RC25242 (W1111-221) November 30, 2011
Computer Science

IBM Research Report

Accelerating Business Analytics Applications

Valentina Salapura, Priya Nagpurkar, Tejas Karkhanis, José Moreira
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Accelerating Business Analytics Applications

Valentina Salapura, Priya Nagpurkar, Tejas Karkhanis, José Moreira
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598, USA

Abstract

Business and text analytics applications have seen rapid growth, driven by the mining of data for various decision making processes. Regular expression processing is an important component of these applications, consuming as much as 50% of their total execution time. While prior work on accelerating regular expression processing has focused on network intrusion detection systems, business analytics applications impose different requirements on regular expression processing efficiency. We present an analytical model of accelerators for regular expression processing, which includes memory, bus-, I/O-, and network-attached accelerators with a focus on business analytics applications. Based on this model, we advocate the use of vector-style processing for regular expressions in business analytics applications, leveraging the SIMD hardware available in many modern processors. In addition, we show how SIMD hardware can be enhanced to improve regular expression processing even further. We demonstrate a realized speedup better than 1.8 for the entire range of data sizes of interest. In comparison, the alternative strategies deliver only marginal improvement for large data sizes, while performing much worse than the SIMD solution for small data sizes.

1 Introduction

We are experiencing a dramatic growth in the amount of unstructured data that our businesses produce. That data can be free-format text (as in e-mails and documents), data stored in variable-length descriptive formats such as XML, images, voice, video and various other forms. Business analytics applications are an enabling technology that allows us to process and mine these data, extracting value that can be useful for the business.

It is particularly challenging to process these unstructured data in a way that fully leverages all the parallelism that modern processors have made available. A widely cited Berkeley study [1] identifies the finite-state machine (FSM) algorithms associated with text processing as the hardest kernel to parallelize among a set of 13 representative workloads (the Berkeley "dwarfs").

FSM-based workloads constitute a significant fraction of non-numeric applications, and regular expression (regex) matching is one of the most popular examples of FSM-based workloads. Not only is regex matching crucial in text analytics, but it also constitutes the core of Network Intrusion Detection Systems (NIDS) and anti-virus scanners, and is a heavyweight component of search-engine indexers, XML processors, and programming language compilers and interpreters.

In this paper, we focus on the efficient processing of unstructured text through regex matching. Our goal is to identify the most effective way to accelerate regex processing for our chosen problem domain. To this end, we provide an analytical comparison of network-attached, I/O bus-attached, memory bus-attached, and inline-SIMD accelerators in the context of text processing applications for business analytics. Network-attached appliances are specialized engines that are often used to preprocess data streams, simplifying subsequent data processing on the host. Other forms of regex accelerators include specialized hardware that attaches directly to either the memory bus or the I/O bus of the host system, and inline accelerators implemented as functional units inside the processor core.

Although specialized hardware accelerators are the fastest regex engines, and are frequently used in network processors for network intrusion detection, we show that they are not the optimal solution for business analytics applications. Unlike network intrusion detection systems applications, where streaming data have to be checked at wire speed, business analytics applications differ significantly in the way they use regex matching. Different requirements of applications result in different optimal solutions.

We show that SIMD processing, i.e. inline acceleration using the SIMD (single-instruction, multiple-data) unit, is the most effective approach for our problem domain. Wide vector SIMD units are available in several modern processors, including the IBM POWER architectures [2, 3], the Cell Broadband Engine [4], and Intel processors that support streaming instruction set extensions (MMX, SSEx, AVX).

We further explore architectural support to enhance SIMD processing for text analytics. We identify three specific requirements for efficiently implementing inline acceleration through the SIMD units. First, the processor must provide fast data movement between the SIMD and the fixed-point units. Second, the processor must provide an efficient "gather" operation to collect data from different sources in one place. Finally, the processor must have facilities to support data streaming. We discuss the rationale of these requirements, and measure the impact of a possible gather operation.

In summary we make the following contributions. First, we present an analysis of text analytics applications. Second, we develop an analytical model of different types of accelerators and an exploration of tradeoffs for text analytics applications. Finally, we discuss architectural enhancements to improve the performance of the in-core SIMD unit for better inline acceleration of regular expression processing in text analytics applications.

The remainder of this paper is organized as follows. Section 2 is an overview of the characteristics of text processing in business analytics applications. Section 3 presents a performance study of the different acceleration options using analytical models. Section 4 described an efficient implementation of FSM code using SIMD processing, while Section 5 discusses the architectural support that we identify as being important for efficient inline acceleration through SIMD processing, and we evaluate its effectiveness in Section 6. Finally, Section 7 discusses related work and Section 8 presents our conclusions.

2 Business analytics applications

In this Section, we discuss both the extent and the way in which regex matching is used in business analytics applications. In these applications, the most important application of regex matching is in *tokenizers*. A tokenizer divides a stream of input characters into distinct words, according to a predefined set of regular expressions (regex). Classically known from compilers and interpreters, tokenizers are also gaining attention as the first stage of any search engine indexer. They are also components in XML processing tools, where they consume 30% or more of the execution time [5, 6]. In this section, we focus on a full text analytics application built using IBM's LanguageWare technology, deployed within the Apache Unstructured Information Management Architecture (UIMA) framework [7].

The Apache UIMA framework is an implementation of the UIMA specification, which defines data representations and interfaces to enable analytics (text analytics solutions) to inter-operate. The IBM LanguageWare product is a set of Java libraries that provides natural language processing features such as language identification, tokenization, relationship extraction, and semantic analysis. Internally, LanguageWare relies on FSM-based algorithms and techniques that are similar to those used in regex pattern matching, and the focus of our acceleration efforts.

The text analyzers we study take as input unstructured text (*e.g.*, news articles, e-mails) and convert it to annotated or structured data. Text analyzers that annotate unstructured data are called *annotators*. The end goal is to detect items of interest in the text analyzed, ranging from simple patterns like e-mail addresses to complex relationships like one company acquiring another. The annotators we use are freely available as part of the LanguageWare workbench [8]. We performed detailed analysis of one of these annotators (namely the *Acquisitions Annotator*) by deploying it within the Apache UIMA framework. We use it to find all information related to acquisitions in a collection of news articles. We collected runtime profiles using oprofile to measure the time spent in different parts of the Aquisitions Annotator. Figure 1 shows the Java-level cycle breakdown for the workload: the application spends approximately 50% of its time in the FSM code that we are aiming to accelerate.

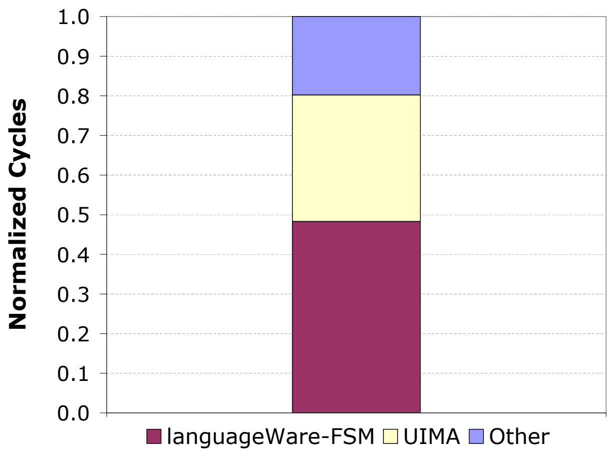

Figure 1. Cycle breakdown for the Acquisitions Annotator in the Languageware workbench. Approximately 50% of the time is spent in FSM processing.

An important factor in the performance of regular expression processing is the size of the input data. In today's world, there are different sources of unstructured text, which are candidates for analysis. Table 1 shows three popular sources, the range for typical message sizes in column two, and the average size in column three. We computed the average sizes as follows. For Twitter, we use statistics for the top 200 Twitter users [9]. For e-mails, we use over

Source	Typical size range (bytes)	Average size (bytes)
Twitter posting	10–140	92
E-mail message	100–10,000	2,456
Web page	10,000–100,000	58,608

Table 1. Message sizes for different popular sources of unstructured data.

Accelerator type	Overhead (cycles)	Bandwidth (Gbit/s)
In-line SIMD	50	2
Memory bus	1,000	5
I/O bus	10,000	20
Network	50,000	50

Table 2. Typical overhead (assuming a 4 GHz clock rate) and bandwidth for the various types of accelerators.

200,000 e-mails from the publicly available Enron e-mail dataset [10]. Finally, for webpages, we use webpages from one Wikipedia dataset [11]. The news articles we used with the Acquisitions Annotator fall in the range covered by e-mails and web pages. Although, in some cases, multiple messages can be combined in a larger batch, true on-line processing requires that each message must be processed as it is generated (or received).

In addition to text analytics and NIDSs, regex are widely used by text editors and tools like *grep*. Modern programming languages also provide APIs, built on regex matching engines, to simplify text processing. For example, many Java web applications use the `java.util.regex` package, both in client- and server-side code, to manipulate text-based content. Our analysis of two such Java enterprise applications, namely IBM Lotus Connections Blogs and Dogear, revealed that respectively 16% and 5% of the application's time is spent in the `java.util.regex` package.

3 Approaches to acceleration

In this Section, we present a simple analytical model of the different alternatives for accelerating regex processing to weigh the benefits and tradeoffs associated with each. We identify the following broad types of accelerators: inline SIMD, memory bus-attached, I/O bus-attached, and network-attached. We characterize each type of accelerator by their maximum processing rate and by their activation overhead.

Let B_{base} be the rate at which the base (unaccelerated) processor can perform regex matching. That is, the base processor can process b data bits in $T_{\text{base}}(b) = b/B_{\text{base}}$ seconds. Let B_{acc} denote the peak processing rate (in bits per second) of a regex accelerator. Accessing an accelerator in a system, however, incurs an overhead that increases the processing time and consequently reduces the effective bandwidth. If we denote the overhead incurred to start and finish the processing in the accelerator with t_{oh}, the total time to process b bits in an accelerated system is given by $T_{\text{acc}}(b) = b/B_{\text{acc}} + t_{\text{oh}}$.

Table 2 summarizes the characteristics (maximum rate and overhead), based on specifications of existing and upcoming systems, for each of the accelerator types. Cascaval et al [12] provide a more detailed description and taxonomy for different types of accelerators. We use the rate of 2 Gbit/s for inline SIMD acceleration, as demonstrated in [13]. Before the inline acceleration can begin, the data must be brought into the register file used by the accelerator. In the assumed implementation, the inline SIMD accelerator does not directly use the general purpose registers. Furthermore, the load/store unit in the general purpose core cannot use the vector register as the address register. Therefore, loading and storing incurs a penalty in terms for dependent scalar instructions. This results in an overhead of around 50 processor clock cycles.

A memory bus-attached accelerator is more loosely coupled compared to the inline accelerator. The instructions for a memory bus attached accelerator are typically not part of the general purpose processor's instruction set architecture. Instead, scalar instructions from the program are executed on the processor to setup the control information required by the memory bus-attached accelerator to process the input data. Overall, startup and finish overheads for accessing the memory bus-attached accelerator are around one thousand processor clock cycles [12]. Because of their specialized nature, memory bus-attached accelerators do achieve much higher processing rate than a general purpose core, as indicated by the 5 Gbit/s processing rate in Table 2.

I/O bus-attached accelerators are invoked through the operating system and require I/O operations. They can be implemented as entire ASICs or collections of ASICs, as the Cavium Networks and Raza Microelectronics systems [14, 15], and achieve processing rates of up to 20 Gbit/s. Invoking these accelerators typically imposes an overhead of approximately 10 thousand processor cycles.

Finally, network-attached accelerators are connected to the system requiring acceleration via a network interface supporting one or more networking protocols. Typically, these accelerators have a higher peak bandwidth (up to 50

Gbit/s) with a bigger mechanical and electrical footprint than the accelerator types described above. The benefits of network-attached accelerators are that these accelerators are not tied to a specific system, and that they can be shared among multiple systems. The disadvantage of network-attached accelerators is the high overhead, of around 50,000 clock cycles, required to go through operating system and networking software.

In the remainder of this section, we present an analytical model that allows understanding the trade-offs associated with each type of accelerator, in comparison to the baseline scalar regex code. Using the bandwidth and overhead concepts introduced above, the speedup $S_{\mathrm{RE}}(b)$ from using the accelerator to process b bits is given by

$$S_{\mathrm{RE}}(b) = \frac{T_{\mathrm{base}}(b)}{T_{\mathrm{acc}}(b)} = \frac{b/B_{\mathrm{base}}}{b/B_{\mathrm{acc}} + t_{\mathrm{oh}}} = \frac{B_{\mathrm{acc}}/B_{\mathrm{base}}}{1 + t_{\mathrm{oh}} B_{\mathrm{acc}}/b} \quad (1)$$

Equation (1) provides several insights. First, as t_{oh} increases, the speedup $S_{\mathrm{RE}}(b)$ decreases. Therefore, system designers should lower the overhead for starting and finishing the acceleration process. If care is not taken to reduce this overhead, the acceleration might not provide significant speedup.

Second, as b increases, $S_{\mathrm{RE}}(b)$ tends to the ideal speedup of $B_{\mathrm{acc}}/B_{\mathrm{base}}$. Therefore, it is important that the acceleration process is initiated for as large values of b as possible. Making b large can be difficult in some applications.

Third, as B_{acc} tends to infinity, $S_{\mathrm{RE}}(b)$ approaches $\frac{b}{t_{\mathrm{oh}} B_{\mathrm{base}}}$. Thus, the value of B_{acc} does not solely determine the performance of the system with an accelerator. The parameters t_{oh} and b are just as important. If the designer has the freedom to vary b and t_{oh}, the ratio $\frac{b}{t_{\mathrm{oh}}}$ must be much greater than B_{base}.

While applications such as network intrusion detection process regex expressions for the vast majority of the application time, business analytics applications require regex processing only a fraction of the time. Let f_{RE} be the fraction of time that an application spends on regex matching when running on a base processor. Then, the end-to-end speedup observed by the applications is given by:

$$S_{\mathrm{e2e}}(b) = \frac{1}{(1 - f_{\mathrm{RE}}) + \frac{f_{\mathrm{RE}}}{S_{\mathrm{RE}}(b)}}, \quad (2)$$

We use Equation 2 to generate speedup numbers for each type of accelerator. The parameter f_{RE} is set to 0.5, representative of the typical business text analytics applications we examined in Section 2. From our in-house experiments we observe that regex processing, when performed on a (unaccelerated) general purpose processor with general purpose libraries such as PCRE (Perl-Compatible Regular Expressions) achieves a throughput of 80 Mbps. We therefore adopt that value as B_{base}. The values for parameters B_{acc} and t_{oh} depend on the type of accelerator.

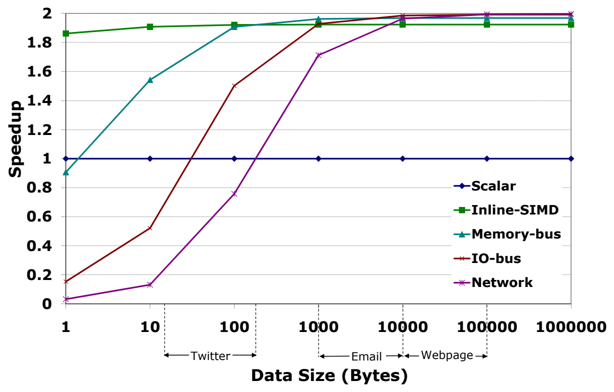

Figure 2. End-to-end application speedup (scalar code = 1) as a function of message size for different types of accelerators.

Results of the speedup model for various regex acceleration options are shown in Figure 2. The figure has five curves, one curve for the baseline general purpose processor without acceleration, and four curves for each of the four types of accelerators. The x-axis shows the data size that is processed with every call to the regex accelerator. The y-axis has the speedup achieved for each type of accelerator. For each type of accelerator seven data points are computed with Equation 2. Lines going through the seven points illustrate the speedup trend as the data size varies. The x-axis is annotated with the range of data sizes provided by each of the three aforementioned common data sources discussed in Section 2.

Results show that for text analytics applications, the end-to-end application speedup saturates at $2\times$. This is because about only 50% of the time is the application is doing regex processing.

For low values of b, around the data size for text messages on Twitter, inline SIMD provides a much higher speedup than the alternatives. This is because for low values of b the acceleration overhead dominates speedup. Memory-attached, I/O-attached and network-attached accelerators are bottlenecked by their much larger overhead. This observation is consistent with first insight provided by equation 2. Therefore, for processing Twitter data the inline SIMD acceleration provides the most benefit.

For larger data sizes, beyond the data size for web-pages, all curves cluster together. The curve for inline-SIMD saturates at speedup of $1.92\times$. The curve for memory bus-attached accelerator saturates at $1.97\times$. The curves for I/O attached accelerator and network-attached accelerator saturate at $2\times$. All these accelerators achieve more or less the same speedup for large data. Clearly, the accelerators with the highest bandwidth attain highest speedup for large data sizes. Large data speedup increases by 8% going from inline-SIMD to network-attached accelerator.

However, this 8% increase in speedup is achieved at a significant increase in size and power, as the network-attached accelerator is a complete separate system. The memory bus- and I/O bus-attached accelerators also incur in increased size (and power consumption) for marginal gains in performance with large data. Considering, the speedup advantage of inline-SIMD for small data and the ability of inline-SIMD to get within 8% of the speedup upper bound for large data, inline-SIMD is the optimal choice for our purpose of speeding up text analytics.

The effect of limited performance advantage of using faster accelerator for only part of the application is a direct effect of the Amdahl's law. The maximum improvement of the overall application performance is limited by the size of the application that accelerator can improve. As described in Section 2, we measure that the annotator spends 50% of its execution time in the FSM code. This effect is even more pronounced for Java enterprise applications where the FSM code accounts for only 5% to 15% of the overall execution time. For Java enterprise applications, usage of regex accelerators would increase the overall system performance by only 5% or 17%, respectively.

4 Regex matching with SIMD operations

We now take a closer look at accelerating regular expression processing using the SIMD unit. Recent contributions focusing on the network intrusion detection problems have shown multiple approaches to do this [16, 17, 18, 19, 20]. This section shows how this applies specifically to code based on FSMs.

FSM codes used in the vast majority of matching engines perform a transition for each symbol of the input stream. Regex matching is an inherently sequential task, and the code is difficult to parallelize, especially in such a way to expose data-level parallelism and exploit SIMD instructions.

One iteration of a FSM typically uses the current input character and its current state to compute its next state. It then produces the corresponding output, updates its current state, and advances the input. In the implementation model we consider, the input character is loaded from the input stream, while the next state is loaded from a transition table. Finally, output data is stored to the output stream. Traditional code performs these tasks with variables stored in scalar registers and manipulated by scalar operations. (This scenario is deliberately simplistic for explanation purposes. Realistic regexp engines may have multiple input pointers, a more complex state and more elaborate semantic actions.)

While a scalar instruction processes single operands at a time, a SIMD instruction processes multiple operands at a time, provided that they are organized in vector registers. Therefore, the designer can adopt an organization where multiple FSM run at the same time by virtue of SIMD instructions. The FSMs operate on distinct input streams, produce distinct output and have their individual state variables, but they might share the same state-transition table. In this organization, multiple instances of FSM variables (*e.g.*, the current states) are kept in one vector register, and a single block of shared instructions (SIMD whenever possible) performs the above tasks for all the FSMs at the same time.

The reorganization of scalar FSM code into SIMD code often involves a radical redesign, because the now-conjoined FSMs must share the same control flow. In order to fuse the code of multiple FSMs into a single, branchless, SIMD-enabled block of code, designers use a combination of predicated instructions, selection instructions and *speculative writing*. Speculative writing is a technique where, instead of using a branch to select code that either generates output or not depending on a condition, the programmer employs branchless code that selects a destination pointer on the basis of the condition, and then stores to that pointer. When the condition is false, the store deliberately writes data into a discarded location.

Provided that enough independent streams are available for parallel processing, there is no limit to the SIMD width from which this approach can benefit: the wider, the better. The performance achieved, though, depends on multiple factors like the fraction of scalar instructions that have a SIMD version (not all do), the need and cost of moving data from general purpose to vector registers, cache effects, and various others.

We consider as an example the software tokenizer presented in [13]. It runs on the Cell processor [4], a multi-core processor containing one 64-bit PowerPC family Power Processing Element (PPE), and 8 Synergistic Processing Elements (SPEs) [21]. On the SPEs, with 128-bit SIMD registers, the approach runs four 32-bit lanes in parallel, each implementing one FSM. The implementation achieves a throughput of 1.01–1.78 Gbit/s per SPE, of 1.01–1.78 Gbyte/s per Cell chip, as the scaling is linear with the number of SPEs.

Although this work focuses on one architecture (in the experimental section) and one specific regex application, it showcases principles and techniques how to use SIMD for accelerating the FSM code. This approach is portable to other architectures and broader application domains.

5 Architectural support

In this section, we focus on those architectural features that benefit parallel FSM-based algorithms like the one we described in Section 4.

The difficulty in efficient processing of regex comes from their data-dependent memory access patterns. Unlike many HPC applications, where the same calculation is performed independently on a large data amount which are arranged in arrays, a regex processing application changes its flow depending on the current input. In an FSM, the current values of state and input are used to compute memory addresses for operations in the transition table.

Unlike HPC applications, text processing applications do not use separate groups of registers to hold data value, data addresses, and control flow information. Instead, data values are used for both address calculation and for control flow. This problem exhibits a similar memory access pattern as a pointer chasing kernel, and it is similarly difficult to optimize.

For the specific case of the regex computation discussed above, there are two memory lookup operations using addressed contained in vector registers. First, the current input pointers (for the four parallel streams) are used to load the next input characters. Second, the input characters and the current state pointers are used to load the proper entries from the state transition table. Each of the four streams has its own pointers (for input and state) and they are different for each of the four FSMsbeing processed. Therefore, one cannot just use a simple vector load to retrieve the data in parallel.

In order for an architecture to provide a good degree of performance to applications that access data-dependent computed locations, at least one (and ideally all) of the following features must be efficiently supported: (1) support for data gather/scatter functionality; (2) data transfer from vector registers (used in the parallel data calculation) to general-purpose registers (used for address generation); (3) support for streaming data. These three tasks are described below.

Feature (1) fetches data from different addresses and packs them into a single vector register. (This is the gather, the scatter is the reverse operation.) For the case of a 128-bit SIMD unit with 32-bit elements, it requires four loads from four different addresses, calculated as offsets from a base address. One option, adopted by many existing vector architectures such as PowerPC's VMX, is to code this operation explicitly as four vector loads followed by three permutes. We note that these instructions can be implemented efficiently in modern RISC processors. Another option is for architectures to include gather/scatter instructions (such as in Larrabee [22]) which can replace the sequence previously discussed with a single instruction. However, implementing these instructions in hardware is complex and requires multiple cycles to complete, since the same number of loads still has to be issued, and data have to be received and packed accordingly in a single register. The limiting factors in implementing scatter/gather operations efficiently include (i) the bandwidth to the memory system, (ii) the number of loads which can be simultaneously issued to the memory, and (iii) the number of data elements which can be returned to the processor in parallel.

Feature (2) is another way to accomplish the operations above. In parallel FSM-based algorithms, the address of the next state of the FSM is computed from the current value of elements in a vector register. The values in the vector register are frequently used as offsets from a base address. To calculate the next address, values have to be copied from the vector registers to general purpose registers (GPRs). RISC architectures do not support such direct transfers between registers, and require explicit load/store instructions. In recent x86 processors, several SSE4.2 instructions perform these transfers requiring additional cycles to execute. On the Cell, these transfers are not needed at all because the Cell has a unified GPR/vector register file. For the class of algorithms we consider, the support for moving data efficiently from vector registers to GPRs is very beneficial.

Finally, feature (3) is motivated by data usage patterns in these algorithms. Some of the data is streaming and not reused. For example, the input text is accessed only once in a streaming fashion, one character at a time. Similarly, the output tokens are accessed only when they are generated and have no locality. On the other hand, some data has to be accessed frequently. For example, some elements of the state transition table (STT) which define the FSM are accessed frequently and repeatedly. Thus, it is useful to be able to specify parts of the address space as non-cacheable to hold the streaming data. Data accessed from this space (*i.e.*, the input and output streams) will bypass the L1 level cache, and will be loaded in registers only. In this way, streaming data does not pollute the cache and displace STT lines which are constantly reused. The frequently used data can then be kept in the L1 cache. Several modern processors support this feature by marking a cache line "transient", i.e., being the first to evict out of cache.

6 Experimental results

We measured the performance of an implementation of the regular expression processing algorithm described in Section 4, provided by the authors of [13]. The implementation is available in both a scalar form (no SIMD instructions) and in a vector form explicitly coded using x86 SSE intrinsics. The measurements were performed on a 3.4 GHz Core i7 2600K (Sandy Bridge) processor. The vector (SIMD) implementation relies on extract and insert instructions to move data between the vector and scalar (GPR) registers. We also experimented with a possible (fake) memory gather instruction that would do the same work performed by four sets of extract, load and insert instructions. We measured what would happen if a 4-way (four 32-bit elements) gather instruction were to have a certain execution cost. First, we modeled the gather as having the cost of three sets of extract, load and insert instructions. Then, we modeled the cost as two of those sets. Finally, we modeled the gather instruction as having the cost of just one vector load. We did not try to explicitly manage the caches.

The results are summarized in Figure 3. Each bar shows the performance (bandwidth) for each of the configurations analyzed. The text inside the bars show the speedup relative to the scalar version. We observe that the SIMD version already achieves better than the 2 Gbit/s we expected and an almost perfect speedup. A very fast gather instruction (with a cost equivalent to a single vector load) further improves that performance by 30%, to almost 5 Gbit/s. Other bars show intermediate configurations, in which the gather instruction has the same cost as 3 or 2 extract/load/insert sets of instructions.

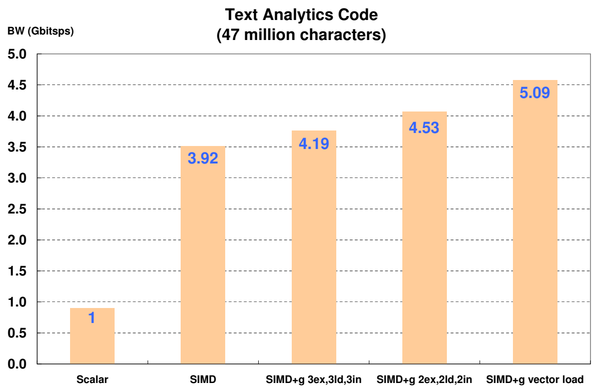

Figure 3. Performance of regular expression processing on a Core i7 (Sandy Bridge) processor. We see a big benefit from SIMD processing and further improvement from a possible gather instruction.

We summarize the impact of inline-SIMD acceleration for the Acquisition Annotator of Section 2 in Figure 4. The SIMD acceleration greatly reduces the time spent in the FSM component, making it negligible compared to the other components. The end-to-end speedup is close to two for the annotator.

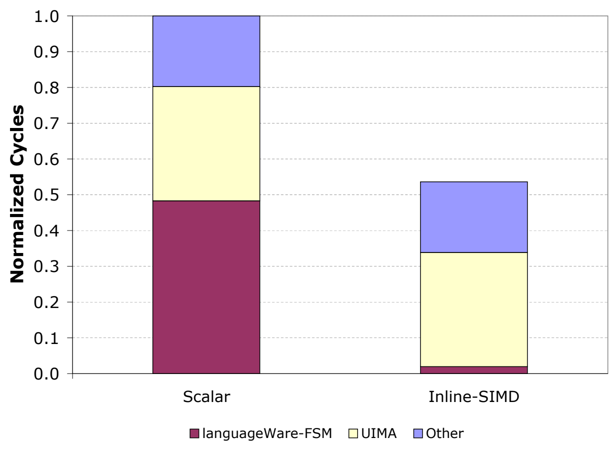

Figure 4. Cycle breakdown for the Acquisitions Annotator, before and after the use of inline-SIMD acceleration. We note that the acceleration makes the FSM component negligible and produces an end-to-end speedup of almost two.

7 Related work

There is extensive literature on efficient approaches to pattern matching, both against exact dictionaries and against regular expressions. Examples include Bloom filters [23], the Aho-Corasick [24] algorithm, specialized state machines [25], and content-addressable memories (CAMs).

Specialized hardware like FPGAs have been proposed to match exact patterns with Bloom filters [26, 27, 28], CAMs [29, 30], or Aho-Corasick FSMs [31, 32, 33], and to match regexs with direct hardware synthesis [32, 34] or FSMs [35]. For example, the parallel regex matching solution proposed by Lee et al. [34] delivers a throughput of 550 Mbyte/s but supports only 25 rules, while Suresh et al. [28] match exact patterns with a Bloom filter-based solution capable of delivering 2.25 Gbyte/s with 80-144 patterns. Commercial ASIC-based solutions [14, 15] claim higher throughput and, in one case [14], a rule set limited only by the amount of available main memory.

Solutions based on commodity hardware support larger dictionaries and rule sets, but their performance can be low [36]. Even recent works [37, 38] on lexical scanner generation for general-purpose processors do not attempt to exploit parallelism, and often generate code (*e.g.*, go-to tables [38]) that resists efficient parallel translation. Recently, Cameron et al. [39, 20] proposed a bit-parallel approach to exploit SIMD, and demonstrated 25× speedup on UTF transcoding workloads.

On the Cell, FSM-based solutions for exact matching include: a register-file-based solution by Iorio and van Lunteren [17] achieving 6.25 Gbyte/s per chip, a core-local SIMD-based Aho-Corasick implementation by Scarpazza et al. [16] capable of 5 Gbyte/s, and a main-memory-based one [18] capable of 275 Mbyte/s per chip (same authors). With Aho-Corasick on large dictionaries, Villa et al. [40] delivered 3.5 Gbyte/s on a 128-CPU Cray XMT, and Petrini et al.[41] delivered 0.5–2.2 Gbyte/s per core on an Intel Harpertown.

8 Conclusions

The significant growth of unstructured data and the rising importance of business intelligence applications urge computer architects to rethink architectural support for these emerging workloads. We focus on finite-state machine-based regular expression matching, which is often at the core of these workloads. We find that inline-SIMD processing is efficient for accelerating regex processing. Moreover, it can be further enhanced through more efficient gather/scatter operations, faster data movement between vector and general-purpose registers, and proper cache management. We find that inline-SIMD is sufficient to achieve end-to-end speedup that is within 8% of the ideal speedup of $2\times$.

We develop a first-order model to validate the choice of inline-SIMD as the regex accelerator. The first-order model provides the following three key insights regarding acceleration: (1) overhead of acceleration must be minimized to achieve a significant speedup, especially when the acceleration is performed on small data sizes, (2) when possible, the data size for acceleration should be as large as possible, and (3) if there is freedom to choose the data size and to choose the accelerator overhead, the ratio of data size and the accelerator overhead must be much greater than baseline processing rate.

To validate the choice of inline-SIMD, we also evaluate memory bus-attached, I/O bus-attached, and network-attached accelerators. We find that smaller overhead of the inline-SIMD relative to the other alternatives makes it the optimal choice for speeding up business analytics applications.

Our findings suggest that the benefits from inline instruction-set improvements in vector processing are sufficient to significantly boost the performance of business analytics applications. Inline SIMD processing provides a much greater speedup, around $1.8\times$ over the baseline, than the alternative accelerators for smaller data size (as in, for example, Twitter text messages). For large data size, for example e-mails and webpages, the inline-SIMD acceleration provides about the same speedup, $1.92\times$ over the baseline, as the competing accelerators. Therefore, we conclude that SIMD processing is the preferred form of acceleration for these applications.

References

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[2] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 system microarchitecture. *IBM Journal of Research and Development*, 49(4/5):505–521, 2005.

[3] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden. IBM POWER6 microarchitecture. *IBM Journal of Research and Development*, 51(6):639–662, 2007.

[4] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell Multiprocessor. *IBM Journal of Research and Development*, pages 589–604, July/September 2005.

[5] M. Nicola and J. John. XML parsing: A threat to database performance. In *CIKM*. ACM, 2003.

[6] E. Perkins, M. Kostoulas, A. Heifets, M. Matsa, and N. Mendelsohn. Performance analysis of XML APIs. In *XML 2005 Conference and Exposition*, Nov. 2005.

[7] Apache UIMA. http://incubator.apache.org/uima.

[8] LanguageWare Workbench. http://www.alphaworks.ibm.com/tech/lrw.

[9] http://twitter-friends.com.

[10] B. Klimt and Y. Yang. Introducing the enron corpus. In *First Conference on Email and Anti-Spam (CEAS)*, 2004.

[11] Kiwix, version 0.5, www.kiwix.org, Mar. 2007.

[12] C. Caşcaval, S. Chatterjee, H. Franke, K. J. Gildea, and P. Pattnaik. A taxonomy of accelerator architectures and their programming models. *IBM J. Res. Dev.*, 54:473–482, September 2010.

[13] D. P. Scarpazza and G. F. Russell. High-performance regular expression scanning on the Cell/B.E. processor. In *23rd Intl. Conference on Supercomputing (ICS'09)*, Yorktown Heights, New York, USA, June 2009.

[14] Cavium Networks. Cavium networks debuts NITROX DPI family of layer 7 content processors with market leading performance (press release), July 2000.

[15] RMI – Raza Microelectronics, Inc. XLR700 processor series, next generation multiprocessing, product brief, July 2008.

[16] D. P. Scarpazza, O. Villa, and F. Petrini. Peak-performance DFA-based string matching on the Cell processor. In *Third Intl. Workshop on System Management Techniques, Processes, and Services (SMTPS), held in conjunction with IPDPS*, Mar. 2007.

[17] F. Iorio and J. V. Lunteren. Fast pattern matching on the Cell Broadband Engine. In *2008 Workshop on Cell Systems and Applications (WCSA), affiliated with the 2008 Intl. Symposium on Computer Architecture (ISCA'08)*, Beijing, China, June 2008.

[18] D. P. Scarpazza, O. Villa, and F. Petrini. High-Speed String Searching against Large Dictionaries on the Cell/B.E. Processor. In *22nd IEEE Intl. Parallel & Distributed Processing Symposium (IPDPS'08)*, Miami, Florida, Apr. 2008.

[19] Z. Lei. XML parsing accelerator with Intel Streaming SIMD Extensions 4, white paper. Intel Website, Apr. 2008.

[20] R. D. Cameron and D. Lin. Architectural support for swar text processing with parallel bit streams: the inductive doubling principle. In *14th Intl. Conference on Architectural support for programming languages and operating systems (ASPLOS'09)*, pages 337–348, New York, NY, USA, 2009. ACM.

[21] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic Processing in Cell's Multicore Architecture. *IEEE Micro*, 26(2):10–24, 2006.

[22] M. Abrash. A first look at the Larrabee new instructions (LRBni). *The Dr. Dobb's Journal*, April 2009.

[23] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[24] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[25] J. van Lunteren. High-performance pattern-matching for intrusion detection. In *25th IEEE Intl. Conference on Computer Communications (INFOCOM 2006)*, pages 1–13, Apr. 2006.

[26] S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull, and J. W. Lockwood. Deep packet inspection using parallel bloom filters. *IEEE Micro*, 24(1):52–61, 2004.

[27] J. W. Lockwood, N. Naufel, J. S. Turner, and D. E. Taylor. Reprogrammable network packet processing on the field programmable port extender (FPX). In *ACM Intl. Symposium on Field Programmable Gate Arrays (FPGA 2001)*, pages 87–93, 2001.

[28] D. C. Suresh, Z. Guo, B. Buyukkurt, and W. A. Najjar. Automatic compilation framework for bloom filter based intrusion detection. In *Second Intl. Workshop on Reconfigurable Computing: Architectures and Applications (ARC'06)*, pages 413–418, 2006.

[29] L. Bu and J. A. Chandy. A CAM-based keyword match processor architecture. *Microelectronics Journal*, 37(8):828–836, 2006.

[30] I. Sourdis and D. Pnevmatikatos. Pre-decoded CAMs for efficient and high-speed NIDS pattern matching, 2004.

[31] C. Chang and R. Paige. From regular expressions to DFAs using compressed NFAs. In *CPM '92, A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, editors, Lecture Notes in Computer Science, No. 644*, pages 88–108. Springer-Verlag, 1992.

[32] B. L. Hutchings, R. Franklin, and D. Carver. Assisting network intrusion detection with reconfigurable hardware. In *10th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'02)*, page 111, Washington, DC, USA, 2002. IEEE Computer Society.

[33] I. Sourdis and D. Pnevmatikatos. Fast, large-scale string match for a 10 Gbps FPGA-based network intrusion. In *13th Conference on Field Programmable Logic and Applications (FPL'03).*, September 2003.

[34] J. Lee, S. H. Hwang, N. Park, S.-W. Lee, S. Jun, and Y. S. Kim. A high performance NIDS using FPGA-based regular expression matching. In *2007 ACM Symposium on Applied Computing (SAC '07)*, pages 1187–1191. ACM, 2007.

[35] J. Moscola, J. Lockwood, R. Loui, and M. Pachos. Implementation of a Content-Scanning Module for an Internet Firewall. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 31–38, Napa, CA, USA, Apr. 2003.

[36] S. Antonatos, K. Anagnostakis, M. Polychronakis, and E. Markatos. Performance analysis of content matching intrusion detection systems. In *4th IEEE/IPSJ Symposium on Applications and the Internet (SAINT 2004).*, 2004.

[37] P. Bumbulis and D. D. Cowan. RE2C: A more versatile scanner generator. *ACM Letters on Programming Languages and Systems*, 2(1-4):70–84, March–December 1993.

[38] A. D. Thurston. Parsing computer languages with an automaton compiled from a single regular expression. In O. H. Ibarra and H.-C. Yen, editors, *CIAA*, volume 4094 of *Lecture Notes in Computer Science*, pages 285–286. Springer, 2006.

[39] R. D. Cameron. A case study in SIMD text processing with parallel bit streams - UTF-8 to UTF-16 transcoding. In *2008 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*, pages 91–98, Salt Lake City, Utah, Feb. 2008.

[40] O. Villa, D. Chavarria, and K. Maschhoff. Input-independent, scalable and fast string matching on the Cray XMT. In *23nd IEEE Intl. Parallel & Distributed Processing Symposium (IPDPS'09)*, 2009.

[41] F. Petrini, V. Agarwal, and D. Pasetto. SCAMPI: A scalable Cam-based algorithm for multiple pattern inspection. In *2009 ACM IEEE Intl. Conference on High-Performance Computing, Networking, Storage and Analysis (SC'09)*, Nov. 2009.