ABSTRACT

The concept of default and its associated painful repercussions have been a particular area of focus for financial institutions, especially after the 2007/2008 global financial crisis. Counterparty credit risk (CCR), i.e. risk associated with a counterparty default prior to the expiration of a contract, has gained tremendous amount of attention which resulted in new CCR measures and regulations being introduced. In particular users would like to measure the potential impact of each real time trade or potential real time trade against exposure limits for the counterparty using Monte Carlo simulations of the trade value, and also calculate the Credit Value Adjustment (i.e, how much it will cost to cover the risk of default with this particular counterparty if/when the trade is made). These rapid limit checks and CVA calculations demand more compute power from the hardware. Furthermore, with the emergence of electronic trading, the extreme low latency and high throughput real time compute requirement push both the software and hardware capabilities to the limit. Our work focuses on optimizing the computation of risk measures and trade processing in the existing Mark-to-future Aggregation (MAG) engine in the IBM Algorithmics product offering. We propose a new software approach to speed up the end-to-end trade processing based on a pre-compiled approach. The net result is an impressive speed up of 3-5x over the existing MAG engine using a real client workload, for processing trades which perform limit check and CVA reporting on exposures while taking full collateral modelling into account.

1. INTRODUCTION

Counterparty Credit Risk (CCR) is a metric used by financial institutions to evaluate the likelihood of the counterparty of a financial contract (referred to as counterparty for short) to default prior to the expiration of the contract. It is critical for a financial institution to predict the CCR of a counterparty when making a trading decision and when pricing the value of a trade. Traditionally, trades are made by human beings, and response time for CCR typically falls into the range of hundredth of milliseconds. The emergence of electronic and e-commerce trading, however, demands a much faster response time and higher throughput over the current generation of CCR software which are designed mainly for human traders. Furthermore, it is also highly desirable to improve the precision of risk computation. A CCR is more precise if its computation takes into consideration more number of market scenarios and/or involves more timesteps. All of these requirements demand highly efficient software implementations and effective utilization of hardware resources.

The Mark-To-Future Aggregation Engine (MAG) is a key component of the risk computation software from Algorithmics that performs statistical measurements of the CCR computation. The current generation of MAG was designed for human traders and sustains a throughput of 3-5 trades...
per second with a latency of up to 300 ms per trade. In this paper, we describe our approach to improve the end-to-end throughput and latency of the MAG engine. The targeted risk precision is defined in terms of 5000 market scenarios by 250 timesteps.

There have been many recent work on performance optimizations for financial codes, many exploit accelerator technologies such as GPGPUs, CELL BE, and FPGAs [2, 1, 8], while others focus on algorithm level improvement such as [6]. One notable work in this area is from Maxeler Technologies on using FPGA to accelerate credit derivatives computation for JPMC [11]. This work focuses primarily on the pricing aspect of a trade as pricing algorithms are highly parallel. A similar effort of employing FPGA to speedup the pricing engine was taken by Algorithmics in the past [4].

In this work, we focus on another critical component of a financial software, the aggregation engine. In contrast to prior published work in risk analysis, our work targets a real production code. We found that optimizing a complex piece of production software requires one to take a holistic approach and to tackle performance bottlenecks at all levels, such as algorithm and data structure design, redundant computation elimination, memory subsystem optimization, and exploiting parallelism. In this paper, we demonstrate the steps taken to identify performance bottlenecks in the MAG engine and the techniques to address some of the overhead. We are able to demonstrate a speed up of 3-5x over the existing MAG engine for the limit checking and CVA reporting on exposures scenario using a real client workload on an off-the-shelf multicore server. We believe this work is a good starting point to closing the gap between the performance of existing MAG engine and the ultimate latency and throughput requirement for an online trading system.

The paper is organized as follows. Section 2 gives an overview of the current MAG engine implementation. Section 3 describes our approach to optimizing performance and Section 4 explains our optimizations for three important kernels. Section 5 presents our approach to doing platform-specific optimizations for the MAG engine using adaptive optimization. Performance results are discussed in Section 6. And we conclude and outline future work in Section 7.

2. THE MAG ENGINE

The Mark-to-future Aggregation (MAG) engine is where statistical measurements for CCR such as Credit Value Adjustment (CVA) and collateral modeling are computed. This section gives an overview of the data structure and computation involved in the MAG engine and the performance characteristics of its current implementation.

2.1 Data Structure

The MAG engine operates on two types of graphs. The top-level graph is called a hierarchy graph where a node represents a type of financial contracts and an edge indicates the existence of some relationship between these contracts. There is one hierarchy graph for each counterparty.

From the hierarchy graph, along with information on statistical measures that a client specifies to compute, a directed graph called the computation graph is derived. There is one computation graph for each counterparty. The computation graph is mostly a tree, in which case it is very much like an expression tree. In a computation graph, nodes represent computation and edges represent data dependence between computation. A node consists of a computation kernel and its internal data called states. States are typically vectors or dense matrices called sheets. A sheet is a two-dimension data structure organized by scenarios and time points. In the current implementation, sheets are in memory sequentially along the scenario dimension. There are two types of nodes in a computation graph, consolidation nodes and transformation nodes. Both types of nodes produce a new result, while only consolidation nodes may modify its own states. When applying computation from a consolidation node, states are often first read and then modified, such as element-wise summation of an incoming sheet into the sheet associated with the consolidation node. A transformation node, on the other hand, does not modify any states.

To give a sense of the scale of the data structure, a typical production run of the MAG engine today may monitor 10,000+ counterparties (i.e., 10,000+ computation graphs). On average, each computation graph contains 10 nodes. And states associated with a computation graph node can be several mega-bytes.

2.2 Trade Risk Scoring Computation

A trade consists of two pieces of information, a trade value sheet and trade parameters. The trade value sheet usually comes from the pricing engine, with simulated floating point values over a set of market scenarios and timesteps. Trade parameters include which counterparty is trading and other information such as the maturity date of the trade. The counterparty information of trade parameters determines which computation graph to be used for trade evaluation.

When evaluating a trade on a computation graph, it typically refers to the process of absorbing the trade value sheet into some consolidation nodes of the graph and/or computing statistical measures on computation graph nodes. A trade can be either read-only or commit. Read-only trades (e.g., what-if or lookup trades) do not modify any state of the computation graph, whereas commit trades do. When evaluating a trade, computation kernels associated with the computation graph are executed in a post-fix order similar to evaluating an expression tree. A computation kernel on a consolidation node takes as input its own state, as well as the output, or states, of its children. A particular leaf node, as selected by the trade parameter, takes as input its own state and the trade sheet. This process of propagating the trade value from the leaf node up is termed contribution absorption process.

Before processing any real time trade, the MAG engine constructs all hierarchy graphs and computation graphs based on previous trades. It also primes modifiable states of a computation graph by evaluating existing trades over the computation graph. The priming process is typically done in an overnight batch run. Once all computation graphs are primed, MAG is said to be in a baseline state and ready to handle real time trade evaluation.

Most CCR computations involve one counterparty, thus evaluation of distinct counterparties are largely independent. This gives the opportunity to do inter-trade parallel processing. However, when multiple commit trades are against the same counterparty, potential data collision may prevent parallelization at the trade level. The current production implementation can use a subset of cores for intra-trade parallel processing and may process several trades concurrently.
2.3 Performance of Current System

The current MAG engine is implemented in C++ and was designed according to software engineering principles. The code is highly object oriented and makes extensive use of templates. As common in object-oriented codes, functions are typically small in size and data structures present themselves as classes and are accessed via small access functions. The actual computations are spread among a large amount of source code. While this modular and abstract software design is favorable from a software engineering point of view, for performance engineering it is often difficult to collect the necessary informations and understand what really happens.

We employed hardware performance monitoring measurements on an Intel SandyBridge EP node using the instrumentation facility of the likwid tool [10, 9] to get a microscopic view of what actually happens when a single real time trade is being processed. The measurement revealed a good average Cycles Per Instructions (CPI), indicating that execution units are busy. The problem, however, is that many executed instructions are not relevant to the real computation in the algorithm. Statistics on the instruction mix showed a poor percentage of floating point operations and low data transfer rates. This is a common symptom for codes that suffer from abstraction induced overhead.

3. OUR APPROACH

3.1 Opcode-based MAG Engine

We design the new MAG engine by defining a framework that breaks down a typical risk scoring computation into basic units of computation kernels that are optimized individually. We refer to such unit of computation as opcodes. Each opcode is implemented as a computation kernel with a clearly defined set of input and output. As a basic computing block of a trace evaluation, each computation kernel is individually tuned. Opcodes are composed into a sequence to express the computation involved in a trade evaluation. We refer to the implementation of an opcode sequence, which is naturally composed of kernel calls, as a pipeline kernel.

Under this framework, each node of a computation graph is an opcode and the computation involved in evaluating a trade can be naturally expressed by a pipeline kernel consisting of computation kernels involved in a post-order traversal of the computation graph.

Figure 1 illustrates the software architecture of the new MAG engine. As it is too time consuming to generate a pipeline kernel for an incoming trade on the fly, the new MAG engine generates pipeline kernels for typical trade processing ahead of time. In the current MAG system, there exists a maintenance time window in which the system is brought down to perform batch processing, such as reconstructing the computation graph and re-priming the system with data according to latest trading information. While a batch processing is underway, a backup MAG engine with an older baseline state would continue to handle any incoming real time trade. We simply piggy-back on this mechanism to generate all pipeline kernels and compile them into binaries. The pipeline kernel library, tuned for specific architectures, is then dynamically linked in. Finally, the aggregation engine loads the symbols from the library and real time trading is ready to begin. During a real time trading, the Trade Parser parses the incoming trade and constructs the symbol name of the corresponding pipeline kernel based on the counterparty and node information. Then the Look-Up Engine loads the appropriate precompiled pipeline kernel and executes it.

As an effective technique to reduce redundant computation, the new MAG engine is also optimized to move computations that do not depend on any real time trade information to the batch processing. Results of pre-computation can be stored inside the MAG Memory and used later by real time trade processing. The trade-off between computation and memory consumption needs to be balanced to make the optimization effective.

3.2 Characterization of Kernels

There are elementary kernels and composite kernels. Elementary kernels typically represent one type of computation and can be classified into three classes:

- **Summation-based** kernels that reduce a vector into a scalar value.
- **Copy- or scale-based** kernels that read one or more vectors and produces a new vector.
- **Sorting-based** kernels that performs sorting.

Both summation- and copy/scale-based kernels can be well optimized by the compiler and are mainly memory bandwidth limited. Sorting-based kernels are more expensive than the previous two and highly input sensitive. They are typically control-flow and instruction throughput limited. Composite kernels, on the other hand, involve different types of computation and often invoke other elementary kernels. For instance, *collatoral* and *CVA* are both composite kernels. Kernels may also differ in their access patterns to sheet data structures. Most kernels iterate over the scenario dimension (also innermost dimension) of a sheet, while others traverse sheets along the time point dimension, resulting in poor spatial locality due to large strided accesses.

3.3 Optimization Approaches

There are three basic approaches to improving the latency and throughput of a software system: reducing instruction path lengths (i.e., reduce the amount of computation), improving CPI (i.e., improve core efficiency), and exploiting
parallelism. In this work, our target hardware is off-the-shelf multi-core server systems. Exploitation of accelerator technologies is left as a future work.

3.3.1 Reducing Computation

When implementing computational kernels, we focus on choosing efficient algorithms and data structures to minimize the amount of computation for each kernel. An important source of inefficiency in existing MAG engine implementation comes from redundant computation, such as loop invariant computation that can be hoisted out of inner loop nests. Another example to reduce computation is to pre-compute the required sequence of Gaussian random numbers, which are very expensive to generate due to its use of log/sqrt/div operations, and store them in memory. During real time trade processing, the pre-computed random numbers are simply retrieved from memory. This optimization effectively trades-off memory for computation.

We also employ value specialization to simplify control-flow in composite kernels, such as CVA and collateral, that use nested if statement on variables with one or only a few possible values. We use C++ template to specialize the few possible values for these variables. Each specialized code will enable the folding of some conditional statement, which in turn may enable further dead code elimination.

3.3.2 Exploiting Parallelism

We exploit both SIMD and thread-level parallelism. SIMD parallelism is exploited mostly on elementary kernels, such as summation- and scale-based kernels, which contains few control-flow and mainly contiguous memory accesses in their computation loops. While some elementary kernels can be automatically vectorized by the compiler, others have to be vectorized manually using SIMD intrinsics or assembly codes. For thread-level parallelism, we rely mostly on OpenMP worksharing constructs.

Synchronization is required after every kernel execution and the overhead can be a limiting factor for scalability. We are careful at parallelizing mainly outer loops with sufficient computation to minimize the OpenMP runtime and synchronization overhead. Most kernels contain outer loops over time points. The parallelism over processing 250 time points is typically sufficient to achieve a good load balancing for the number of hardware threads available on a typical server. Since input data (i.e., input sheets) sizes are primarily fixed by the number of scenarios (5000 scenarios) and time points (250 points) modelled by the aggregation engine, we face a severe strong scaling situation if targeting many more processor cores.

3.3.3 Optimizing for Memory Hierarchy

The main task consideration is to traverse data in a hardware prefetcher friendly fashion, expose more cache locality, and employ first touch placement to benefit from the parallel memory interfaces available on current multi-socket servers. For all kernels, we make sure that data is transferred in a stride one fashion. If a kernel accesses some sheet data in a strided fashion, spatial blocking along the scenario dimension is used to still make good use of prefetching capabilities and exploit spatial locality.

3.3.4 Algorithm Improvements

For all kernels, the algorithms used are reviewed to choose the more efficient one. Our high performance library allows one to choose between accuracy and speed for certain kernels as, e.g. the summation kernel (see Section 4.1). Other targets for algorithmic improvements are the select operation and standard deviation calculations.

4. KERNEL OPTIMIZATION

In this section, we select three key kernels to demonstrate the various optimizations that are applied.

4.1 Summation Kernel

Finite-precision floating-point summation is a very common underlying operation for many statistical computations used in financial risk analysis. Therefore, it is critically important to have a summation algorithm that is both fast in speed and produces an accurate result. For example, inside MAG, Kahan summation [5] is used in the Credit Value Adjustment (CVA) computation to boost the summation accuracy, while compromising on speed. This is because the Kahan summation algorithm achieves good accuracy by introducing additional arithmetic operations. As the demand for real-time CVA and aggregation rises drastically, speed or performance becomes an aspect that cannot be readily compromised.

We developed a shift-reduce summation algorithm which uses a cascading technique to sum a sequence of floating-point numbers. It has the same error bound as the well known pairwise summation algorithm [3]. That is an error bound proportional to \( \log_2(N) \) for summing a sequence of \( N \) numbers.

The advantages of our shift-reduce algorithm over the pairwise summation are summarized below:

- It is non-recursive (as inspired by the concept of a shift-reduce parser) and hence avoids the overhead associated with recursion, such as that of acquiring and deleting the stack in each recursive call.
- The additional book keeping storage required by shift-reduce is proven to be a maximum storage of \( 64 \times \text{sizeof(float)} \). This temporary storage can be contained in the L1 cache and does not incur much load latency which means small overhead.
- It can be readily SIMD optimized. The SIMD optimized shift-reduce implementation has identical error bound as the non-SIMD optimized shift-reduce implementation.
- The shift-reduce algorithm enables a more effective SIMD optimization than the pairwise summation algorithm. This is because the prologue and epilogue handling for data misalignment are done only once, outside of the main loop body.

In our kernel library the user can choose to either sacrifice some accuracy for speed and use the faster SIMD-vectorized shift-reduce kernel or use an optimized Kahan-Babuska [7] algorithm for highest accuracy.

4.2 Tail Kernel

The tail kernel, or quantile computation, is often used in the context of computing Value-at-Risk (VaR). VaR is a widely used risk measure of the risk of a devastating loss on
a specific portfolio of financial assets under some infrequent circumstance. Given a probability percentile and a particular timestep, the tail kernel, in the context of the aggregation engine, identifies which market scenario(s) would exceed, or fall below, the threshold. A commonly used statistical routine to implement the tail kernel is the \texttt{std::nth_element} function. This routine performs a quick select operation without performing the complete quick sort. The tail kernel is parallelized along the timestep dimension, this means that doing the select operation on a vector of length 5000 is the atomic operation. Because select is not bandwidth limited and takes significant time this kernel scales very well. Due to imperfect pipelining this kernel also benefits from using SMT threads.

The mean kernel, which does summation computation, along with the tail kernel are the two key kernels used by CVA. That is, CVA may utilize either the mean kernel, or tail kernel, as controlled by a parameter. Therefore, optimizing the two kernels well via SIMD vectorization and SMP parallelization as we have shown naturally lead to drastic performance improvement on CVA.

### 4.3 Collateral Kernel

The collateral kernel is used to model the collateral posted to comply with a CSA (credit support annex from an International Swaps and Derivatives Association master agreement) between two counterparties. The input is the aggregated value of a set of deals that fall under the CSA. Collateral is by far the most costly kernel and a difficult optimization target because of its many configuration parameters.

The collateral algorithm requires many input parameters, such as the lag periods, which are time windows between when collaterals are posted to when they are actually received. The algorithm adjusts each input value, i.e., \texttt{sheet}(t_j, s_i) at a particular time \( j \) and scenario \( i \), with collaterals that have been posted prior to \( t_j \) and are received by \( t_j \). This requires taking lag periods into account. It is possible that a collateral is posted not on simulation time point (i.e. not on a defined \( t_j \)) and its effect needs to be interpolated to a nearest time point. Brownian bridge interpolation is therefore used. The interpolation process requires normally distributed numbers which are typically expensive to generate as expensive operations such as log, divide and square root are needed. Other algorithms for generating normally distributed numbers may require rational polynomials. The bridge also requires expensive operations in computing the required coefficients. Three specific optimizations are applied to the collateral kernel. These are pre-computation or caching of results generated by intermediate operations, SMP parallelization, and input data transpose. Input data transpose is beneficial if a kernel accesses the data with large offsets orthogonal to the data layout in memory. The overhead introduced by the transpose operations is compensated by the more efficient data access in the kernel. An alternative to this is spatial blocking, which also improves the data access pattern for the cost of slightly bigger memory consumption for the necessary caching of intermediate results.

A severe performance limitation in collateral is the involving control flow. This branch overhead could be prevent as most is invariant against the sheet data. This static control flow was partly addressed by using template parameters which gives the compiler the opportunity to remove static control flow. Still not all cases can be resolved by templates. It can be shown that by removing all static control flow and applying spatial blocking a runtime improvement of up to a factor of two can be reached. Still much work is still needed to employ these actions in general for the production collateral kernel.

### 5. ADAPTIVE OPTIMIZATION

Different processors often require architecture specific optimizations for good performance. Because this might involve using binary incompatible SIMD instruction set extensions, a mechanism is required to support different processor flavors and still provide architecture specific optimizations.

The following requirements must be met:

- One single kernel library supporting all necessary processor types
- Ability to detect the underlying architecture and pick an optimized kernel at runtime
- No restrictions against supported processor microarchitectures
- Ensure low or no overhead introduced via the mechanism

With the above requirements, we designed a general adaptive mechanism for runtime kernel selection. The implementation works as follows:

- On startup, a centralized init routine is called once, detecting on which architecture the code is running. On X86 this stage is based on querying the \texttt{cpuid} instruction, which provides detailed information about supported processor features.
- The init routine enables the use of an optimized kernel by setting a function pointer to the proper architecture specific version.
- A generic portable default version is used when no architecture specific version is available.

![Figure 2: Mechanism for providing optimized kernel variants.](image)
The kernel is then invoked via the function pointer during runtime of the application.

Different choices are available for providing specifically optimized kernels. The simplest option is to use an optimizing compiler. To meet the portability requirement, code auto-dispatching is used. In case the compiler does not support auto-dispatching, a variant build is used. The implementing mechanism is based on a combination of make and c preprocessor techniques and can be realized with a single source strategy. If necessary, an architecture specific kernel can also be implemented by hand using intrinsics or assembly language. Which kernel is used for a specific processor type must be explicitly set in the init routine. For every kernel, a portable fallback is provided. This gives the library developer the flexibility of either hooking up a hand or a compiler optimized kernel into the adaptive framework. The framework also allows the use of different compilers for different processors. The options for providing optimized kernel variants and the implementing mechanism for the variant build are illustrated in Figure 2.

The overhead associated with kernel invocation via a function pointer versus direct call was evaluated using a micro benchmark. The result varies slightly depending on the architecture but a 6-8 cycles overhead is measured on a SandyBridge EP system. This is insignificant in light of thousands of cycles a kernel takes.

The adaptive optimization mechanism implemented simplifies the build environment as well as the library usage and execution environment as there is only one library being built from one single source code base.

6. PERFORMANCE EVALUATION

In this section, we will evaluate the performance improvements of our techniques. We will present scaling results for each of the three classes of elementary kernels. Due to the importance of the collateral kernel, we cover it in more depth in Section 6.3. Finally we will present a complete pipeline kernel to demonstrate an end-to-end trade processing for some real client cases.

6.1 Experimental Setup

The performance numbers are benchmarked on a 2.2 GHz SandyBridge EP system running RHEL 6 Linux. The system consists of two 8-core sockets with each core capable of 2-way Simultaneous Multi-threading (SMT). Each core has a private 32KB L1 cache and a 256KB L2 cache. The L3 cache is 20MB and shared among all cores on a socket. All computation kernels are compiled with Intel’s V13 compilers.

The main input and output sheets are 250 rows by 5000 columns of either single- or double-precision floating point numbers. Some of the values are randomly initialized via `drand48()`. The input trade data is always assumed to be streamed in with no reuse. This is because in the real system, a trade sheet is delivered to aggregation engine via a TCP/IP port from the pricing engine. Reuse of data associated with a computation graph depends on whether the trade is hitting the same counterparty or not. A rule of thumb in the financial trading situation is that the majority of the trades are against a small number of counterparties. Since the cache behavior is highly input dependent, for benchmarking purposes, we consider the worst case scenario.

The generation of a normally distributed number requires some real client cases.

6.2 Elementary Kernel Performance

Table 1 shows typical kernel performance for each type of elementary kernels. All data is in terms of throughput in sheets/s using single precision floating point accuracy. The sheet size is 250x5000 resulting in a sheet size of 5MB.

<table>
<thead>
<tr>
<th>Class, kernel</th>
<th>Sequential per Socket</th>
<th>per Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summation</td>
<td>2722</td>
<td>8087</td>
</tr>
<tr>
<td>Scale ConvertCurrency</td>
<td>1469</td>
<td>7057</td>
</tr>
<tr>
<td>Sort LeftTail</td>
<td>106</td>
<td>1064</td>
</tr>
</tbody>
</table>

Table 1: Kernel throughput in sheets/second using single-precision floating point where each socket has 8 cores w/ 2-way SMT, and each node has 2 sockets.

where all sheets are streamed from the memory.

6.3 Collateral Kernel

The collateral kernel uses Mersenne Twister as its uniform random number generator which feeds into a inverse CDF transformation to produce normally distributed numbers. It then generates random bivariate normal variables for a specified variance covariance values by consuming two normal numbers. The bivariate variables are used in bivariate brownian Bridge interpolation. It is also possible that a univariate brownian bridge is used for interpolation and in such case, only one normal number is consumed.

The generation of a normally distributed number requires many expensive operations. For instance, a typical trade may consume several millions of normal numbers. Therefore, it is advantageous to pre-generate the numbers during the batch processing session and let the collateral kernel consume pre-generated numbers as needed during real-time trade processing. To measure the effect of precomputation, we evaluate a synthetic trade that is configured with 1532 call days for the collateral kernel and uses bivariate brownian bridge. This trade consumes a pool of 12.8 million normal numbers, which is approximately one pair of normal num-
Table 2: Performance of the collateral kernel with memoization and precomputation

<table>
<thead>
<tr>
<th>Optimization</th>
<th># of instr. throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>before optimization</td>
<td>4.55e+11  1.97</td>
</tr>
<tr>
<td>with precomputation</td>
<td>2.98e+11  1.83</td>
</tr>
<tr>
<td>with precomputation and memoization</td>
<td>1.92e+11  2.72</td>
</tr>
</tbody>
</table>

The computation graphs resulted from an actual client production run can result in more than 50,000 pipeline kernels. A pipeline kernel can sometimes describe multiple linearized paths from different leave nodes up the computation graph. The client is also given the freedom to describe which of the pipeline kernels should be high volume or low volume by marking the counterparties to be high or low volume accordingly. A pipeline kernel which comes from walking the computation graph belonging to a counterparty with high volume tradings is also marked as high volume. The computation graph nodes associated with the high volume counterparties are always stored in memory, whereas the low volume counterparties data are stored on disk and are memory mapped in and out on demand. The pipeline kernel which we have selected is a high volume kernel. It consists of calling a variety of statistical kernels, such as CVA, collateral, and tail. It also consists of simpler kernels that do currency conversion (i.e., element wise multiply and update) and contribution absorption (i.e., element wise addition of two sheets). As the memory updates to the consolidation nodes need to be all or none, temporary storage for each node is created. The actual memory update takes place at the end, when execution of a entire path succeeds. For what-if trades, there would be no update to the consolidation nodes and the temporary storage is discarded at the end. Relevant statistical results are returned by the kernel. To get a profile of where time is spent, we execute the pipeline kernel using a single thread run. The percentage of overall time taken by each computation kernel is shown in Figure 4.

Figure 4 shows that 60.5% of time is spent in the collateral kernel. The RightTail and RightTailExposure kernels are both heavy time consumer. The difference between these two tail kernels is whether the negative values in result are clamped to zero or not. The total temporary storage consumed by this pipeline kernel is approximate 9 MB. (The storage any pipeline kernel takes is typically less than 35 MB.)

The update related kernels are the contribution absorbing kernels. That is, it takes an input sheet and performs element-wise addition to the result sheet. When currency conversion is required, element-wise multiplication with with
conversion factors is done prior to the addition. These kernels are fairly lightweight and yet they take total 8.5% of the time. This is because it performs branching inside the innermost loop. That is, it has condition checks to determine whether each element is to be clamped to zero, or negated first, prior to the addition. Loop unswitching optimization should help tremendously with the update kernels.

Time spent in Other is due to memset, memcpy related operations associated with the allocation of temporary storage. These operations are internally replaced with highly optimized variants provided by the Intel compiler.

The effect of the collateral kernel can easily exceed 60% of the end-to-end time if the configuration requires more interpolation points. The kernel which takes less than 1% of time is omitted from discussion.

In MAG’s production run, over 50,000 pipeline kernels get enumerated from 10,000+ computation graphs. Each pipeline kernel on average invoke approximate 50 computation kernels, although extremely large or small pipeline kernel that invoke more than 1000, or less than 10 kernels can also be found. We begin to tackle the problem of optimizing the entire pipeline kernel as a whole by first attempting to remove the unnecessary memcpy operations between sheets. The result of our initial attempt at pipeline kernel optimization, along with each computation kernel tuning, have resulted in an impressive 3-5x speedup in trade processing in the end-to-end MAG production environment.

The new aggregation engine delivers for 2000 randomly selected trades from 1948 actual customer trades a 3x end-to-end speedup by increasing the trade processing throughput from 6 trades to 18 trades/s. Another heavier client workload has shown a 5x improvement by increasing the throughput from 2 to 10 trades/s.

7. CONCLUSION AND FUTURE WORK

We presented a holistic performance engineering effort for the IBM Algorithmics Mark-to-future Aggregation (MAG) engine. We introduced an approach of linearizing the computation graph into pipeline kernels, thus effective remove the graph traversal overhead from the real time. Together with the kernel tuning effort we can present an impressive 3-5x speedup using standard multi-core servers measured in an end-to-end MAG production environment. With an adaptive single library solution, we provide a flexible and robust setting as a future proof in terms of usage of hardware accelerators.

The success is based on addressing all frontiers starting with software structure, algorithmic improvements, SMP thread parallelization and architecture specific optimizations such as SIMD vectorization, as well limited cross kernel optimizations.

To reach a higher throughput goal, we will explore new accelerator technologies such as the GPGPUs for speeding up the collateral kernel. More aggressive strategy in cross kernel optimizations will also be explored.

8. ACKNOWLEDGMENTS

The authors would like to acknowledge Jose Moreira for reviewing the paper and to acknowledge Ettore Tirotti, Bob Francis, Rajesh Bordawekar for exploring the next step for this work.

9. REFERENCES


