In addition to the large amounts of data which needs to be filtered, there is a growing need for large enterprise data centres to be more cost effective, and to use less energy. A heterogeneous system can not only improve performance in raw terms, but also improve performance when measured against power and cost of equipment. These metrics are typically quoted as performance per watt and performance per dollar. A slower, cheaper system could be the preferable option over a fast, expensive system if the former system is more efficient to buy and run, informally referred to by the idiom 'more bang for the buck'.

1.4 Related Research

Document classification is an active area of research, with a number of papers regularly dedicated to the creation, analysis, and tuning of filtering algorithms. A large number concentrate on Naïve Bayes classification [AKC+00] [APK+00] with different statistical models (for instance multi-variate Bernoulli model [Sch03] or a multinomial model [MN+98]) with a focus on effectiveness rather than performance.

There has also been work looking into the efficiency of different architectures in the area of information filtering [CS12] [HLL13] which found that GPUs had the highest performance, but FPGAs had the highest performance per watt.

This report more specifically relates to, and builds upon, work by Wim Vanderbauwhede et al. which has primarily focused on the use of Field Programmable Gate Arrays (FPGAs) to accelerate a document parser and scoring system [VFA+13] [Van13] [CMV+12] with significant success. Details of the work in these papers is discussed throughput this report.

1.5 Report Structure

The following sections of the report will detail and discuss the problem of efficient heterogeneous implementations for document filtering using OpenCL and will compare results between devices, and to a traditional CPU implementation of the same algorithm.

Chapter 2 will contain details on the background of OpenCL, the architecture differences between devices used, and further details of document filtering using a Naïve Bayes classifier.

Chapter 3 will discuss implementation details and optimisations of the document filtering software.

Chapter 4 will evaluate the performance of the document filtering software over a range of different devices, with a comparison to a traditional CPU implementation. The chapter will end with a discussion of the results.

Chapter 5 will summarise and discuss the results from the evaluation, and will indicate further work opportunities to increase performance and efficiency further.

Chapter 2

Background

2.1 OpenCL

2.1.1 Language Overview

Open Computing Language (OpenCL) is a framework for writing applications designed to be executed across heterogeneous platforms such as central processing units (CPUs), graphics processing units (GPUs), and accelerator devices such as the Intel Xeon Phi and Field-Programmable Gate Arrays (FPGAs). The language is an open standard maintained by the Khronos Group [1]. OpenCL is based on the C99 language standard [989] and can be loosely thought of as a subset of the C99 language with some additional keywords. One of the major omissions from the OpenCL language is the lack of dynamic memory allocation. The first release of the OpenCL standard (OpenCL 1.0) was on the 8th of December 2008. There have been three subsequent stable releases of the standard, OpenCL 1.1 and 1.2, on the 14th of June 2010 and 15th of November 2011 respectively and OpenCL 2.0 on the 18th of November 2013. The version with most hardware support is OpenCL 1.1 [Gro] and this standard forms the basis of all discussion on OpenCL in this report.

OpenCL splits the program code into two sections called 'host' and 'device'. Host code runs on the CPU and is written in a traditional language such as C or C++. The host code executes in the same way as a traditional application and makes use of APIs exposed by OpenCL to define how, and where, OpenCL device code is executed.

OpenCL applications are called kernels. A kernel is a function that executes on an OpenCL capable device such as any CPU or GPU manufactured within the last few years. Since the focus of OpenCL is on parallelism, a kernel is written from the perspective of a single thread instead of requiring explicit threading code that is required by more familiar models for multi-threading/parallelism such as POSIX threads. An OpenCL work item is similar to a POSIX thread with a notable exception that OpenCL has no traditional stack, any 'function' calls by the OpenCL kernel are actually in-lined by the compiler. OpenCL kernels are also compiled at runtime thus the host code, and the compiler itself, can make device-specific optimisations for optimal performance.

2.1.2 Models

To accommodate the variety of devices OpenCL is designed to work with, a more abstract memory model, and concurrency and execution model was designed to allow the programmer full access to the target devices'

[1] `http://www.khronos.org/opencl`

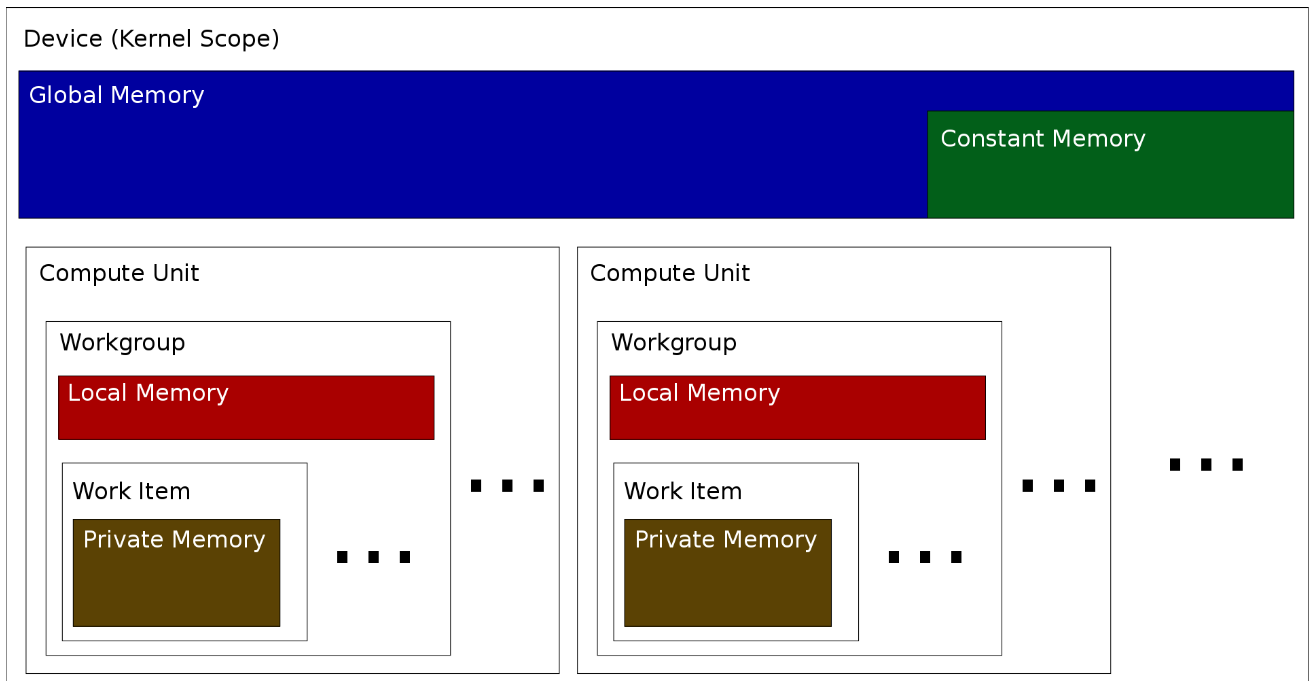

Figure 2.1: Simplified visualisation of OpenCL Memory Model

capabilities. This complicates the developer's job, by requiring them to think at a lower level of abstraction than what they may be used to in traditional C/C++ applications, however this is required to allow the application to make the most out of the target hardware.

Memory Model

For a traditional C/C++ application, the application programmer can make use of the primary memory space of the computer, typically DDR2 or DDR3 RAM. There are caches on the CPU die however no direct control over their contents is offered. For CPUs this is not an issue, all modern CPUs support automatic caching. Problems occur when considering other devices, such as a GPU. GPUs do not typically have automatic caching thus a programmer must make use of OpenCLs abstract memory model, and hardware vendors must map this model to the physical hardware. The OpenCL memory model defines four different types of memory, each of which will be outlined below. A simplified diagrammatic representation of the memory model is shown in Figure 2.1.

Global memory This memory is accessible by all compute units and is similar to primary memory on a traditional CPU system. All work items are able to access and update everything stored in this memory. For a GPU this is memory mapped to the primary memory of the GPU, called GDDR, and is typically 1-4GB in size. When transferring data between the host's primary memory and the OpenCL device, this is where the data has to reside. Global memory is denoted as '__global' in an OpenCL kernel.

Constant memory Constant memory resides in the same space as global memory and is used for data which is accessed simultaneously by all work items, and for variables whose values never change. Constant memory is denoted as '__constant' in an OpenCL kernel.

Local memory This is scratch pad memory which is only visible to a single compute unit on the device. Local memory can be split up into distinct sections if a single compute unit is executing multiple work groups simultaneously. Local memory is generally on-chip memory and thus has faster access time than global

memory. Local memory is typically on the order of tens of kilobytes in size. Use of local memory is either statically declared in the kernel (e.g '__local float[64] localArray') or dynamically allocated at runtime by the host. OpenCL kernels do not offer dynamic memory allocation during kernel runtime.

Variables in local memory are not what would be called 'local variables' in a traditional application, OpenCL local variables are in private memory. Variables in local memory are shared across a single work group, which can contain multiple work items.

Private memory This is memory that is unique to an individual work item. Any local variables and non-pointer kernel arguments are private by default. These are typically mapped to registers although they can be spilled over to higher latency memories, such as local memory, if there is insufficient space available.

Concurrency and Execution Model

As shown in Figure 2.1, OpenCL uses a number of different terms to describe its execution model. There is no explicit 'thread' in the OpenCL model, OpenCL has work items which are the closest relative to traditional threads and, for the most part, the terms can be used interchangeably. The OpenCL Concurrency and Execution Model is split into two parts, host and device, and both will be outlined below.

Host

Host program As discussed previously, a host program is a traditional program (written in C/C++ for instance) and is executed on a CPU. The host program is responsible for managing the execution of kernels by creating and setting up command queues for memory commands, kernel execution commands, and synchronisation of commands.

Context The host program defines a context for the OpenCL devices. The context will include the kernels to be executed, the available OpenCL devices, and the memory objects used by kernels.

NDRange This stands for N-Dimensional Range and defines how work items are organised. Work items can be arranged into one, two, or three dimensions. This can be designed to increase performance and improve clarity of kernel code. For instance, modifying a two dimensional matrix would warrant having a two dimensional range as this would be the most natural way to map work items to matrix elements.

Device Type OpenCL places devices into three different classifications. CPUs, GPUs, and Accelerators. An example accelerator device would be the Intel Xeon Phi [2].

Device

Compute Unit This is a generic term used to describe a streaming multiprocessor (SMX) on a GPU, or a core of a CPU. For some devices (such as a GPU) a compute unit will have dedicated programmer accessible local memory, visible only to itself. A single compute unit can be assigned multiple work groups.

Kernel As discussed previously, OpenCL code which runs on an OpenCL device is known as a kernel. These are written from the perspective of a single work item.

Work group This is a collection of work items. An entire work group is assigned to one compute unit. The work items that are associated with the work group can share local memory and can synchronise with one another at explicit synchronisation points in the kernel.

[2] http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html

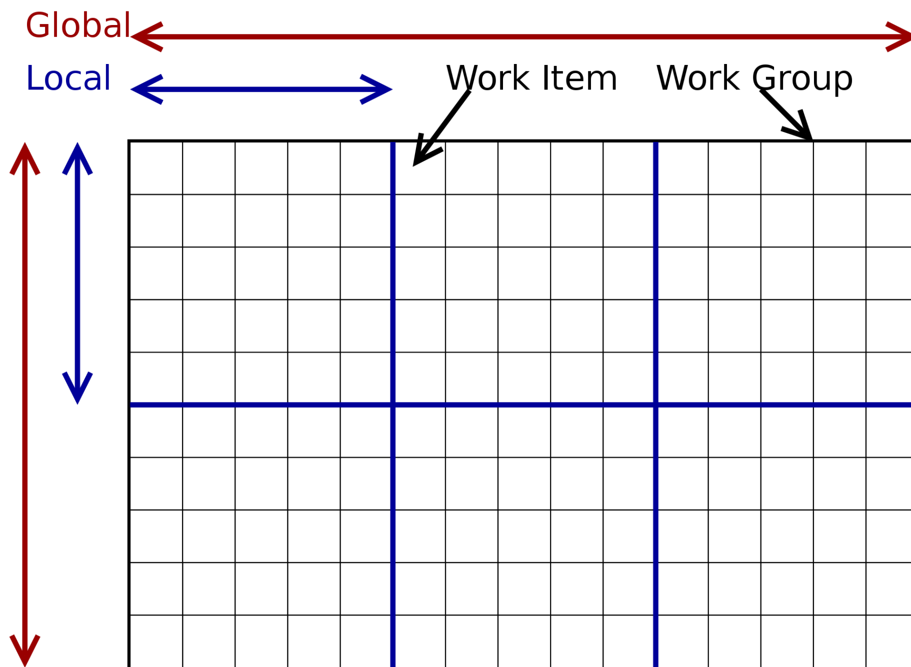

Figure 2.2: Simplified visualisation of OpenCL Execution Model with a 2D range

Work item A work item executes the kernel code. Each work item is assigned a work group. Every work item can access the local memory of its work group and can access all global and constant memory. A work item will also have private variables which may or may not fit inside private memory.

Global range This represents the total number of work items in each dimension. The result of the multiplication of the global size in each dimension is the total number of work items being executed for the corresponding kernel.

Local range This represents the total number of work items in each dimension for a single work group. The global size for a dimension has to be an integer multiple of the local size for that dimension for the work group size to be valid. This is to ensure work groups can fit inside the global dimension size evenly.

Figure 2.2 shows diagrammatically the relationship between global and local range, and work items and work groups. Shown is a two dimensional range, with global size of 15 for dimension 0, and 10 for dimension 1. Local size is 5 in both directions creating 6 work groups with 25 work items each.

Just as with the Memory Model, different devices will realise the Concurrency and Execution model differently. These differences will be discussed in the next section.

2.2 Device comparisons

There are numerous important architecture differences between different classes of device. At the high level, CPUs are aimed at being good all rounders, suitable for a multitude of tasks whereas GPUs have focused on the problem of pixel value generation in a highly parallel fashion. This had led to the evolution of different device architectures that are suitable for different types of task. The important differences will be outlined in this section.

2.2.1 CPU and GPU

The CPU is typically referred informally as the 'brain' of the computer, not only does it run the user's wide variety of software, it also acts as the overall controller for the entire system. This necessitated a one-size-fits-all style of approach when designing the CPU chip and the corresponding instruction set architecture. A CPU typically has a high clock speed, in the range of 3-4GHz for a modern desktop. Each die can contain a small number of cores, typically four, that execute instructions simultaneously but independently.

A GPU is a highly parallel oriented chip, with hundreds of cores running at a slower clock speed (about one fifth of a CPU core) and they have a restricted instruction set primarily designed for simple, repetitive tasks needed for video output. For instance, a kernel on the GPU is unable access the CPUs primary memory space, the host CPU needs to explicitly transfer data to the GPU before kernel execution can begin. With the growth of the use of GPGPUs, the restricted instruction set is becoming less of an issue, and CPUs and GPUs are becoming able to communicate and share directly with plans to introduce unified memory spaces.

Threading and Cache

A CPU has a few hardware threads, typically one per CPU core, with occasional context switching between software threads. The CPU has multiple levels of cache, typically two or three, to reduce memory access latency by exploiting spatial and temporal locality of memory accesses. It does this by automatically copying data in the same 'cache line' (around 64 bytes) of the requested memory from primary memory into the cache. The idea behind this is that a program will likely need data values nearby the memory location just accessed, for instance iterating over the array. This is called spatial locality. Temporal locality is where a data value used now is likely to be used again a short time later on. Exploiting these localities can lead to significant gains in performance of one or two orders of magnitude as main memory can take tens or hundreds of clock cycles to respond to a request.

GPUs have many threads to keep compute units busy, if some threads are waiting on memory accesses other threads are available to run. Latency is thus hidden rather than reduced. To help facilitate this latency hiding, the context switch for GPU threads is designed to have insignificant cost.

For the Intel CPU used in the experiments, each CPU core has its own private L1 cache with a L2 cache which is shared between all four cores. For the nVidia GPU used in the experiments, there are fourteen streaming multi-processors (SMX) which have a private L1 cache. All SMXs share a L2 cache.

Arithmetic Throughput

GPUs use more of the transistor budget on arithmetic units than memory cache compared to CPUs. This makes GPUs more suitable to high density arithmetic and repetitive computations than a CPU.

Execution Model Implementation

CPUs have a very free implementation of the OpenCL execution model. Threads are free to take their own execution path, and aren't directly affected by another thread's progress except from at explicit synchronisation points.

GPUs, on the other hand, have a Single Instruction Multiple Threads (SIMT) execution model. This means that GPUs have one instruction unit for multiple arithmetic units. Threads are partitioned into groups which

execute the same instruction simultaneously. For nVidia GPUs these are referred to in the CUDA [3] model as 'warps' and typically contain 32 threads. This can increase performance as this reduces the number of fetch and decode operations on instructions required. A performance consequence of this is in code with conditional branches.

```
if (someCondition)
{
    nonTrivialCode();
}
else
{
    someMoreNonTrivialCode();
}
```

Figure 2.3: Divergent Warp Example

For a simple example, consider the situation when someCondition is true for half of the threads, and false for the other half of the threads. For a CPU each thread executes the instructions by itself and some threads can be executing 'nonTrivialCode()' at the same time some threads are executing 'someMoreNonTrivalCode()'. However, on a GPU the threads in the true branch execute their instructions while the other threads remain idle, and only after the true branch has been completed, the false branch threads execute. This is known as a divergent warp. Divergent warps can considerably reduce performance as it reduces the number of threads executing in parallel and should be avoided when developing an OpenCL kernel for a GPU as much as possible.

In addition to instruction execution, memory accesses to any of the on-board memories are done on a per warp basis. For ideal performance, addressing should be such that threads in a warp access coalesced memory locations to reduce memory transaction cost. For instance, an array where access is based on a hash function can result in poor performance as threads in a warp are likely to be accessing significantly different parts of memory. If all threads access consecutive memory locations, then one or two reads can be enough to supply data required by tens of threads, rather than one read per thread.

Extra performance consideration

In addition to the architecture differences between CPUs and GPUs, there is a separate consideration that must be made for GPUs and other discrete devices, the PCIe bus. GPUs are unable to directly access main memory like a CPU, thus before any code is run on a GPU, data must be transferred between primary memory and the GPUs on-board memory. This can take a significant period of time in comparison to the code runtime. This is due to the relatively slow speed of the PCIe bus in comparison to on-board memory access. A GPU is able to access its global memory at a rate of around 256GB/s on modern chips. Contrast this to the PCIe link speed which can be as low as 250MB/s per lane. The most common systems have PCIe 2.0 slots which run at 500MB/s per PCIe lane so, with 16 lanes available for the GPU, this gives a maximum theoretical transfer speed of 8GB/s, 32 times slower than access from on-board memory.

2.2.2 Bridging the gap - APUs

Both CPUs and GPUs are strong in their respective application areas. Ideally, a system should be able to make best use of both types of device with as little overhead as possible. One of the obvious potential bottlenecks is

[3] https://developer.nvidia.com/about-cuda

the PCIe bus to devices discrete from the CPU. This led to the design of an Accelerated Processing Unit (APU). These have a traditional CPU in addition to a GPU (or FPGA) on the same die and are a good example of a Heterogeneous System Architecture (HSA). This design isn't new, CPUs have typically contained a small GPU, especially in the small laptop market. With the growth of Heterogeneous Systems, there has been a trend to increasing the GPU performance on die with the CPU. This reduces the gap between the CPU and the GPU and removes the PCIe bus overhead. The primary consumer implementation of this is AMDs HSA [4]. The joint die allows power to be managed so that different applications can make most effective use of the CPU and GPU cycles available. In addition, there is also work going on to completely unify address spaces, to make the heterogeneous architecture more seamless to work with.

For the purposes of this report, APU discussion ends here. At present consumer devices are relatively immature and for a majority of applications, a discrete GPU or accelerator card is more likely to achieve the best overall performance.

2.2.3 Intel Xeon Phi

The Intel Xeon Phi is similar in many respects to a CPU, with around 60 x86-compatible CPU cores on a discrete card. The device additionally has 8GB of on-board GDDR5 memory, similar to a GPU. The device sits in a PCIe slot like a GPU and thus has the same performance considerations for the PCIe bus as a GPU. A detailed look at the inner workings of the Intel Xeon Phi architecture can be found on Intel's website [5].

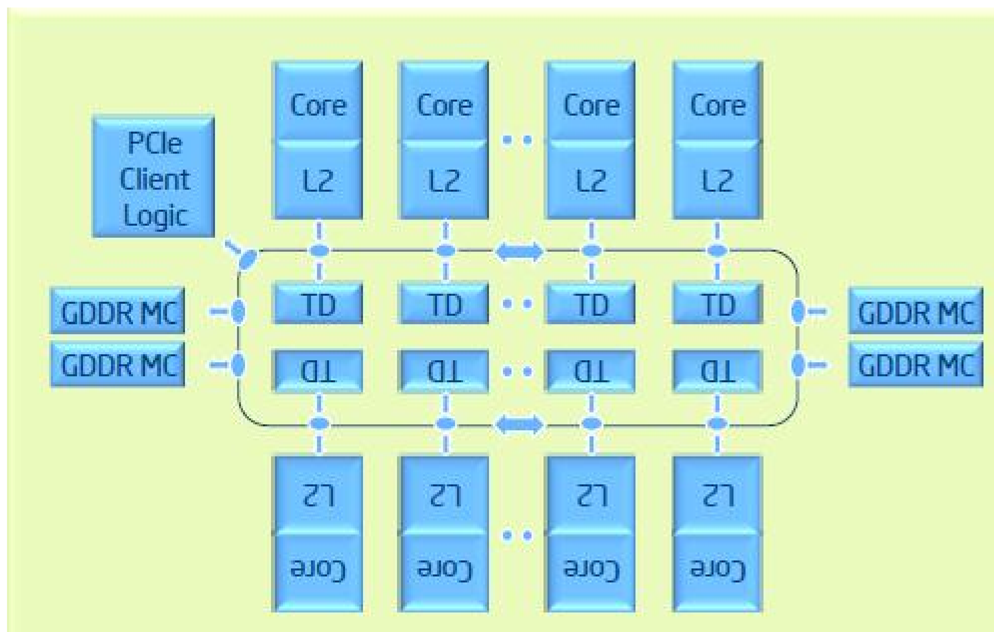

Figure 2.4: Intel Xeon Phi Microarchitecture - Copyright Intel Corporation ©

Each CPU core is on a bi-directional ring bus to memory and each core is capable of 4-way hyper threading resulting in 240 hardware threads per device. Clock speeds are typically lower than a CPU at around 1GHz. The Intel Xeon Phi is designed to allow CPU-like programming versatility with the high performance and efficient power use of an accelerator chip.

A figure showing a brief overview of the microarchitecture is shown in Figure 2.4. This is taken from the

[4] http://developer.amd.com/resources/heterogeneous-computing/

[5] http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner

Intel website linked above. It shows a subset of the x86 cores and their corresponding private L2 cache, all connected to the rest of the device by the bi-directional ring bus.

In addition to being an OpenCL capable device, the Xeon Phi is capable of running applications written in traditional source code such as C, C++, or FORTRAN. The device itself contains a Linux operating system which is automatically started up and loads a file system, allows secure shell logins, and enables a TCP/IP stack to facilitate communication as if the device was a separate internal network device.

Performance Considerations

In addition to the PCIe bus consideration, the ring bus to memory needs to be taken into consideration. This is most notable with communication between cores. Communication between neighbouring cores will be fast, however communication between cores on the 'opposite side' of the ring will be significantly slower.

2.3 Document Classification

As discussed in Section 1.4 on related research, the work in this report directly relates to and builds on work by Wim Vanderbauwhede et al. The purpose of their work is to classify a stream of text documents based on significant terms in the documents. A classic example would be emails, where spam emails tend to have specific terms to indicate their class. An assumption is that most of the documents will not be of interest (mostly not spam in our example) so the profiles of significant terms discriminates highly. The classification is achieved by a Naïve Bayes classifier with n-grams, similar to the work by the University of Waterloo [PS03] for text classification and Greek [MAP06] and German [Sch03] academics for spam filtering itself.

The most recent papers by Wim Vanderbauwhede et al. discuss the use of FPGAs for high throughput document filtering. The inherent parallelism in document classification (as is also true with many data-centric algorithms) and the parallelism focus of FPGAs, was found to be a suitable pairing to create a power efficient, high throughput scoring system for parsed documents [VFA+13] [Van13]. The results of the latest paper show that, while an FPGA can score at a median performance of 749MB/s [VFA+13], the CPU can only parse at 426.2MB/s making the parsing on the CPU the bottleneck to higher performance. This report will take the same initial steps, implementing the scoring on a GPU, and then, due to the parser being the bottleneck to performance, look into having parsing and scoring on the GPU and other devices. This is similar to the work by Altera Toronto Technology Center which focuses on performance per watt ratios [CS12] instead of only looking at raw performance.

2.3.1 Parser

Structure

The parser is a Finite State Machine originally implemented in C++ but designed to also be suitable for FPGA implementation. The parser is able to make full use of all hardware threads available to it from the CPUs [Van13]. The output from the parser is a series of terms (English language words) in a sixty four bit number. Each term can contain up to twelve characters, with each character being represented by five bits. The last four bits of the term are used to identify the number of characters in the term. If a term is larger than twelve characters, the remainder of the word is truncated. Given the average English word length of 8.23 characters [Par] this is an acceptable encoding.

Baudot Encoding

The encoding used to convert an eight bit ASCII character into five bits is a variant of Baudot encoding [Ste]. Most text compression algorithms are lossless, the original text can always be returned during a decompression stage. Baudot code is different, it is lossy. The compressed text can still be read as English, just with some character substitutions. The algorithm is outlined in Figure 2.5:

```
to5BitEncoding(char c)
{
    if (c == 'i' || c == 'j' || c == 'I' || c == 'J')
    {
        return 1;
    }
    if (c == 'o' || c == 'O')
    {
        return 0;
    }
    if (isLowercase(c))
    {
        tcode = c - ((v > 106) ? ((v > 111) ? ((v > 117) ? 4 : 3) : 2) : 0);
        return tcode - 87;
    }
    if (isUpperCase(c))
    {
        tcode = c - ((v > 74) ? ((v > 79) ? ((v > 86) ? 4 : 3) : 2) : 0);
        return tcode - 55;
    }
    if (isNumeric(c))
    {
        return c - 48;
    }
}
```

Figure 2.5: ASCII to 5 bit encoding pseudo code

In short, characters which look like a 1 (‘i’, ‘j’, ‘I’, and ‘J’) are converted to 1. Characters which look like a 0 (‘o’, ‘O’) are converted to 0. ‘u’, ‘v’, ‘U’, and ‘V’ are treated to be ‘U’. All other characters have their ASCII value reduced so that they fit into numbers lower than 32. All lower case characters are now upper case meaning now, in total, there are 10 (numeric) + 26 (letters) - 4 (‘I’, ‘J’, ‘O’, and ‘V’) = 32 characters and can be represented by five bits.

2.3.2 Scorer

The array of terms from the parser is passed onto the scorer, prior this report this could be another CPU routine, or an FPGA. The beginning of documents is marked by a term representing the document ID. The last four bits, the term length field, are left at 0 to indicate that this is a document start.

Terms and N-Grams

Each term is taken in turn. N-grams (up to tri-grams) are created with the new term. An n-gram is essentially a concatenation of the newest term onto the end of the last n-1 terms, up to an n-gram length of twelve as with the

original terms. The reason behind this is that an English word on its own right isn't always significant but with the previous words to place it in context, it may be. A simple example from the area of spam filtering is the word 'Rolex'. On its own it's not necessarily important whereas with the tri-gram 'Cheap Real Rolex', it becomes more significant and indicates that the document in question is likely spam.

Profile

After the n-grams have been generated, each one is taken in turn and looked up in the profile of terms. The profile is a hashed data structure of (term, weight) pairs. The weight represents how common the term is in one class, compared to the other. If the term exists in the profile, the weight of the term is added to the document's score. It is this score which is ultimately used to determine a document's classification. If the score is above a certain user defined threshold value, the document is significant to us and classified as *spam*, otherwise it is classed as *not spam* in our example.

A profile is typically quite large, 128MB in this case. As such, the profile is organised in such a way that it is a hashed array allowing for O(1) (constant time) lookup rather than a linear, or binary search which would be O(n) and O(log n) time respectively (where n is the number of elements in the profile). To work out where a term is in the profile, a simple bit shift and masking operation is executed to obtain an offset into the profile array. This allows for the possibility of collisions with other terms thus probing is used (up to four addresses from location calculated by hash function) to reduce the likelihood of missing a term in the profile.

Bloom Filters

With the profile being large, it will always be mostly resident in main memory. For the CPU system this is off chip, and for GPUs and accelerator devices this is on the same board, but both have significantly higher latency than caches which reside closer to the CPU cores and GPU compute units. With the profile only containing a subset of possible terms, and the requirement to look up four terms due to hashing collisions, the latency to go to main memory can end up being quite significant. To reduce the need to go to main memory, a small data structure (4KB or so) is employed called a bloom filter. Burton Bloom first defined this data structure in a famous paper: "Space/Time Trade-offs in Hash Coding with Allowable Errors" [Blo70].

A bloom filter is used to store the possibility that a term exists in the profile. With the size of the bloom filter being small, it's likely it will migrate into a cache of the CPU and can be copied into the local memory of the GPU allowing for efficient lookup of the bloom filter and minimised trips to memory for non-existent profile terms.

A bloom filter works as follows. It is an array of bit values where 0 indicates absent and 1 indicates present. When constructing the bloom filter, each entry in the profile is run through a number of hash functions (for example, four) to get a number of numerical values. These numerical values are indices for bit positions in the bloom filter. These bit positions are set to 1. This can be summarised in Equation 2.1.

$$bf[i] = (i = h_j(term_k))?1 : 0, \forall i \in 0..S_{BF} - 1], j \in 1..N_{hfs}, k \in 1..N_{terms} \quad (2.1)$$

where S_BF is the bloom filter size in bits, N_{hfs} is the number of hash functions, N_{terms} is the number of terms in the profile, $term_k$ is sixty bits and h_j is a hash function that hashes sixty bits into ($log_2 S_{BF}$) bits.

To use the bloom filter, the term to be looked up is run through the same hash functions. If all locations returned by the hash functions have their bit set to 1, then it's likely the term is in the profile. Even if n-1 out of n positions are set to 1, with the nth set to 0, the profile is known not to contain the term. However, it is possible

that a term can have all n bit positions set to 1 but still be absent from the profile. Think of the case where term 1 sets positions a, b, c, and d, and term 2 sets positions c, d, e, and f. An absent term 3 could hash to positions b, c, d, and e which, although they are all set to 1, the term isn't contained in the profile. In other words, a bloom filter cannot have a false negative result (all terms in profile can be looked up) but can have false positive results like term 3.

This is the space-time trade-off that Bloom discusses. A bloom filter can be very small compared to the data structure it's supporting but lookups can take more time due to the false positive results. If the false positive rate is low enough, the time saved from the reads from memory avoided by true negatives outweighs the time wasted looking up false positive results. To investigate the effect of false positives, when experiments are conducted, three bloom filters were used. One bloom filter is generated from the profile, another has all zero bits to represent a situation with no false positives, and the third has all one bits to represent a situation with as many false positives as possible. The last two give a range of possible throughput values possible by using a bloom filter with the all zero bits bloom filter being the best case, and the all one bits bloom filter being the worst case.

Chapter 3

System Implementation

3.1 Base Implementation

The project implementation is split into two primary sections. The first is the scoring only implementation, the second is the parsing and scoring implementation. The scoring only implementation made up the entire initial project plan and involved porting the existing scoring code from C++ to OpenCL. After the implementation of the scoring code, and the initial analysis of results was conducted, it was determined that the parsing code (also written in C++) was the bottleneck to the system and the project then began to investigate also porting the parsing code to OpenCL.

This section will detail the implementation of both systems before going on to discuss optimisations in the next section.

A note on terminology when discussing blocks of contiguous memory. When a contiguous block of memory is allocated on the host CPU it is normally referred to as an array. When the array is transferred to an OpenCL device it is referred to as a buffer. These are functionally the same with the primary distinction being where they are allocated.

```
// One time set up
platforms = getPlatforms()
devices = getDevices(platforms[0])
context = createContext(devices)
queue = createCommandQueue(context, devices[0])
program = buildProgram(context, sourceCode)
kernel = createKernel(program, kernelMethodName)

// repeat for each buffer
buffer1 = createBuffer(context, options, size)
setArg(kernel, 1, buffer1)
writeBuffer(buffer1)

resultsBuffer = createBuffer(context, options, size)
setArg(kernel, n, resultsBuffer)

// Execute kernel and get results back
enqueueKernel(queue, kernel)
readBuffer(resultsBuffer)
```

Figure 3.1: OpenCL Host Pseudo-code

3.1.1 Both Implementations

Most OpenCL applications are implemented in much the same way from the host code's perspective. There is a significant amount of boiler plate code to detect OpenCL platforms and devices available on the target system. Each device that is going to be used needs its own command queue. An OpenCL command queue is as the name suggests, any commands for data transfers or kernel executions are placed on the device's command queue and executed in order of arrival to the queue (although out of order execution is possible). Command queues, memory, program, and kernel objects are all managed by an OpenCL context. The kernel itself also has to be compiled, it is read in from disk as a string and built for the required devices. This is what allows a single kernel to run on any available device selected, it's compiled at run time. After compilation has completed, the kernel's arguments are set to the appropriate device buffers. The host's arrays that contain data the kernel needs are then transferred to the device's buffers and the kernel is executed. After the kernel has finished executing, any buffers relating to results, in this case the document scores, are written to the host CPU program back by en-queuing a read buffer command.

As discussed in the background, the Naïve-Bayes classifier makes use of a profile. This is a 128MB file containing the terms and associated term weights. This is read in from disk to main memory by the C++ host device code, and transferred to the OpenCL device as a buffer of unsigned long values to the OpenCL kernel. It does not change between invocations of the kernel, as all sets of documents are scored using the same profile.

The use of a bloom filter was also discussed in the background. This is a 4KB array of bit values used by the OpenCL kernel to see if there is a chance that the document term being scored exists in the profile. As with the profile, this is read in from disk and passed in as a buffer to the OpenCL kernel. It also does not change between invocations of the kernel because the profile does not change.

OpenCL host code is summarised in the pseudo-code in Figure 3.1. When creating buffers, the options that can be passed in include whether or not the buffer is read-only, write-only, or both.

```
documentNumber = get_global_id(0) + 1 // work item ID
startIndex = docAddresses[documentNumber]
endIndex = docAddresses[documentNumber + 1]
score = 0
for (term in terms[startIndex..endIndex]) {
    generateNGrams(term)
    for (ngram in ngrams) {
        address = getTermAddress(ngram)
        profileEntryVector = profile[address]
        for (i = 0; i < 4; i++) {
            profileEntry = profileEntryVector[i]
            if (profileEntry contains ngram) {
                score += getWeight(profileEntry)
            }
        }
    }
}
scores[documentNumber -1] = score
```

Figure 3.2: Scoring Pseudo-code

3.1.2 Scoring Only Implementation

In addition to the profile and bloom filter, the scoring only implementation reads in the parsed document terms from disk. These are passed to the OpenCL kernel and are used to calculate a document's score.

The array of terms is a one dimensional array thus the C++ code has to iterate over the array and find the beginnings of documents to know how many work items to create (one per document) and over what range of terms represent each document. This simply involves the creation of an index containing the location of terms with length zero as this was used to delimit document beginnings in the original parser. The array of document marker addresses is prefixed by the number of documents and has the total number of terms added to the end of the array. The total term number is required by the work item scoring the last document as there is no next document to determine where to stop scoring terms at. This buffer is also transferred to the device.

All buffers are read only with the exception of the scores buffer. This is a write-only buffer with length equal to the number of documents being scored. It is this buffer that the kernel writes the results to and is the only buffer that needs to be written back to the host memory after kernel execution.

After the kernel has the arguments set to the appropriate buffer, and the data has been transferred to the device, the kernel can execute. During kernel execution, each work item iterates over the terms for their associated document. A work item knows which document to score because the location of the document starting term is the work item's ID + 1, up to the term at the work item's ID + 2 into the document index buffer. Each term in this range is scored and the total score for the document is stored in a scores buffer at location equal to the work item's ID.

The OpenCL kernel code is summarised in the pseudo-code in Figure 3.2. For the actual OpenCL kernel, the method calls for getTermAddress() and getWeight() are written as a single line containing the required bitwise operators such as shift and exclusive or.

After the kernel has finished executing, and the scores buffer has been written to a host device array, the actual classifications of each document is recovered. A document is classed as significant (spam in the spam or not spam example) if the score is above a certain user defined threshold.

Level of Parallelism

With the kernel written so that a single work item scores a single document, this is fairly coarse-grained parallelism. This was chosen as it was believed to be most suitable for the target architecture, a GPU, even though that documents of differing lengths can cause performance issues as work items will end up waiting on other work items in the same warp to finish. However, an alternative was looked into. Instead of a work item scoring one document, which could result in some work items scoring documents of vastly different sizes, they could score a section of the total number of terms thus ensuring each work item would have the same amount of work to do as the rest. The problem with this is the extra work required at the beginning of the kernel, and when adding part of a document's score to the scores buffer.

At the beginning of the kernel, a work item would have to work out which document it was scoring which would involve either a linear or binary search over a buffer of document ID locations until it found the highest document ID location that was less than the work item's starting location. Also, before beginning scoring, the previous n terms would have to be read by the work item to populate the n-grams array so not to miss one involving its first term.

During scoring, a work item would be continually looking to see if it had reached the end of a document and, if so, executing a globally atomic add on the scores buffer to ensure its results aren't overwritten by another work item. Atomic operations are expensive and their placement would result in a conditional branch added to the main scoring loop, both of which can cost heavily performance-wise, especially in a GPU where the SIMT architecture discussed can result in divergent warps on GPUs.

In addition to the complications of having term-level parallelism, as will be shown in the Results section, the GPU scored significantly faster than the CPU parser, thus the CPU parser was the bottleneck meaning any potential improvements to the GPU scorer wouldn't result in overall increases in system throughput.

```
documentNumber = get_global_id(0) + 1 // work item ID
startIndex = docAddresses[documentNumber]
endIndex = docAddresses[documentNumber + 1]
state = SKIPPING
score = 0
for (char in documents[startIndex..endIndex]) {
    if (char is alphanumeric) {
        // If inside a tag, keep skipping characters
        updateState(state, 0)
        if (state is WRITING) {
            term += to5BitEncoding(char)
        }
    } else if (char is whitespace) {
        // If valid term, time to score this term
        updateState(state, 2)
        if (length(term) > 0 and state is FLUSHING) {
            score += scoreTerm(term)
            term = 0 // Parse a new term now
        }
    } else if (char is '<') {
        // Start skipping characters
        updateState(state, 3)
    } else if (char is '>') {
        // Stop skipping characters
        updateState(state, 4)
    } else {
        // Keep skipping characters
        updateState(state, 1)
    }
}
```

Figure 3.3: Parsing and Scoring Pseudo-code

3.1.3 Parsing and Scoring Implementation

For the parsing and scoring version, the host code is very similar. The only differences relate to the documents themselves. For scoring only, the documents were already parsed and represented by terms. For parsing and scoring, the plain text is instead read in. The TREC data collection is read in as a single text file and passed to the OpenCL kernel as-is. For the document marker array, instead of detecting zero length terms, the C++ code searches through the plain text and stores the character locations where the sub-string *<DOC>* begins. This sub-string acts as a direct replacement to the zero length term locations, and is transferred to the OpenCL kernel in the same way.

Again, each work item is responsible for a single document. During the kernel invocation, each work item parses their respective document character by character and, as soon as a full term has been parsed, scores the term. A term is ready to score when a white space character has been reached and the term is of non-zero length. Any characters inside tags, for instance the start and end *DOC* tags, are ignored. After all characters have been parsed, the scores are stored in the scores buffer, the same way as in the scoring code.

After the kernel has finished, the scores are written back to the host and the classifications are calculated as before.

The parser is a state-machine parser optimised for speed, as designed for the CPU-FPGA Hybrid Implementation [Van13] by Wim Vanderbauwhede et al. The OpenCL kernel code is summarised in the pseudo-code in Fig-

ure 3.3 where to5BitEncoding(char) is the Baudot encoding discussed and shown in Figure 2.5, scoreTerm(term) contains the profile lookup and scoring addition code from Figure 3.2, and updateState(state, number) works out the parser state machine transition based on the current state, and what character has just been found. During implementation, an array-based and a case statement based method for updating state was investigated, neither had a performance advantage so the case statement method was used. States tell the parser to either skip characters, write characters to the term, 'flush' the term (aka score the term), or that a tag has been encountered.

Level of Parallelism

With the kernel written so that a single work item parses and scores a single document, this has the same coarse-grained parallelism as the scoring-only kernel. The reasons for this choice are the same, especially given that the parsing and scoring kernel contains the same code for the scoring method.

If a more fine grained method is considered, where each work item scores part of a document such that all work items parse the same number of characters then, in addition to the scoring concerns, the parsing stage has its own concerns. A work item would not only need to work out which document it is parsing but also work out where in the document it is starting from. The work item would initially need to work backwards to see whether it is inside a tag or not and, if the work item is outside a tag, start parsing from n words back from its starting location to generate the correct n-grams. Even if this situation was viable, the five branches in the parsing code still pose a performance issue as each work item will still be parsing and scoring different sizes of words thus the work items in a single warp will still diverge.

3.2 Optimisations

OpenCL kernels can run on any device with a suitable compiler. This portability is helpful when using different types of devices which may come from a multitude of vendors. This portability has a cost however, there is no guarantee that the kernel will run well. For instance, CPUs and GPUs are significantly different in terms of how they cache and GPUs are more favourable towards smaller tasks which can be executed in parallel. This requires a developer to consider the architectural differences of different devices and tune each kernel for each device. Two significant optimisations that were employed during this project are discussed below. The "Local Size" optimisation discussed in Section 3.2.1 involves changing the number of work items in a work group to suit the target device. The second optimisation, "Mapped Buffers" discussed in Section 3.2.2, is a simple optimisation to remove any unnecessary data transfers.

A third optimisation using bloom filters was also employed however this is discussed in the next section as the three bloom filters, and their effect on performance, demands a more thorough analysis.

3.2.1 Local Size

When executing a kernel, the host code specifies the number of work items to create (the global size) and the number of work items per work group (the local size). With a single work group being assigned to a single compute unit, and one compute unit being able to have multiple work groups simultaneously, the local size can have a significant effect on kernel performance. Different devices react better to larger work groups, for instance GPUs which can use large work groups to improve latency hiding. The type of algorithm also has an affect on the optimal local size, certain characteristics prefer larger work groups to others. For these two reasons, there is no hard and fast rule dictating what local size to use for what device and algorithm, experimentation to fine tune the kernel is the only way.

Local size can have a value from 1 up to 512, 1024, or 2048, depending on the device. This is specified by the *CL_DEVICE_MAX_WORK_GROUP_SIZE* constant that can be queried at runtime by the host for each device. The value is typically a power of two. The only other restriction to the local size is that it must be a factor of the global size (for instance a local size of 64 and a global size of 4096). Given that global size can be any number, for instance the number of documents, it is common to round up the global size to the nearest number that makes the desired local size a factor. The kernel can then conduct a bounds check on work item global IDs and have the few work items extra return straight away. These extra work items do not affect performance as they only appear as part of the last work group.

Another benefit of modifying the local size is occupancy of a compute unit. A work item uses a certain amount of memory, its private variables. A compute unit only has a limited capacity for these variables thus, based on the kernel code, can only simultaneously handle a maximum number of work items. Say the compute unit can handle 2048 work items at one time, thus any combination of local size (assuming powers of two) will allow for 100% occupancy of the compute unit (say two work groups with size 1024 work items). If the unfortunate situation occurs where 2040 work items can be handled, a local size of 1024 will result in a single work group being handled by the compute unit resulting in just over 50% occupancy. If the local size was 32, there would be 63 work groups assigned to the compute unit, resulting in just under 99% occupancy. A higher occupancy can make better use of a compute unit's resources and increase overall throughput.

Specific to this project, the Tesla C2075 and Intel Xeon Phi 5110P performance for the parsing and scoring kernel had an in-depth analysis relating to their local size.

Tesla C2075

The experiment found that the Tesla C2075 reacted well to a large local size, likely assisting with latency hiding of memory accesses and keeping compute units as busy as possible, and achieved the best performance with a local size of 128.

The graph of the local size effect on the Tesla C2075 throughput for the parsing and scoring kernel is shown in Figure 3.4. There is a gradual improvement in performance up to a local size of 128 where performance then starts to drop off for larger work groups due to a compute unit's occupancy going down. In this example, choosing a poor work group size, for example 1 work item, will increase the time to score a set of documents by a factor of five.

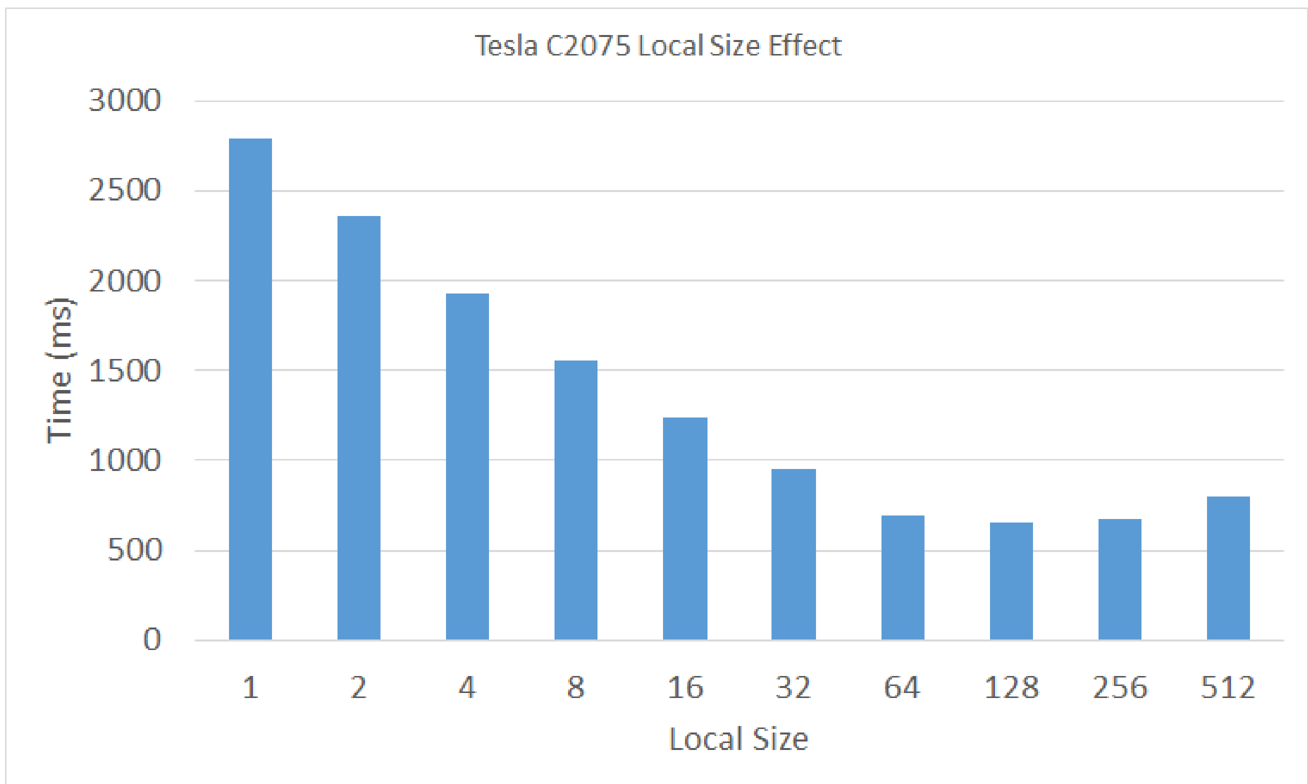

Figure 3.4: Tesla C2075 Local Size Effect

Intel Xeon Phi

The Intel Xeon Phi 5110P reacted best to small local sizes, with a big jump in performance when going from a local size of 16 to 8. The optimum performance was with a local size of 4, which is likely due to the 4-way hyper-threading employed by the device. Normally, the vectorisation capabilities of the device would be expected to play a part in performance, with the local size having a less drastic effect on performance. The Intel Xeon Phi's vector processing unit (VPU) officially supports floating point and integer numbers [1] so either the VPU has an insignificant effect on this application, or the compiler isn't mature enough yet to make full use of its capabilities. This project was not concentrating on the Intel Xeon Phi so precise reasons why this behaviour is being observed was not investigated.

The graph of local size effect on Intel Xeon Phi throughput for the parsing and scoring kernel is shown in Figure 3.5. There is a gradual improvement in performance as the local size is reduced from 512 to 16. Interestingly, there is a massive improvement when moving to a local size of 8, the performance increases by over a factor of two. In this example, choosing a poor work group size (512) will increase the time to score a set of documents by a factor of three.

Regarding overall performance changes, the Tesla C2075 has a greater benefit of an optimum local size to the Intel Xeon Phi however the Intel Xeon Phi has a much sharper reaction when nearing the optimal value whereas the Tesla C2075 has a more gradual change to its throughput.

[1] http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-vector-microarchitecture

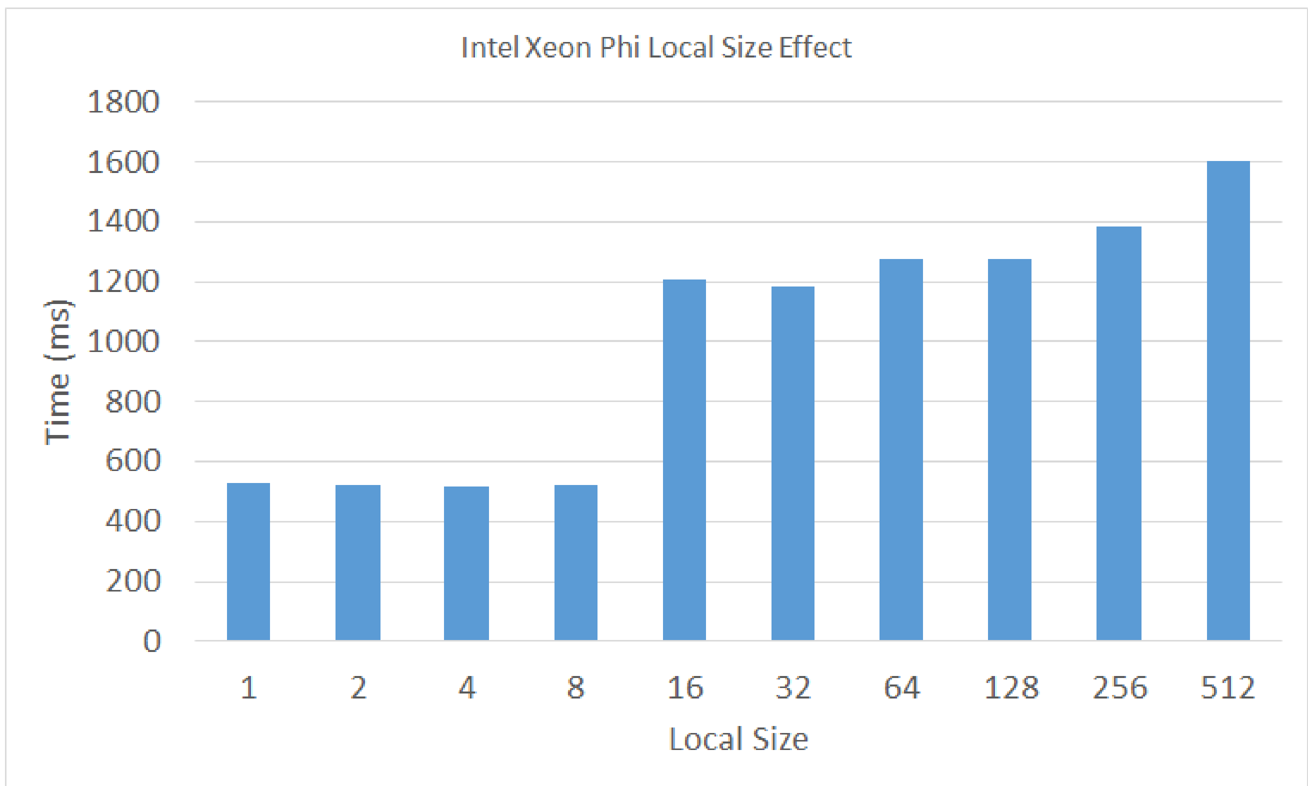

Figure 3.5: Intel Xeon Phi 5110P Local Size Effect

3.2.2 Mapped Buffers

When using an accelerator device such as a graphics card, the data used by the kernel has to be transferred over the PCIe bus by en-queueing a write buffer command. This incurs an unavoidable time cost as the PCIe bus has limited bandwidth. When running OpenCL code on the CPU this is not the case as the OpenCL buffers can stay in primary memory. Unfortunately, the write buffer command does not realise that this is the case and, when executing a write buffer command for the CPU, copies the host array from primary memory to another location in primary memory. As well as duplicating memory usage this incurs an avoidable time cost.

The solution to the issue is to map the OpenCL buffer to the host array. This is done by executing an en-queue map buffer command instead of a write buffer command. The OpenCL buffer can then read from the host address space and removes the requirement for the unnecessary duplication of memory. The benefits of mapping a buffer come with very little overhead as the command takes an insignificant amount of time to execute. This speeds up the CPU implementation by up to ten milliseconds. This in practice only increases throughput by up to 5% in the best case for this application however, given that it's a simple optimisation, removing a small overhead is still worthwhile.

Chapter 4

Experiments

4.1 Devices Used

For the purposes of evaluating the performance of both the scorer and the scorer and parser, a Mac Pro (Early 2008 - MacPro3,1) was used. This was fitted with two Intel Xeon E5462 processors, and a nVidia Tesla C2075 GPU. In addition, there was a separate machine fitted with two Intel Xeon Phi 5110P cards. An AMD system with a traditional CPU setup with 4x AMD Operon 6366 HE CPUs was also used. A brief overview of the hardware will be given to provide some perspective to the results which follow.

4.1.1 Mac Pro System

The Mac Pro runs Mac OS X version 10.8.5 (Build Version 12F45). Both the nVidia and Intel OpenCL SDKs were installed and up to date at time of writing.

CPU The motherboard has two CPU sockets, both with an Intel Xeon E5462 [1] processor. This is a quad core processor with a 2.8GHz clock speed and 12MB of Level 2 cache per processor.

Primary Memory There is 14GB of PC2-6400 (800MHz) DDR2 RAM installed, with error correcting code (ECC) enabled.

GPU The GPU is an nVidia Tesla C2075 [2]. The device has 6GB of dedicated GDDR5 memory running at 1.5GHz giving an internal memory bandwidth of 144GB/s. There are 448 “CUDA Cores” with a clock frequency of 1.15GHz. It has been installed in a PCIe 1.0 slot with 16x lane bandwidth giving a total theoretical bandwidth between its on-board memory and main memory of 4GB/s (250MB/s per lane).

4.1.2 Intel Xeon Phi System

The host system is running the SUSE Enterprise Server 11 Linux distribution. This system was used for running the software system on the Intel Xeon Phi. There are two Intel Xeon Phis fitted and these were used to investigate the performance of a single and a dual Intel Xeon Phi set up. The system does not have a graphics card fitted.

[1] http://ark.intel.com/products/33084/Intel-Xeon-Processor-E5462-12M-Cache-2_80-GHz-1600-MHz-FSB

[2] http://www.nvidia.co.uk/docs/IO/43395/NV-DS-Tesla-C2075.pdf

CPU There is a single Intel Xeon E5-2620 [3] fitted. This is a six core processor with hyper-threading to give twelve hardware threads. The standard clock frequency is 2GHz with a maximum turbo frequency of 2.5GHz. There is a total of 15MB "Intel Smart Cache" available which acts as Level 3 cache.

Primary Memory 16GB DDR3 RAM

Intel Xeon Phi The Intel Xeon Phi is the 5110P [4] variant. There are sixty physical CPU cores running at 1.053GHz, and the cores are 4-way hyper-threaded resulting in 240 hardware threads. There is 8GB of on board memory running at 5GT/s for a theoretical maximum memory bandwidth of 320GB/s. Each device has been installed in a PCIe 2.0 slot with 8x lane bandwidth giving a total theoretical bandwidth between its on-board memory and main memory of 4GB/s (500MB/s per lane).

4.1.3 AMD System

The system is running the Fedora 18 Linux distribution. This system was used solely for running the software system on the AMD CPUs and does not have a graphics card fitted.

CPU There are four AMD Opteron 6366 HE [5] (high efficiency) processors fitted. Each processor has 16 cores running at a base clock speed of 1.8GHz with a maximum turbo frequency of 3.1GHz. When eight or more cores are active, turbo frequency goes down to 2.3GHz. There is 1MB of Level 2 and 16MB of Level 3 cache available per processor.

Primary Memory 512GB DDR3 RAM

4.2 Experiment Set up and Input Files

4.2.1 Data Transfer

The first experiment conducted was to investigate the effect of the PCI Express bus transfer rate on overall system performance. The classification system involves the transfer of very small (4KB) OpenCL buffers, up to very large (100s of MB and above) OpenCL buffers. Both present potential performance concerns. Small transfers can result in low throughput as the waking up of the PCI Express bus and setting up of data transfer can overshadow the actual transfer time. Large transfers, even though being they may be being transferred at the maximum possible transfer rate, will, due to their size, take up significant amount of time to transfer. If the transfer time is significant in regards to the actual time to score the transferred documents, use of an accelerator device could be inefficient, irrespective of the processing power of the accelerator.

There are currently four PCI Express bus standards [6] in place, the first two are in use in the systems discussed above. Each standard doubles (or nearly doubles) the theoretical maximum bandwidth of a single PCIe lane in each direction. Table 4.1 shows this alongside the sampling rate in gigatransfers per second (GT/s) and encoding of data.

[3] http://ark.intel.com/products/64594/intel-xeon-processor-e5-2620-15m-cache-2_00-ghz-7_20-gts-intel-qpi

[4] http://ark.intel.com/products/71992/intel-xeon-phi-coprocessor-5110p-8gb-1_053-ghz-60-core

[5] http://products.amd.com/en-us/OpteronCPUDetail.aspx?id=813&f1=&f2=&f3=Yes&f4=&f5=&f6=&f7=&f8=&f9=&f10=&f11=&

[6] http://www.pcisig.com

PCIe Standard	Sampling rate per lane	Data encoding	Single lane speed	Sixteen lane speed
PCIe 1.0	2.5 GT/s	8b/10b	250MB/s	4GB/s
PCIe 2.0	5 GT/s	8b/10b	500MB/s	8GB/s
PCIe 3.0	8GT/s	128b/130b	985MB/s	15.75GB/s
PCIe 4.0	16GT/s	128b/130b	1969MB/s	31.51GB/s

Table 4.1: PCIe Bus Speeds (from PCISig published specifications)

As Table 4.1 shows, theoretical bandwidth nearly doubles on each iteration. The change from 8b/10b encoding, with 20% transfer overhead, to 128b/130b encoding, with around 1.56% transfer overhead, allowed an increase in of bandwidth of nearly 100% even though the sampling rate only increased by 60%.

The Mac Pro system connects the Tesla C2075 to a sixteen lane PCIe 1.0 slot and the Intel Phi system connects the Intel Xeon Phi to an eight lane PCIe 2.0 slot which theoretically has the same bandwidth as the sixteen lane PCIe 1.0 slot (up to 4GB/s).

nVidia's CUDA SDK supplies a bandwidth test which was used to determined the bandwidth of the Mac Pro system. The tool, called "bandwidthTest" has a mode which is referred to as "shmoo" that runs a series of tests of increasing size, from 1KB up to 64MB to give the full range of possible values.

4.2.2 Document Classification

For each device, the system (scoring only or scoring and parsing) was run over a number of different profiles. There are twelve profiles in total, each 128MB in size. The twelve profiles each had a different combination of terms relating to the two classes a document could come under. Given that the profile is the same size, the profiles themselves do not affect performance, only the classification of documents. Each profile run would be repeated several times.

For the scoring only system, a pre-parsed collection of terms for the TREC collection is given. The parser and scoring only system receives the original TREC collection text file. The TREC collection is a 236MB text file containing 77876 documents in plain text. Each document is enclosed by DOC tags and contains a document number, a date and time, a category, headline, and a series of paragraphs for the document itself. An example document from the collection is shown in Appendix A.

Each device is given three different bloom filter inputs. The first is the "normal" bloom filter. This bloom filter is actually generated from the profiles and will result in the same classification as the non bloom filter runs.

The second is the "all hits" bloom filter. This bloom filter is created to be completely full and thus always result in a hit, irrespective of whether or not the term is in the profile. This is seen as a worst-case scenario performance- wise when running with a bloom filter, the case where no reads from main memory are prevented. This will result in the same classification as the non bloom filter runs.

The third is the "no hits" bloom filter. This bloom filter is created to be completely empty and thus never results in a hit, irrespective of whether or not the term is in the profile. This is seen as a best-case scenario performance wise when running with a bloom filter, the case where all reads from main memory are prevented. This results in a completely zero score classification for all documents however, alongside the "all hits" bloom filter, results in a range of performance values possible when using a bloom filter.

4.3 Data Transfer Results

Figure 4.1 shows the data transfer speeds between host and device on the Mac Pro system. The theoretical maximum bandwidth of the system is 4GB/s and, with large enough transfers, around 3.2GB/s (80% of theoretical) is achieved. This is a lightly loaded system so bus contention isn't a factor during these experiments.

Transfers that are less than 100 kilobytes, for instance the 4KB bloom filter, suffer very low transfer speeds, with the bloom filter being transferred at around 850MB/s. With the next smallest buffer to be transferred being on the order of hundreds of kilobytes (scoring results), the slow transfer rate is not going to cause significant impact on overall performance.

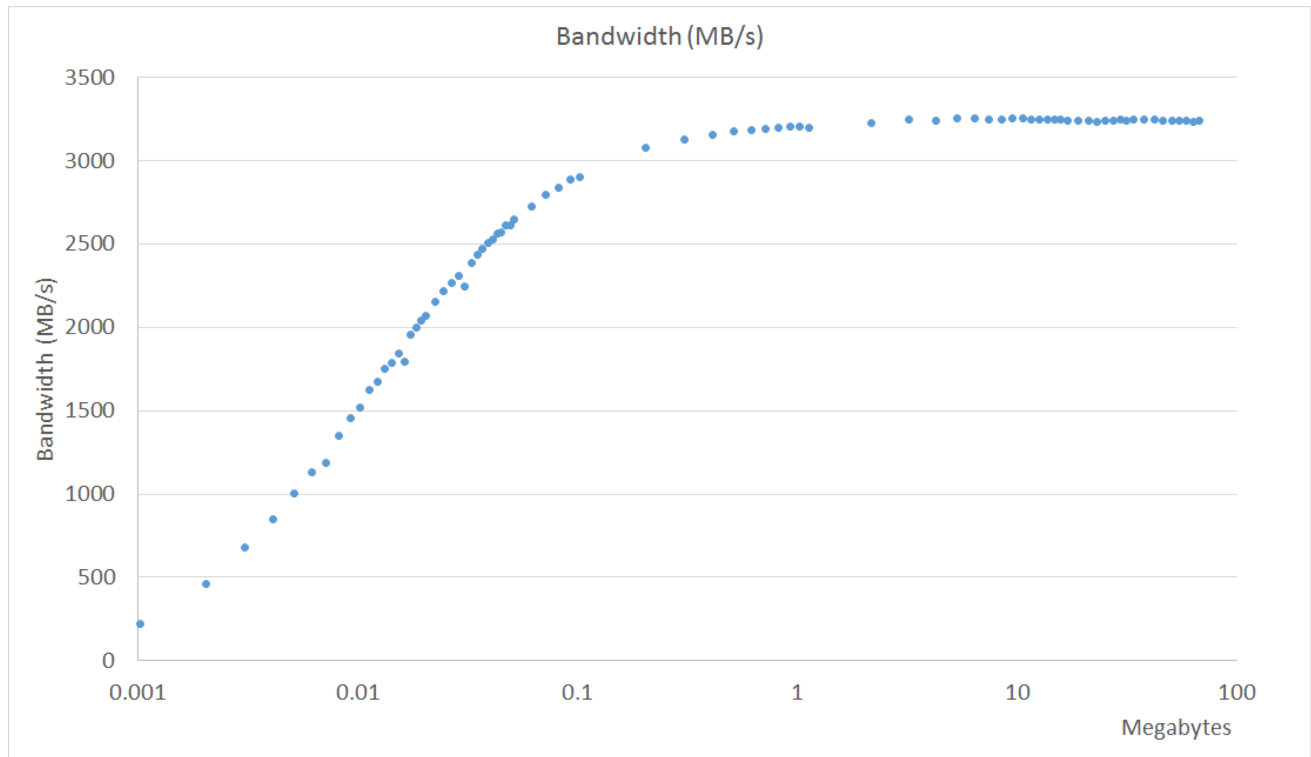

Figure 4.1: PCI Express bus transfer rates

4.4 Scoring Only Results

This experiment is to investigate how well GPUs score documents, having been given a parsed document set. Scoring is a highly parallel task thus GPUs are suited for this kind of task.

Table 4.2 contains the results from the scoring only experiments. The results are the average time and throughput of the system over the twelve profiles.

Each profile run is repeated ten times to improve the accuracy and reliability of the wall clock times. The exception to this rule is the AMD system which, due to its significantly higher throughput, required one hundred repetitions per profile to achieve reliable results.

The C++ Single Threaded and C++ Multi-threaded results are there for reference and to indicate how effective parallelism is for document filtering.

Device	Test	Time (ms)	Throughput (MB/s)
2x Intel Xeon E5462 C++ Single Threaded	No Bloom Filter	2537	93
	Bloom Filter	2617	90.2
	Bloom Filter All Hits	4023	58.7
	Bloom Filter No Hits	2006	117.4
2x Intel Xeon E5462 C++ Multi-threaded	No Bloom Filter	390	605.1
	Bloom Filter	336	702.4
	Bloom Filter All Hits	550	429.1
	Bloom Filter No Hits	257	918.3
2x Intel Xeon E5462 OpenCL	No Bloom Filter	222	1063.1
	Bloom Filter	156	1512.8
	Bloom Filter All Hits	298	791.9
	Bloom Filter No Hits	104	2269.2
nVidia Tesla C2075 OpenCL	No Bloom Filter	258	914.7
	Bloom Filter	232	1017.2
	Bloom Filter All Hits	294	802.7
	Bloom Filter No Hits	216	1092.6
Intel Xeon Phi 5110P OpenCL	No Bloom Filter	381	619.4
	Bloom Filter	409	577
	Bloom Filter All Hits	456	517.5
	Bloom Filter No Hits	402	587.1
4x AMD Opteron 6366 HE OpenCL	No Bloom Filter	87	2712.6
	Bloom Filter	50	4720
	Bloom Filter All Hits	96	2458.3
	Bloom Filter No Hits	39	6051.3

Table 4.2: Scoring Only Results

Figure 4.2 takes each device's best result and compares them in a bar graph. The AMD system has the clear lead, having nearly three times the performance of the second fastest system. It's important to note however that the CPU figures are only used for reference, there would need to be some part of the system conducting the parsing section making the scoring slower.

The Tesla C2075 GPU can score in excess of 1GB/s showing that the CPU parsing code is the bottleneck of the system, running at 427MB/s [Van13].

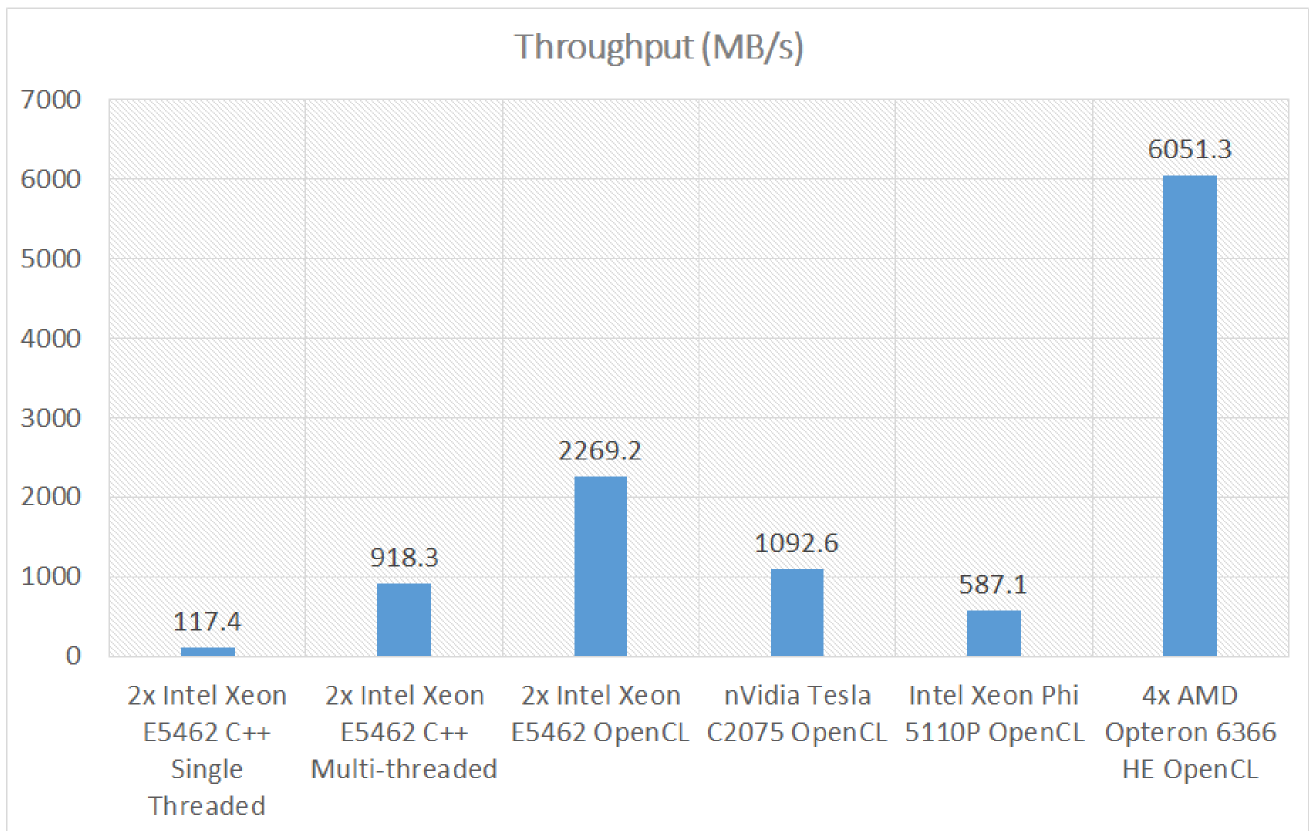

Figure 4.2: Scoring Only Device Comparison

4.5 Parsing and Scoring Results

Having shown that scoring on the GPU is faster than the parsing on the CPU (427MB/s [Van13]), the parsing and scoring experiment was set up to investigate if GPUs and the Intel Xeon Phi are suitable for both parsing and scoring, in an attempt to remove the CPU parsing bottleneck.

Table 4.3 contains the results from the parsing and scoring experiments. The results are the average time and throughput of the system over the twelve profiles. All experiment runs involved OpenCL without using any of the pre-existing C++ code.

Each profile run is repeated ten times to improve the accuracy and reliability of the wall clock times. The exceptions to this rule are the Intel CPU and nVidia GPU system which, due to the fact that both devices were running simultaneously and run at slightly different rates, required different numbers of repetitions to finish at the same time. The “No Bloom Filter” experiment required twenty repetitions in total (ten for each device), “Bloom Filter” and “Bloom Filter No Hits” required twenty three repetitions (CPU had three extra), and “Bloom Filter All Hits” required twenty two repetitions (CPU had two extra). The experiment with the two Intel Xeon Phi 5110Ps had twenty repetitions in total, ten for each device.

Device	Test	Time (ms)	Throughput (MB/s)
2x Intel Xeon E5462	No Bloom Filter	699	337.6
	Bloom Filter	648	364.2
	Bloom Filter All Hits	782	301.8
	Bloom Filter No Hits	590	400
nVidia Tesla C2075	No Bloom Filter	649	363.6
	Bloom Filter	800	295
	Bloom Filter All Hits	876	269.4
	Bloom Filter No Hits	756	312.2
2x Intel Xeon E5462 & nVidia Tesla C2075	No Bloom Filter	386	611.4
	Bloom Filter	401	588.5
	Bloom Filter All Hits	455	518.7
	Bloom Filter No Hits	379	622.7
Intel Xeon Phi 5110P	No Bloom Filter	499	472.9
	Bloom Filter	514	459.1
	Bloom Filter All Hits	568	415.5
	Bloom Filter No Hits	524	450.4
2x Intel Xeon Phi 5110P	No Bloom Filter	279	845.9
	Bloom Filter	289	816.6
	Bloom Filter All Hits	313	754
	Bloom Filter No Hits	280	842.9
4x AMD Opteron 6366 HE	No Bloom Filter	239	987.4
	Bloom Filter	450	524.4
	Bloom Filter All Hits	474	497.9
	Bloom Filter No Hits	426	554

Table 4.3: Parsing and Scoring Results

Figure 4.3 takes each device's best result and compares them in a bar graph. From a single device perspective, the AMD System takes the lead easily. For the dual device perspective, the two Intel Xeon Phi 5110Ps come out top, with just over 800MB/s throughput. This is still beaten by the single device AMD system however, given that the AMD system has 64 CPU cores at 2.3GHz versus the two Intel Xeon Phis lower clocked 120 cores at around 1GHz, this is as expected.

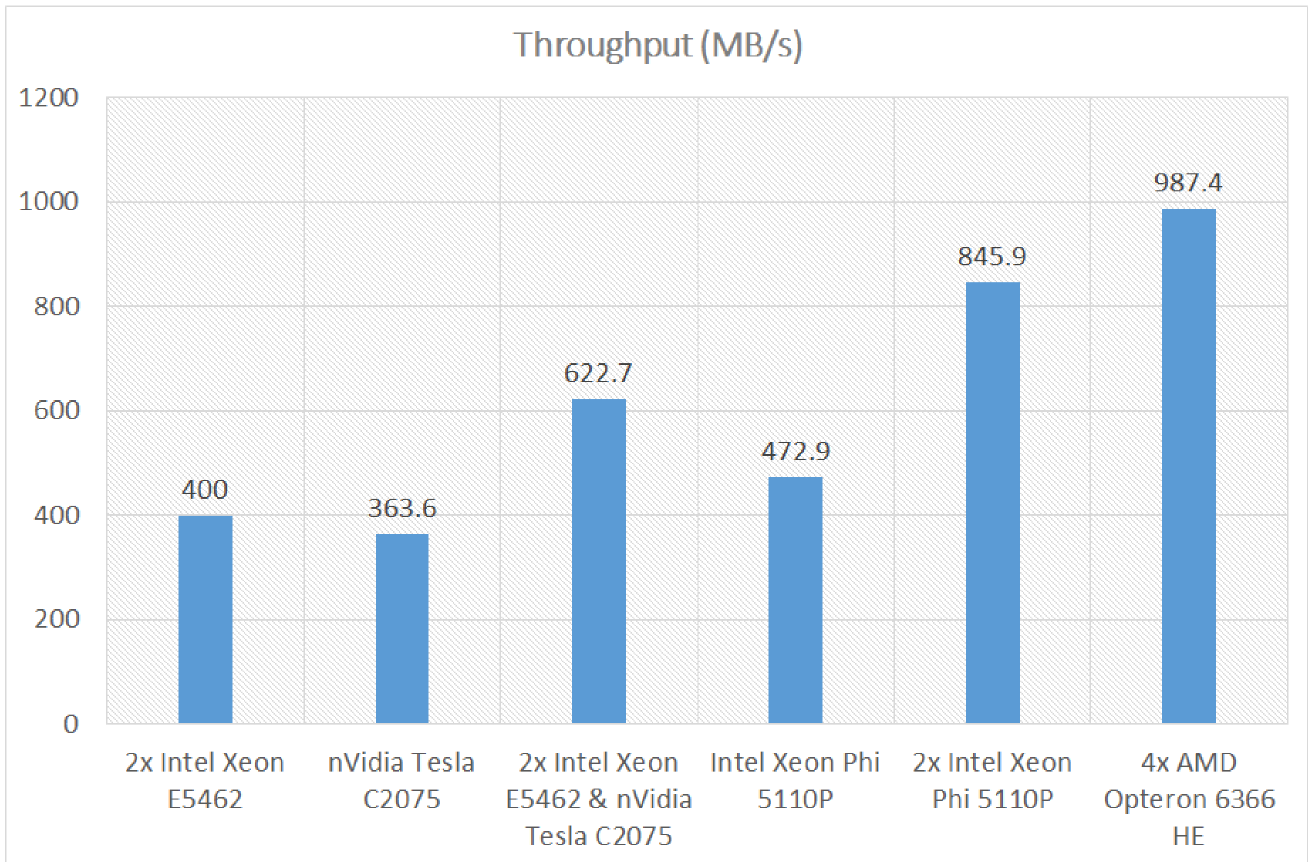

Figure 4.3: Parsing and Scoring Device Comparison

4.6 Discussion of Results

4.6.1 Data Transfer

Data transfer rates between host and device are often a source of performance issues when using accelerator chips. In the document classification system, around 300MB is required to be transferred for each kernel invocation. Transfer times, as measured during actual kernel executions, are around 110 milliseconds.

For the scoring only results, this is up to half of the total time spent to score some documents. For instance the nVidia Tesla C2075 bloom filter experiment has a scoring time is 232ms. This is a significant overhead. Given that the PCI express bus speed was the original 1.0 specification with 4GB/s maximum throughput, moving to a graphics card with similar compute performance, but with PCIe 3.0 capabilities would reduce the data transfer overhead to around 30ms, improving overall time to 152ms, a reduction of around one third.

For the parsing and scoring results, the transfer time only represents at most one fifth of the total time. Consider the Intel Xeon Phi, the no bloom filter experiment runs in 499ms. This is still a substantial overhead however, the move to PCIe 3.0 would only reduce this time to around 429ms, a throughput improvement of around 16%.

The simple reason behind the differences in effect of transfer rates on performance between the scoring, and parsing and scoring code is that in the latter, the OpenCL kernel spends more computational time on each byte of data transferred. Both scoring, and parsing and scoring require a similar amount of data to be transferred yet

the latter does more work thus parsing and scoring on the one device is the better option based on data transfer overhead alone. It is also worth noting again that due to the speed of parsing on the C++ system being low compared to the scoring speed on a GPU, the improved data transfer rates would only reduce time in getting classification results back, but not improve throughput overall given that the parser is the bottleneck. For the parsing and scoring kernel, the data transfer time reduction is beneficial as the CPU isn't the bottleneck given that it has no real computation to execute.

4.6.2 C++ versus OpenCL CPU throughput

Although the scoring only experiments on the CPU were initially only for reference, they also highlight a potential benefit of running a highly data parallel algorithm on the CPU using OpenCL. The C++ version uses POSIX threads to become multi-threaded and, with eight hardware threads, experience up to 7.82x better throughput compared to the single threaded version. This is as expected, the maximum improvement is 8x however threading will inevitably result in a bit of overhead. The surprise arrives when looking at the OpenCL version of the scorer, it has around twice as high throughput compared to the multi- threaded C++ version, thus making it nearly 20x quicker than the single threaded version.

Looking at the results alone, this initially seems unusual however can be explained. With POSIX threads, the document set is split evenly into eight sets, one for each hardware thread. In terms of scheduling and memory access, this is as granular as things go, each thread goes through its documents one by one. For OpenCL, it has the same number of work items as there are documents. For the TREC data set, this is 77876 documents. OpenCL is free to schedule work groups on the CPU cores allowing for more granular scheduling. Each work group has a local size of 1 so this is equivalent to scheduling individual documents to CPU cores. This can enable the runtime system to better hid memory access latency, resulting in the higher throughput. Also, given that documents can be of different lengths, if a POSIX thread has significantly longer documents than its counterparts, it will still be working away while the others have finished. This can still happen on the OpenCL system but with the granularity being significantly finer, it can be mitigated as the cores with the work items parsing or scoring shorter documents are free to take more work on.

4.6.3 Effect of Bloom Filters

The bloom filters were designed and added to the OpenCL kernel with the idea that they will be able to save trips to the on-board memory for the GPU. With GPUs having manual caching, the OpenCL kernel is responsible for moving the bloom filter from global memory to local memory. The benefit of this is that the bloom filter is guaranteed to be in the faster memory. This contrasts with the CPU where the bloom filter can still bring benefit by migrating to L3 or L2 cache on the CPU but, since CPUs implement automatic caching, this is not guaranteed to happen.

When a variable is moved in to cache, it isn't just that data value that is read in, an entire cache line is read in. A cache line is typically on the order of tens of bytes, with 64 bytes being the cache line size on all devices used. The idea behind this is locality of reference, if a value at address x is read from, it is likely the same process will need addresses near by in the near future. The benefit of this is for the scoring only code, a read of one term will read in the next seven terms (8 terms are 64 bytes) into cache making subsequent reads faster. This also benefits the parsing and scoring only code, a read of one character will read in the next sixty three characters (64 characters are 64 bytes) making subsequent reads faster. This also holds true for GPUs as reads from memory are done in chunks of around 64 bytes.

Scoring Only

Use of the bloom filter when running the scoring only code benefits almost all devices, with the Intel Xeon Phi being the exception. Even on the single threaded C++ code, a 26% performance increase is observed in the best case (the "no hits" bloom filter). With a realistic application of the document filtering being the case where most documents, and thus terms, are not of interest to the classifier, the "no hits" bloom filter is not only the best case, but is close to the average case as well.

Both the Intel and AMD CPU systems experience just over a factor of two increase in performance when making use of the bloom filter in the best case. This is strong evidence to suggest that the bloom filter is being moved into L3, or L2 cache and staying there for most of the kernel execution. This means the bloom filter can be checked without going to primary memory, and if the bloom filter returns a "miss" then the primary memory won't be read from at all.

The nVidia GPU has a modest performance increase of around 11% with the normal bloom filter, and 19% in the best case. If one was to compare solely with the CPU this may seem disappointing however the performance increase is still significant, especially considering it's all from a 4KB buffer with little computational overhead. Also, since there is no automatic caching, the benefit of caching is limited by the memory model, and the kernel code itself.

The Intel Xeon Phi has peculiar results compared to the other device, the bloom filter results in a performance drop of around 5%. This is perhaps due to the complicated cache mechanism required given the high number of independent cores. Each core has its own, small L2 cache thus the bloom filter is unlikely to be present in all of the cores simultaneously thus the potential benefit of the bloom filter is small. It is possible for a single copy of data in cache to be available for all cores, thus maximising cache capacity as the bloom filter is only stored once, but this makes access significantly slower. Since communication between cores and access to memory is dictated by a ring structure between cores, following a shortest distance algorithm [7], this set up would appear to be unsuitable for a bloom filter whose benefit relies on fast local access.

As discussed previously, the scoring code runs significantly faster on all devices in comparison to the parser, thus it is the parser which is the system bottleneck. Improvements in overall system performance can thus be obtained by having the parsing and scoring executed on the same device.

Parsing and Scoring

Use of the bloom filter provides no guarantee of improved throughput results when looking across different devices. The Intel CPU benefits by up to 20% in the best case whereas all other single device runs experience a performance hit of as little as 5% with the Intel Xeon Phi to as much as 44% with the AMD CPUs. Overall, the benefit is reduced or mitigated entirely in comparison to the scoring only kernel.

Considering the Intel CPU system initially, it is the only system which still has a performance improvement of up to 20%. In the scoring code the performance improvement exceeded 100%. The bloom filter is used no more and no less than in the scoring only code. The two primary reasons for the reduced improvement are related to the amount of extra work in this kernel, and the extra variables required. As was identified earlier, the parser is the bottleneck. This fact will remain true even in this implementation which parses terms and scores them as soon as a full term has been parsed. This means that, in terms of instruction count, a reduced proportion is in the part of the code which would benefit from the bloom filter. As such it is natural that, percentage wise, throughput will not improve as much with a bloom filter. The second relates to cache use itself. The scoring only code had

[7] `http://www.intel.co.uk/content/dam/www/public/us/en/documents/datasheets/xeon-phi-coprocessor-datasheet.pdf` Page 10

very little in terms of private variables; it read in a term, created three n-grams, had a few variables to denote term range, score, and work item ID. This meant that less cache space was being used by private variables, increasing the chance that the bloom filter will not only migrate to CPU cache, but stay in cache for an extended period of time. The parsing and scoring kernel has around twice as many private variables, making fitting a bloom filter into the cache unlikely to be possible for an extended period of time.

The AMD CPU experiences the biggest performance drop of nearly 50%. This contrasts heavily with the larger than 100% increase in performance for scoring only code. Everything discussed for the Intel CPUs is relevant to the AMD CPUs. The AMD CPU has 16MB of L2 cache in comparison to the 12MB on the Intel CPUs however, with the AMD CPUs core count being 16 compared to the Intel's count of 4, the amount of L2 cache per core is three times higher (3MB) on the Intel CPU compared to the AMD CPU (1MB). This makes the situation of cache demand even more severe and is likely to contribute to the heavy performance drop. The bloom filter will be being read in regularly, removing other variables, just for those variables to be read in shortly after, removing the bloom filter. This is similar to the situation where primary memory and the swap space on disk in a virtual memory system are continually being read from and written to. This is known as thrashing and is responsible for significant drops of performance, similar to those experienced by the AMD system.

The nVidia GPU experiences around a 20% drop in performance in the best case by using a bloom filter. Even with the GPUs local memory being under control of the programmer, the extra demand of the local memory by the parser can have similar effect as on the CPU. The extra variables will affect the occupancy of a compute unit as less work items can be assigned simultaneously. The reduced number of work items per compute unit could also be affecting the memory latency technique effectiveness, resulting in the drop in performance. Also, unless all threads in a warp experience a miss, the warp will have to wait for the few threads with a hit to retrieve their value from memory, removing all benefit from the bloom filter miss by the other threads.

The Intel Xeon Phi again has around a 5% performance drop, identical to the scoring only version. With their being no difference in the bloom filter performance change, the reasons discussed in the scoring only section are just as valid for the parsing and scoring section.

In general, the bloom filter has a negative affect on most devices and could easily be ignored in a real world implementation. Given that the Intel CPU has some benefit from a bloom filter, it could still be beneficial to try the bloom filter on the target system as a performance benefit on some devices is evidently still applicable.

4.6.4 Overall

In previous work, the parser was running on the CPU at 427MB/s [Van13] and was the bottleneck of the system when scoring was conducted on an FPGA. This is also true with the GPU however, if both the CPU and GPU are parsing and scoring their own set of documents, it is possible to achieve in excess of 620MB/s, around a 45% increase in performance on a fairly typical two CPU and one GPU system. In comparison to the CPU and GPU results individually, there is only around a 15% overhead when running the devices in parallel. Most of this overhead comes from the CPU time required to detect the beginnings of documents before sending the data off to the GPU. If the CPU was used solely to read in documents from a network feed and work out the locations of document beginnings, it's likely the case that in a multi-GPU set up, the GPUs could each be used to parse and score, with the CPU host having plenty of cycles free to coordinate the system. Also, the PCIe bandwidth is unlikely to be a bottleneck as the GPUs can be transferring data and executing kernels at slightly different times meaning a multi-GPU set up is unlikely to have significant overhead and scale almost linearly with the number of devices. This can be shown in the dual Intel Xeon Phi runs, which experience a near linear increase (80%) compared to the single Intel Xeon Phi. This is with both devices requiring documents at the same time, and would represent the worst case in a real life application. In reality, the devices would be at different stages of classification meaning CPU cycles are more likely to be used more evenly over a set time frame, rather than bursts of high activity, followed by period of little activity while both devices are parsing and scoring.

Chapter 5

Conclusion

5.1 Summary of Results

The use of GPUs, and OpenCL in general, for document filtering has potential benefits when it comes to performance. The Mac Pro System was able to parse and score documents using the GPU only at a rate of 363.6MB/s. The Tesla C2075 is the enterprise grade equivalent of a mid-range consumer graphics card so any reasonably modern desktop machine would have a similar throughput. To place the throughput rate in the context of classifying documents in a network stream, 363.6MB/s is 2.84Gbit/s. If this machine was asked to classify an Ethernet stream, it could handle nearly 10Gbit/s Ethernet connection on its own as average Ethernet utilisation is typically no more than 30%. By using the CPU as well, the 622.7MB/s (4.86Gbit/s) throughput could handle a 15Gbit/s Ethernet connection. If, instead of a network feed, the system was asked to filter documents from disk then the GPU could filter the documents in real time from a hard drive where sustained read speeds are typically less than 150MB/s. The CPU and GPU hybrid system could filter documents in real time from a solid state drive where read speeds don't exceed 550MB/s.

The use of bloom filters is a mixed bag when it comes to improving performance, as such the recommendation would be to investigate the performance difference when targeting a hardware set up as different architectures respond more positively to the data structure, even different variants of an architecture (Intel Xeon and AMD Opteron CPUs) react differently to one another.

The Intel Xeon Phi was not the primary focus of the work, the GPU was the primary hardware target. The Intel Xeon Phi has solid performance for a single device, reaching 472.9MB/s (3.69Gbit/s) meaning it can filter a 10Gbit/s Ethernet connection in its own right.

Document filtering is a suitable application for OpenCL and GPU acceleration. With each device able to classify its own distinct set of documents, the throughput of a single system can scale well with the number of devices on a single system.

5.2 Future Work

On a single system, there can be upwards of four GPUs which can each independently score documents. With modern GPUs being significantly more powerful than the nVidia Tesla C2075, which achieves a throughput of 363.6MB/s, it wouldn't be hard to envisage upwards of 1GB/s per GPU, for instance, an nVidia GTX Titan or GTX 780 Ti system. The move to nVidia's GeForce line (and equivalent consumer lines for other manufacturers)

is acceptable in this scenario as double floating point precision and other enterprise class features are not required in the document filtering system. A single system can handle four GPUs so it could then process documents at a rate of 4GB/s. Given the targeted application being the classification of documents on a network stream and that Ethernet utilisation is rarely in excess of 30% of capacity for an extended period of time, this 32Gbit/s rate is enough to handle 100Gbit/s Ethernet. As such, further work would be to make use of a more modern, high end multiple GPU set up to investigate the performance that can be achieved from devices attached to a single motherboard.

Neither the data structures or the algorithm were designed with the Intel Xeon Phi architecture in mind thus the performance of the device has the potential for improvement. With an in-depth review of the inner workings of the architecture, a more efficient implementation is perhaps possible.

With Wim Vanderbauwhede et al. having implemented scoring code on the FPGA, with the view of also implementing the parsing code on an FPGA [Van13], the success of the parsing and scoring implementation on the GPU further suggests significant benefits can be obtained by doing the same for an FPGA.

In addition to the above, a detailed analysis of both the performance per watt and performance per dollar values of each set up will be of benefit to large organisations such as ISPs, who would be implementing such a system. In addition to the system's throughput, for-profit organisations also place significant emphasis on systems which are cheap to buy and run.

Appendices

Appendix A

Example TREC Document

```
<DOC>
<DOCNO> APW19990101.0001 </DOCNO>
<DATE_TIME> 1999-01-01 13:31:15 </DATE_TIME>
<BODY>
<CATEGORY> usa </CATEGORY>
<HEADLINE> Former NBC Chairman Adams Dies </HEADLINE>
<TEXT>
<P>
NEW YORK (AP) -- David Adams, a former NBC chairman who was
regarded as one of television’s most popular executives, has died.
He was 85.
</P>
<P>
Adams died Sunday at Cedar Manor Nursing Home in Ossining, N.Y.,
located north of New York City in Westchester County. He was a
resident of nearby Croton-on-Hudson.
</P>
<P>
He joined the network’s legal staff in 1947, after working for
the Federal Communications Commission in Washington, D.C., and
serving in the Army. He went on to assume responsibility for the
network’s legal department, its Washington office, program
standards and practices, research and planning, advertising,
publicity and promotion, and relations with affiliated stations.
</P>
<P>
Following a leave from NBC in 1969, when he was senior executive
vice president, Adams returned as a member of the company’s board
and as executive vice president of its corporate staff. He was
appointed chairman in the early 1970s and served as vice chairman
from 1975 until his retirement in 1980.
</P>
<P>
He is survived by his wife, Ilyana, of Croton-on-Hudson; a
step-daughter, Kristina Lanin-Stufano of Hamden, Conn.; a
daughter-in-law, Audrey Adams of Scarsdale; two grandchildren and
two step-grandchildren.
</P>
</TEXT>
</BODY>
</DOC>
```

Bibliography

[989] ISO/IEC 9899, *C99 language standard*, `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf`, [Last Accessed 28th March 2014].

[AKC+00] Ion Androutsopoulos, John Koutsias, Konstantinos V Chandrinos, George Paliouras, and Constantine D Spyropoulos, *An evaluation of naive bayesian anti-spam filtering*, arXiv preprint cs/0006013 (2000).

[APK+00] Ion Androutsopoulos, Georgios Paliouras, Vangelis Karkaletsis, Georgios Sakkis, Constantine D Spyropoulos, and Panagiotis Stamatopoulos, *Learning to filter spam e-mail: A comparison of a naive bayesian and a memory-based approach*, arXiv preprint cs/0009009 (2000).

[Blo70] Burton H Bloom, *Space/time trade-offs in hash coding with allowable errors*, Communications of the ACM **13** (1970), no. 7, 422–426.

[CMV+12] S Chalamalasetti, Martin Margala, Wim Vanderbauwhede, Mitch Wright, and Parthasarathy Ranganathan, *Evaluating fpga-acceleration for real-time unstructured search*, Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on, IEEE, 2012, pp. 200–209.

[CS12] Doris Chen and Deshanand Singh, *Invited paper: Using opencl to evaluate the efficiency of cpus, gpus and fpgas for information filtering*, Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on, IEEE, 2012, pp. 5–12.

[Gro] Khronos Group, *Opencl 1.1 language specification*, `http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf`, [Last Accessed 28th March 2014].

[HLL13] Hua He, Jimmy Lin, and Adam Lopez, *Massively parallel suffix array queries and on-demand phrase extraction for statistical machine translation using gpus*, Proceedings of NAACL-HLT, 2013, pp. 325–334.

[MAP06] Vangelis Metsis, Ion Androutsopoulos, and Georgios Paliouras, *Spam filtering with naive bayes-which naive bayes?*, CEAS, 2006, pp. 27–28.

[MN+98] Andrew McCallum, Kamal Nigam, et al., *A comparison of event models for naive bayes text classification*, AAAI-98 workshop on learning for text categorization, vol. 752, Citeseer, 1998, pp. 41–48.

[Par] Ravi Parikh, *Average number of characters in an english word*, `http://www.ravi.io/language-word-lengths`, [Last Accessed 28th March 2014].

[PS03] Fuchun Peng and Dale Schuurmans, *Combining naive bayes and n-gram language models for text classification*, Advances in Information Retrieval, Springer, 2003, pp. 335–350.

[Sch03] Karl-Michael Schneider, *A comparison of event models for naive bayes anti-spam e-mail filtering*, Proceedings of the Tenth Conference on European Chapter of the Association for Computational Linguistics - Volume 1 (Stroudsburg, PA, USA), EACL '03, Association for Computational Linguistics, 2003, pp. 307–314.

[Ste] Stephen, *Baudot and ccitt character codes*, `http://rabbit.eng.miami.edu/info/baudot.html`, [Last Accessed 28th March 2014].

[Van13] Vanderbauwhede, Wim, Frolov, Anton, Rahul Chalamalasetti, Sai, and Margala, Martin, *A Hybrid CPU-FPGA System for High Throughput (10Gb/s) Streaming Document Classification*, 2013.

[VFA+13] Wim Vanderbauwhede, Anton Frolov, Leif Azzopardi, Sai Rahul Chalamalasetti, and Martin Margala, *High throughput filtering using fpga-acceleration*, Proceedings of the 22nd ACM international conference on Conference on information & knowledge management, ACM, 2013, pp. 1245–1248.