Performance and Power Impacts of Autotuning of Kalman Filters for Disparate Environments

Brian Gravelle[1] and Boyana Norris[1]

University of Oregon, Eugene OR 97401, USA,
gravelle@cs.uoregon.edu, norris@cs.uoregon.edu

Abstract. In both high-performance computing and Internet of Things (IoT) applications, it is important for programmers to utilize the hardware as efficiently as possible in terms of both performance and power usage. Automatic methods of tuning code for such efficiency have been developed and used for many years in HPC, but less so in other areas. In this paper, we apply these methods to the domain of IoT to explore the feasibility of using autotuning in this domain. Our case study examines the application of roadway traffic analysis. We use Computer Vision techniques to count the cyclists, pedestrians, and cars as they travel through a video of a campus intersection. A significant portion of the tracking computations is the Kalman filter method, which is used to estimate the paths of objects through the video. This Kalman filter is the target of this autotuning study. We present results for the performance and power usage of the system with multiple autotuning techniques and optimized BLAS libraries on three disparate architectures. Our results show that autotuning is both feasible on and beneficial to IoT platforms.

1 Introduction

Fifty years ago Moore predicted the rapid advancement in complexity and speed of computer processors. These improvements have allowed for the development of more and more complex computer systems that tackle larger and larger problems. These advancements have given a great deal to HPC users, but have come at the cost of more difficult programming. To achieve peak possible performance on a super computer, the developer must spend hours tuning the program to the exact specification of the system; the cache size, vector length, and so on all must be taken into account to squeeze the best possible performance out of the machine. As a result of these complexities, a great deal of effort is expended but often results in sub-optimal performance.

Several methods are available to assist programmers with these challenges. For example, vendor-tuned versions of linear algebra libraries such as BLAS [14, 15] and LAPACK [1] incorporate many optimizations but are unable to apply those that rely on a larger perspective of the algorithm such as loop fusing and certain methods of minimizing memory traffic. Additionally, there is a variety of optimizing compilers (Intel, Portland Group, and Pluto [8]), but studies[23] have shown that these often don't utilize all available optimizations. As a result of

these short-comings, other methods of producing optimized code are constantly being explored.

A more robust way to tune a program is to make use of an autotuning system that can automatically transform code to find optimal or near optimal performance. These transformations can occur at or immediately before compilation so that the source code can be written and distributed as simple, readable code and then optimized separately for each system in use. For this study we use the Orio autotuner [18] which optimizes the code based on empirical tuning. The tuning is done by specifying the portion of the code that is relevant and annotating it with tuning options. Orio takes this information and applies potential optimizations to the code, tests each version, and saves the fastest one for the user to reintegrate with the rest of the program.

As research has shifted focus from maximizing FLOPs to maximizing FLOPs per Watt, the process of tuning has remained largely the same but now has two optimization goals. Many tuners now incorporate dual optimization that targets power (or energy) usage and speed. Additionally computer hardware is becoming increasingly complex as systems move to incorporate heterogeneous cores such as GPGPUs, Intel MICs, and FPGAs. Which must be considered in addition to the CPU when tuning a program.

Recently, the Internet of Things (IoT) has grown at a rapid pace and is projected to continue become an integral part of our lives. There are many challenges that face IoT applications that are quite similar to those in HPC. Even simple IoT processors are beginning to incorporate multiple cores, vector units, and GPUs. These complications require more difficult tuning of code, particularly as applications in IoT become more sophisticated and demand more computational power. Likewise, IoT devices often operate under strict power limits to extend battery life or minimize cost. Given these similarities to HPC, the realm of IoT offers another area that autotuners could make a significant impact.

In this paper, we explore Kalman filters as a case study of autotuning on disparate architectures. The Kalman filter is a common tool used for the estimation of states in dynamic systems; in this case, for the estimation of paths for objects moving through a two-dimensional space. We use Orio to optimize the filter and underlying linear algebra computations and compare the results to optimized BLAS implementations to demonstrate the effectiveness of autotuning. These experiments are performed on Intel Xeon server, and IBM Power8 server and a cluster of low-power ARM devices. These architectures are chosen to compare the results of autotuning for typical HPC target systems with the results from autotuning for IoT type devices as a first attempt at autotuning for IoT.

The rest of this paper is organized as follows. Section 2 presents background information on autotuning and Kalman filters. Section 3 explains our experimental methodology and test environment. Section 4 presents and analyzes our results. Future work is described in Section 5. Section 6 concludes.

2 Background

In this section we present background information about the autotuning techniques studied in this paper and the case study used.

2.1 Autotuning

In High Performance Computing, each program must be tuned to achieve the best possible efficiency on the target system. This tuning can be an arduous and time consuming task for even the most experienced developers. Automatic methods used to ease this process are called Autotuning and fall into two categories: empirical and analytic.

Empirical autotuning generates and tests the various optimizations and saves the best result. This method is limited by the size of the search space and the techniques used to explore it which determine how long the process will take. Analytic autotuning creates a model of the program and a model of the system which are then analyzed to determine the best optimizations to apply to the program. The modeling approach is faster and does need to be run on the target system, but it can be limited by the ability to model the program and system.

On the empirical side, several systems are available to test numerous code transformations and determine the best variation. CHiLL [12] is a framework for applying loop transformations to a program. OpenTuner [2] is a framework for creating program specific autotuners. The Orio [18] system allows a programmer to annotate the program with potential optimizations including loop transformations, compiler flags, OMP pragmas and more. The authors of [13] explore multicore autotuning on stencil computations. In [19], Jordan et al. present an autotuner for parallel programs that produces multiple optimal versions that can be selected from at runtime depending on system parameters. For heterogenous systems, Chaimov et al. developed OrCL, [11] that tunes for GPUs, Intel MICs, or multicore processors. To make up for the time-consuming nature of empirical autotuning and increase portability, [34] takes an online approach to tuning. In a setup similar to JIT compilation, the systems switches versions as the iterations of the loop run eventually settling on one that is fastest.

The analytic method builds a model of the program and the hardware to predict the best possible program optimization. This approach is taken in compilers to improve results and also in some autotuning systems [37]. These models tend to be limited by a lack of knowledge concerning different inputs or other run-time characteristics. Other methods combine the model results with empirical testing to restrict the search area and avoid the limitations of the model. For example, the authors of [21] use a model of the program to limit the search space and then explore those options empirically. The authors in [5] utilize machine learning based on empirical runs to create a more accurate model of the program. The PLuTo system [9] combines the methods to optiize programs for both data locality and parallelism.

While the primary goal of these systems has been optimizing performance of a program, there are also efforts to optimize for power or energy [24, 3, 11].

The systems use single or multi-target optimization to tune for a combination of performance, energy, and power.

2.2 Code Generation

Alternatively to tuning the code domain specific languages exist that can create highly optimized programs based on knowledge of the domain. These systems can take advantage of a wider view of the problem and possible inputs to the system to identify optimizations that may not be possible in a more general system.

One such system is BTOBLAS [23]. This system takes matrix operations as input and transforms them into C code. By taking the full equation as an input it can combine loops, avoid replicating computations, and perm other operations that are not possible using regular BLAS libraries. For multithreaded applications the threads can wait to synchronize until the end of the computation, rather than being initiated at each BLAS call.

2.3 Object Tracking in Videos

The first step in any analysis of video data is identifying which parts are interesting (foreground) and which are not (background). For analysis of traffic, a common method of differentiating between the two is by using motion to identify areas taken up by moving vehicles. This process is most basically done by developing a background image and subtracting the most recent frame from that image. Pixels that are different from the background are shown in white those that are similar are in gray and those that are the same are shown in black. Thus large blobs of white can be interpreted as cars or other moving objects.

The most basic form of background subtraction is to subtract the current frame from the previous one, but this is overly simplistic and produces poor results. A better way is to develop a background model as the video progresses. In both[17] and [16] the authors develop the model by first subtracting from the background model, then averaging the background model with the current frame, ignoring the sections where movement occurred. An improved option calculates likely values for each pixel in the video and differences the new frame from those likely values (known as the Gaussian Mixture Model). Several versions of this were produced [32, 38] and OpenCV has a built-in version, based on [38], making it easy to use. This process has the advantage of being able to handle changing background (i.e. buildings being constructed, weather changes, day/night) and oscillating motion (i.e. trees in the wind). Unfortunately, it can lose track of objects that stop for extended periods, such as those at intersections.

Alternatively, objects such as cars, bicycles, and pedestrians can be identified based on specific features in a process called feature extraction. Features are identified using large datasets and machine learning and then applied to the video analysis to identify traffic. Feature extraction can be more computationally intensive than other methods, but produces good results and can work when cars

or people are stopped at traffic lights. In [26, 27, 29, 28, 30], Shirazi et al combine the background subtraction and feature extraction to get the best of both worlds.

Once the objects are identified it is necessary to match object from one frame to the next. One common method is to use the shape of the object as a template and test that against candidate in the next frame in what is called "normalized area of intersection" [17]. In [16], Fang et al. extend this technique to include corner detection method to handle occlusion. Color and other attributes are included in the calculation in [30].

A slightly different technique is to predict the next location of an object based on its speed and direction and then find the object that best fits that estimation. This method often incorporates Kalman filters to maintain the predictive model and adjust it as measurements and errors discovered. The simplest method is to apply one Kalman filter to each object being tracked and assume constant velocity of the objects as in [4]. Some implementations ([6, 31]) extend this method use one Kalman filter per corner of each tracked object. Stauffer and Grimson [33] include a size parameter along with the position, while Shirazi and Morris [30] resolve ambiguities with the color and size information much like the normalized area of intersection method. We focus on tuning a basic Kalman implementation in our case study and discuss it further below.

A more advanced method of tracking is to use the features of different objects to match them from frame to frame. This method, used in [26, 27, 29, 28, 30], identifies several features (corners or edges usually) for each object in one frame and the locates objects with similar features in the second. Like the previous discussion of feature extraction, it can require significantly more computation than the other methods. In [30] Shirazi and Morris combine position, size, and features for improved accuracy.

2.4 Kalman Filters

Kalman filters [20] were originally introduced in 1960 to provide a solution to filtering and prediction problems in dynamic systems. Initially, the filters targeted the signal processing involved in communication and control systems. Since then, the Kalman filter has been applied to many dynamic systems in fields including physics, signal processing, economics, and climate science. In this section, we present some background information to enable the reader to better understand the rest of the paper. Interested readers are directed to the following sources: [7, 35, 10, 20, 22].

The Kalman filter is used to estimate the state of some dynamic system based on measurements, models, and associated errors of that system. This information is combined into a set of matrices that represent the whole system (Table 1). The sizes of the matrices are based on the number of states in the system model (n) and the number of states measured in each time step (m). While some applications have larger state sizes, most require only a small number of states, resulting in matrices that are on the order of 10×10. In addition to the matrices, there are also two vectors; x of length n that holds the estimated state and y of length m for the current measurements.

Table 1: List of matrices involved in the Kalman filter.

Name	Description	dimensions
A	system dynamics	$n \times n$
C or H	measurement matrix	$m \times n$
Q	process (system) noise	$n \times n$
R	measurement noise	$m \times m$
P	error covariance	$n \times n$
K	Kalman gain	$n \times m$

At each time step, the predicted state and the most recent measurement are compared and these values, combined with the error matrices are used to estimate the state at that time. This estimation is based on weighting the measurement and model (expressed in the Kalman gain) and emphasizing one over the other based on error values. The accuracy of the Kalman filter relies heavily on the accuracy of the error and covariance matrices since these are integral in determining the Kalman gain with each step.

$$\hat{x}_{new} = A\hat{x} \tag{1}$$
$$P = APA^T + Q \tag{2}$$
$$K = PC^T(CPC^T + R)^{-1} \tag{3}$$
$$\hat{x} = \hat{x}_{new} + K(y - C\hat{x}_{new}) \tag{4}$$
$$P = (I - KC)P \tag{5}$$

The Kalman equations are shown in Equations 1 - 5. Equations 1 and 2 are the predict portion, estimating the next state based on past estimations and system model. Equations 3 - 5 and the correct section, updating the estimated state and Kalman gain based on the measurement and predicted state. There are additional versions, such as the Extended Kalman Filter and Square Root Kalman Filter that implement improvements such as time variant error functions, or other changes to improve performance or accuracy.

2.5 Kalman Filter Tracking

For the case of tracking objects, positions are sampled at certain time intervals by the tracking system and the Kalman filter is used to match the objects between measurements. These measurements can come from many types of sensors including radar, video, or GPS, which will determine the model used in the system dynamics matrix. Our example uses measurements of position in 2 dimensional Cartesian coordinates that represent positions in the video frame. Other systems can expand on this to include measurements of velocity, 3 dimensions, and object attributes (i.e. size, color).

For tracking, the Kalman prediction functions (see Section 2) are used to estimate the next position ($\hat{x}_{new}$), which is used as a basis for determining which measurement(s) in the next time step corresponds to the same object. Once

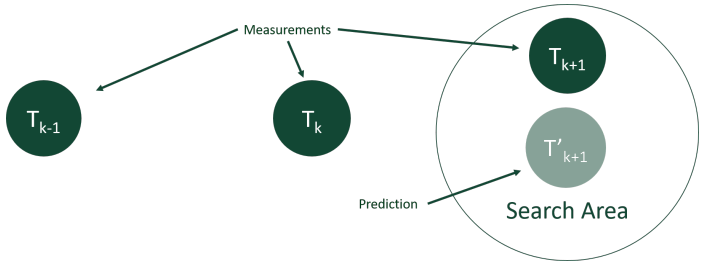


Fig. 1: Example of Kalman prediction for object tracking. T_{k-1} is the previous position T_k is the current, T'_{k+1} is the prediction, and T_{k+1} is the measurement

selected, these measurements (y) are then used in the update equations to adjust the model in preparation for the next prediction step. A simple diagram is shown in Figure 1.The measurements inherently include a certain amount of noise, and the model cannot replicate all aspects of the system, so the Kalman filter is used to combine all the available information into one estimate. By using the model and measurement, the filter can handle situations such as missing measurements in one or more time-steps or unexpected maneuvers.

Adjusting this process to work for many targets rather than a single target is a simple matter of sorting out which measurements go with which targets. The challenges mostly fall into the categories of targets becoming overlapped and separating. These situations are generally handled by removing targets after they have been missing for some time and having a "trial period" before a new object is fully accepted. The overlapping period can be handled by allowing several missed time steps before the object is removed or allowing multiple targets to "own" a measurement. Since each object operates independently, the system maintains a separate filter for each.

3 Methodology

In this study we compare several methods of optimizing a Kalman filter used in tracking objects through a video. In this section, we discuss the data, systems, and autotuning techniques used for the experiments.

3.1 Data

We used three types of data to test the system. First, we generated simple projectile motion data for 900 tracked objects. We used this data to test the tracking system separate from the overhead of actually processing the video.

For video results we captured a video of approximately 10 minutes of moderate traffic at a campus intersection. This video includes cyclists, pedestrians, and cars traveling in all directions. Each frame of the video contains about 5 to 15 objects that must be tracked.

To further stress the system, we generated videos with various numbers of objects moving through. These ten minute videos contain 25 to 500 objects in each frame so that we can explore how the number of required filters impacts the results.

3.2 Systems

For the experiments we use three different machines. Two of the machines used are servers similar to those found in modern HPC clusters. One is an Intel(R) Xeon(R) CPU E5-2699 v3 @ 2.30GHz with AVX instructions for vector processing. The other is an IBM Power8 @ 2.0GHz. Such machines are common targets for autotuning as they are regularly used in HPC clusters. The other machine is a Raspberry Pi with an ARMv7 CPU @0.6GHz. The ARM CPUs used here are similar to those common in embedded systems. Kalman filters are often used in embedded systems used for signal processing, computer vision, and other applications. We think that applying autotuning techniques in the embedded environment could bring performance and power benefits to that area.

On the ARM and IBM machines, we use GCC 4.9 as the C compiler. For the Intel machine, we used Intel's compiler (version 17.0) and compare the results with GCC (version 4.9) to give a broader view.

In all cases, we compared the results of the tuning to the untuned code and versions of the BLAS that are available for the system. For all machines, we used the ATLAS [36, 25] implementation of BLAS, which uses empirical autotuning methods to build an optimized library upon installation. For the Intel machine, we also included Intel's MKL CBLAS library, which is specifically optimized for the architecture.

3.3 Autotuning the Kalman Filter

Autotuning can be applied at different levels of the program. In this case, we autotuned at the level of individual linear algebra operations and at the level of full Kalman operations.

Tuning the individual linear algebra functions allows the autotuner to find optimizations local to each function and produce an optimized library that could be available for other applications as well. By separating out the tuning of each function, we also make the load on the programmer significantly easier since the operations are commonly reused and the functions small compared to tuning a full algorithm. Similarly, this method reduces the cost of tuning as fewer combinations are possible, so the search space is much smaller.

Tuning the linear algebra functions separately also presents some drawbacks. If the goal is to create a general purpose library then the results of the autotuner will depend significantly on the input. Orio is able to tune to a variety of different input types and sizes, but this process extends the time required for tuning and may not be optimal for all input sizes unless the tested sizes are extremely fine-grained. Furthermore, if the linear algebra functions are tuned and not the

overall algorithm, then some optimization options, such as loop fusion, cannot be used.

When tuning the Kalman filter as a whole, the first step is to write Kalman functions in which the matrix operations are inlined. The resulting functions are long and complex but the programmer has the option to fuse many of the loops to improve performance. The global view also allows the programmer or autotuner to use optimizations that minimize data traffic or reuse calculated values. Additionally, the tuning specifications enable tuning for specific application and input scenarios.

However, by tuning to a single application the results cannot be reused, thus necessitating more work in the future. Also, the number of loops in the overall algorithm is far greater than the individual operations (since most of the operations occur many times). As a result, the effort extended by the programmer and the search space available to the tuner are much larger.

Listing 1.1: Orio Annotation Example

```
transform Composite(
unrolljam = (['i','j','k'],[UI,UJ,UK]),
vector = (VEC, ['ivdep','vector always']),
regtile = (['i','j','k'],[RTI,RTJ,RTK])
)
for (i = 0; i <= rows_a-1; i++)
  for (j = 0; j <= cols_b-1; j++)
    for (k = 0; k <= cols_a-1; k++)
      mat_c[j + cols_b * i] +=
            mat_a[cols_a * i + k] *
            mat_b[cols_b * k + j];
```

In both cases, we used Orio to apply and test a set of optimizations to the section of code being tuned. These included loop unrolling, register tiling (the matrices are small enough to fit in registers), and vectorization of the loops. An example annotation is shown in Listing 1.1. The tuning parameters; UI, UJ, UK, RTI, RTJ, RTK; are set to reasonable ranges by the programmer for Orio to iterate over. The "U" parameters specify unroll depth while the "RT" determine the register tiling for each loop. The VEC parameter is a boolean that indicates whether the SIMD commands should or should not be included. Additionally, the optimization flags for the compilers are tested to determine if they provide significant improvement. Once tuned the results could be compiled into the complete application.

To this point all the autotuning discussed has focused on single-core performance (with SIMD options); however, parallelism has become a vital part of modern performance at all levels. We chose to focus on single threaded because the most promising parallelism strategy is tuning the code for a single thread and incorporating that into a multithreaded program (i.e., have the tracking program run Kalman updates concurrently). We include some parallelism for

completeness, adding simple OpenMP pragmas to the code prior to tuning. This method guarantees that the tuning will take the parallelism into account.

4 Results

In this case study, we present results from two types of experiments. We first compare the performance of each version of the program and then compare the power use. It is important to note that the autotuning is targeting performance not power, so the latter measurement is merely a side effect.

4.1 Performance Evaluation

Our performance results are shown in Figures 2–4, which represent the speedup of various implementations over the "base" case of the simple C code that was later autotuned. For the BLAS comparisons we use Intel's MKL, openBLAS, and ATLAS depending on system availability. The tuned versions refer to the tuning of individual linear algebra functions ("tuned LA") and the tuning of the algorithm as a whole ("tuned KAL"). The tuning with the OpenMP pragmas is listed as "LA OMP".

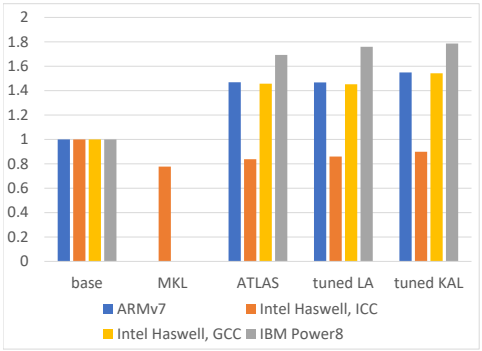


Fig. 2: Speedup for BLAS and tuned implementations over the untuned C version using generated data.

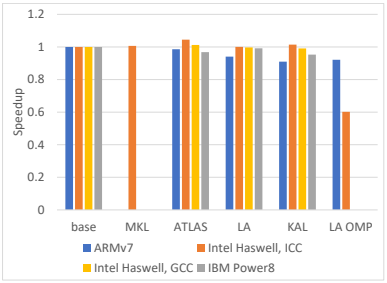
The results using synthetic data (Figure 2) show that, in many cases, significant speedup can be achieved with autotuning or other linear algebra libraries. A speedup of 1.47x is achieved by ATLAS and the tuned linear algebra library when on the ARM machine. Algorithm level tuning of the filter improves this to 1.55x speedup. The IBM machine showed slightly better results with 1.70x, 1.76x, and 1.77x speedups. For both of these machines tuning the code through

the use of general libraries or algorithm specific tuning shows significant improvement to the system.

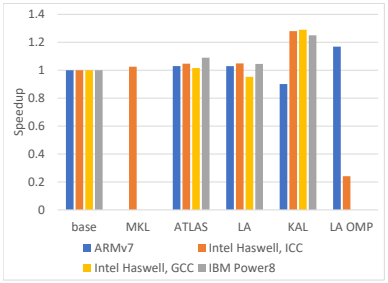
On the Intel system we used the Intel Compiler and GCC for a broader comparison. With GCC each of the versions had a speedup of roughly 1.4x, corroborating the results from the ARM and IBM machines. However, when using the Intel compiler, all of the versions (including Intel's MKL) experienced a slowdown. We suspect this is the result of Intel's ability to tailor the compiler to the specific architecture even more effectively than autotuning.

Additionally, we have the results from incorporating the Kalman filter into the full video analysis . The results from the small video (Figure 3a) show that generally the autotuning was unable to improve the performance of the program. On the ARM machine, tuning produced a small slow down, but the ATLAS library did achieve a slight speedup. The Intel machine had similar results to the synthetic data, but with less extreme slowdowns.

The results from the largest video (Figure 3b) show that the tuning and use of libraries has almost no impact on the performance with the exception of tuning the whole algorithm. When tuned at the algorithm level, we show a speedup of 1.33x on the Intel machine and 1.25x on IBM. This result indicates that tuning at the algorithm level rather than the library level has significant potential for improving performance.



(a) Small campus video with approximately 10 targets in each frame.



(b) Large video with approximately 500 targets in each frame.

Fig. 3: Speedup for BLAS and tuned implementations over the untuned C version for different numbers of targets per frame.

When OpenMP statements were inserted prior to tuning (Figures 3a and 3b) on the Intel machine, we noted a slowdown from the base code. This slowdown worsened as the number of threads increased from 4 to 64. The small matrix size prevented the benefits of thread level parallelism from overcoming the overheads.

On the ARMv7 the results were quite different, with a slight slowdown on the small video, but a speedup of 1.18x when the large video was used.

To explore how the performance varies with the number of objects tracked we tested a range of videos on the Intel machine. The number of tracked object ran from 25 to 475 in intervals of 25 for a total of 19 measurements. The results are shown in Figure 4. Almost all the variations remain consistent around the same time as the base, except the algorithm-level tuning which achieves 1.7x speedup with 475 objects per frame.

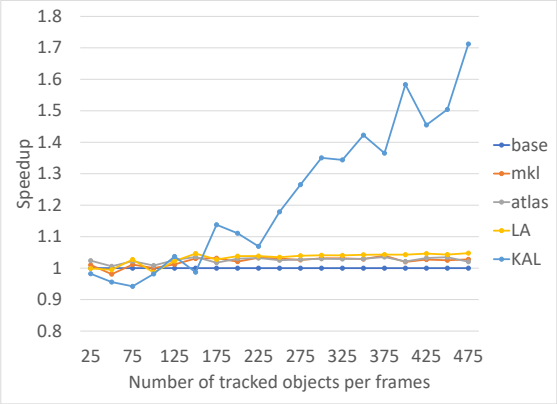


Fig. 4: Speedup for BLAS and tuned implementations over the untuned C version compared to the number of objects tracked on the Intel machine.

Overall, for some of the implementations we achieved improvement with respect to the original library and non-library-based versions, encouraging further efforts on extending Orio to exploit more available architectural features. Reliance on the compiler alone did not typically yield satisfactory results, with a surprising slowdown observed for the Intel ICC.

4.2 Power Evaluation

To explore how autotuning impacts power use, we take power measurements of the machines as the various versions of the program ran. While the tuning was intended to improve performance, it also impacts the power usage and understanding those effects is necessary to create autotuners that optimize for power use. We present the results of the power measurement in tables 2 - 3. We include the average total power and the average power above idle for each version. We also include percent different from the base version to assist with the comparison.

Table 2 shows the average power measurements taken on the ARM processor while the smaller video was being processed. Tuning at the algorithm level showed the most improvement, using 25% less power above idle than the base case did. The other tuning and library options also showed significant power reductions. For the large video on the ARM processor, the results were quite different. All versions had worse average power use than the base case with as much as 28% increase for the algorithm level tuning.

Table 2: Average Power Usage Processing the Small Video on ARMv7

	base	ATLAS	LA	KAL
average	3.89	3.54	3.48	3.45
difference from idle	1.75	1.40	1.31	1.34
% less than base	0	8.97	10.63	11.32
% less than base diff	0	19.91	23.50	25.13

Table 3: Average Power Usage Processing the Largest (500 objects) Video on ARMv7

	base	ATLAS	LA	KAL
average	3.31	3.38	3.40	3.64
difference from idle	1.18	1.24	1.27	1.51
% less than base	0	-1.99	-2.58	-10.07
% less than base diff	0	-5.59	-7.24	-28.21

The total energy use for processing each video on the ARMv7 is presented in Table 4. For the smaller video, ATLAS and the tuned versions each use slightly more energy than the base version. For the larger video the versions have minimal impact on the energy use, except for the algorithm level tuning which improves the energy use by 1.31x. These results are quite similar to those seen in Figures 3a and 3b which show slight declines in speed for the smaller video and mostly flat performance for the larger one. Notably, the large video shows only 1.07x speedup while the energy has 1.31x reduction with the algorithm level tuning, indicating that some power is saved through means other than racing to the finish.

4.3 Tuning Results

The tuning included four main optimizations: loop unrolling, tiling, SIMD pragmas, and optimization flags. The first three were applied to individual loops while the last one was applied to the overall program.

Table 4: Total Energy Use During Processing on ARMv7

	base	ATLAS	LA	KAL
small video (Ws)	253.3	279.7	285.0	286.4
version/base	1.00	0.91	0.89	0.88
large video (Ws)	42349.01	39663.68	39799.81	32223.55
version/base	1	1.07	1.06	1.31

For the library-based tuning the results (Tables 5, 6, 7, 8, and 9) show slight differences between the architectures. For example, the Intel and IBM processors uses the maximum tiling or unroll parameter on half the transformed loops, while the ARM processor prefers shallower unroll depths than the server architectures. Also, the server architectures were far more likely to unroll or tile multiple loops within the nest. Finally, The Intel processor chooses to vectorize the loops in all cases, but ARM and IBM only do so in half the loops.

The algorithm-level tuning focused on two functions: *predict* (Table 10) and *correct* (not shown for space considerations). Of the two, *predict* is simpler, and the resulting tuning was very similar to the library functions. The *correct* function was much larger in size and more difficult to tune (see Section 4.4). All three architectures required only minor transformations to the code, enabling the compiler to optimize effectively. In addition to the autotuning optimizations, some loop fusion and data reuse optimization was performed by hand on *correct* and *predict* since the global view of the algorithm was available.

Table 5: Orio parameters for matrix addition (function add_matrix())

	CLFAGS	RT i	RT j	UL i	UL j	Vectorization
Intel XEON	-O3	2	1	2	1	true
IBM Power8	-O1	1	1	2	1	false
ARMv7	-O2	6	1	1	1	false

Table 6: Orio parameters for multiplying a matrix by scalar (function multiply_matrix_by_scalar())

	CLFAGS	RT i	RT j	UL i	UL j	Vectorization
Intel XEON	-O2	1	1	1	1	true
IBM Power8	-O3	1	1	1	3	true
ARMv7	-O3	6	1	1	1	true

Although unsurprising, the differences between architectures and complexity of the tuned code underscore the importance of automatically generating and

Table 7: Orio parameters for computing transposition (function transpose_matrix())

	CLFAGS	RT i	RT j	UL i	UL j	Vectorization
Intel XEON	-O2	6	2	4	1	true
IBM Power8	-O3	2	1	2	1	true
ARMv7	-O1	6	1	1	1	true

Table 8: Orio parameters for matrix multiplication (function multiply_matrix())

	CLFAGS	RT i0	RT j0	RT i1	RT j1	RT k1
Intel XEON	-O1	6	6	6	1	1
IBM Power8	-O3	1	2	1	1	1
ARMv7	-O3	2	1	1	6	1

	UL i0	UL j0	UL i1	UL j1	UL k1	Vectorization
Intel XEON	4	1	3	1	5	true
IBM Power8	4	1	1	1	3	false
ARMv7	3	1	1	4	1	true

tuning the code to a particular architecture instead of attempting to produce optimized versions manually.

4.4 Qualitative Evaluation

Although it is easier than tuning the code by hand autotuning still requires effort by the programmer. Annotating small functions, such as the individual linear algebra functions is trivial as each contains only one or two sets of nested loops; each of which receives an annotation. Tuning the larger Kalman functions is more difficult as they are a set of linear algebra operations containing dozens of nested loops, each of which requires individual annotation and tuning variables.

In terms of code size, most functions were originally less than 100 lines long, while the *correct* function was 440 lines. As expected, some of the transformations (register tiling, loop unrolling) increase code size. Figure 5 shows the ratio between of the code size of the tuned version and the number of lines in the original untuned version. While some functions, such as *correct* did not expand

Table 9: Orio parameters for LUP factorization (function compute_LUP())

	CLFAGS	RT i1	RT j1	RT k1	RT i2	RT i3	RT j3	RT i5	RT j5
Intel XEON	-O3	1	1	6	1	1	6	6	1
IBM Power8	-O3	6	6	2	1	1	1	6	1
ARMv7	-O2	1	2	2	1	2	2	2	1

	UL i1	UL i2	UL i3	UL i5	UL j1	UL j3	UL j5	UL k	Vect.
Intel XEON	1	2	2	2	1	1	1	2	true
IBM Power8	1	1	2	4	1	1	1	1	false
ARMv7	1	3	1	1	2	1	1	1	false

Table 10: Orio parameters for the Kalman prediction (function predict())

	CLFAGS	RT i02	RT i03	RT j03	RT i04	RT j04	RT i1	
Intel XEON	-O3	2	1	6	1	1	1	
IBM Power8	-O1	2	6	1	2	2	1	
ARMv7	-O2	2	1	2	2	1	6	
	RT j1	RT i2	RT k2	RT i3	RT j3	RT k3	RT i4	
Intel XEON	6	1	1	1	2	1	6	
IBM Power8	6	1	2	1	1	1	1	
ARMv7	6	1	1	1	6	1	2	
	RT j4	RT k4	RT i5	RT j5	UL i02	UL i03	UL j03	
Intel XEON	1	1	2	6	2	1	5	
IBM Power8	2	2	6	1	4	5	1	
ARMv7	6	2	2	2	2	1	1	
	UL i04	UL j04	UL i1	UL j1	UL i2	UL k2	UL i3	
Intel XEON	1	3	4	1	1	1	2	
IBM Power8	1	3	2	1	5	1	1	
ARMv7	2	1	1	1	4	1	1	
	UL j3	UL k3	UL i4	UL j4	UL k4	UL i5	UL j5	Vect.
Intel XEON	1	5	3	1	5	2	1	true
IBM Power8	1	2	1	2	1	5	1	true
ARMv7	4	1	1	3	5	5	1	false

significantly, more complex code resulted in tuned code size that was up to 26.8 times larger than the original version. There was also significant variation in code bloat among the different target architectures.

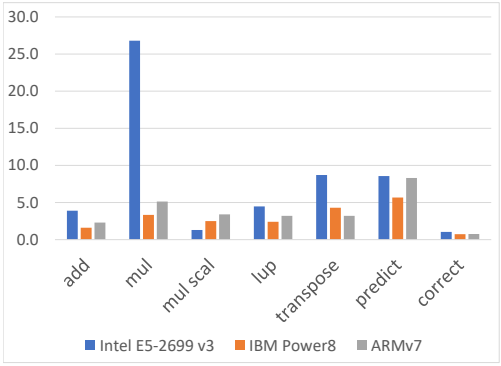


Fig. 5: Ratio of the code size of the best autotuned version w.r.t. the original unoptimized version.

There is a similar difference in the time it takes to tune. The smaller functions took much less time to tune, so we were able to exhaustively explore the search space. Larger functions necessitated the sampling of the search space with a limit set on the number of runs. In those cases, we used random sampling with a focused sample once a maximum is found, except for the correct function which needed a lower cap on the runs, so we utilized the Simplex search strategy. Full details are available in Table 11.

Table 11: Approximate time to tune each function.

Function	Search Strategy	Intel XEON	IBM Power8	ARMv7
addition	Exhaustive	10 min	10 min	30 min
transposition	Exhaustive	10 min	10 min	30 min
scalar-matrix multiplication	Exhaustive	10 min	10 min	30 min
matrix-matrix multiplication	Random (100,000)	40 min	40 min	1 hour
LUP factor	Random	40 min	40 min	1 hour
Kalman predict	Random (1,000,000)	30 min	30 minutes	1 hour
Kalman correct	Simplex(100,000)	3 hours	3 hours	6 hours

5 Future Work

For future work focuses on the continued improvement of autotuning techniques available in the Orio framework. In particular, the addition of new code transformations and power as a tuning target or constraint.

While we included some algorithm level tuning in this case study, those optimizations were added by hand. Orio can be extended to perform those types of transformations. Furthermore, additional parallel techniques using pthreads, Intel TBB, or MPI will be explored.

Also, we will add power as a target for tuning. While important for HPC research, IoT presents another interesting target since many of the applications run indefinitely (e.g., traffic lights, environmental sensors) and have inconsistent power supplies. These challenges add to the difficulty of tuning for power use.

6 Conclusion

We presented a case study of autotuning Kalman Filters on three disparate architectures. Our autotuning took several approaches, a linear algebra library based approach, an algorithm level approach, and tuning within OpenMP annotated loops. The performance and power results show clear areas where autotuning research can improve the performance of both HPC and IoT targeted software.

References

1. W Kyle Anderson, William D Gropp, Dinesh K Kaushik, David E Keyes, and Barry F Smith. Achieving high sustained performance in an unstructured mesh cfd application. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, page 69. ACM, 1999.
2. Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 303–316. ACM, 2014.
3. Prasanna Balaprakash, Ananta Tiwari, and Stefan M Wild. Multi objective optimization of hpc kernels for performance, power, and energy. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, pages 239–260. Springer, 2013.
4. Erhan Bas, A Murat Tekalp, and F Sibel Salman. Automatic vehicle counting from video for traffic flow analysis. In *Intelligent Vehicles Symposium, 2007 IEEE*, pages 392–397. Ieee, 2007.
5. James Bergstra, Nicolas Pinto, and David Cox. Machine learning for predictive auto-tuning with boosted regression trees. In *Innovative Parallel Computing (InPar), 2012*, pages 1–9. IEEE, 2012.
6. David Beymer, Philip McLauchlan, Benjamin Coifman, and Jitendra Malik. A real-time computer vision system for measuring traffic parameters. In *Computer Vision and Pattern Recognition, 1997. Proceedings., 1997 IEEE Computer Society Conference on*, pages 495–501. IEEE, 1997.
7. Samuel S Blackman. Multiple-target tracking with radar applications. *Dedham, MA, Artech House, Inc., 1986, 463 p.*, 1986.
8. Uday Bondhugula, A Hartono, J Ramanujam, and P Sadayappan. Pluto: A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*, 2008.
9. Uday Bondhugula, A Hartono, J Ramanujam, and P Sadayappan. Pluto: A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*, 2008.
10. Amarjit Budhiraja, Lingji Chen, and Chihoon Lee. A survey of numerical methods for nonlinear filtering problems. *Physica D: Nonlinear Phenomena*, 230(1):27–36, 2007.
11. Nick Chaimov, Boyana Norris, and Allen Malony. Toward multi-target autotuning for accelerators. In *Parallel and Distributed Systems (ICPADS), 2014 20th IEEE International Conference on*, pages 534–541. IEEE, 2014.
12. Chun Chen, Jacqueline Chame, and Mary Hall. Chill: A framework for composing high-level loop transformations. Technical report, Technical Report 08-897, U. of Southern California, 2008.
13. Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 4. IEEE Press, 2008.
14. Jack J Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):1–17, 1990.

15. Jack J Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J Hanson. An extended set of fortran basic linear algebra subroutines. *ACM Trans. Math. Soft*, 14(1):1–17, 1988.
16. Wei Fang, Yong Zhao, Yule Yuan, and Kai Liu. Real-time multiple vehicles tracking with occlusion handling. In *Image and Graphics (ICIG), 2011 Sixth International Conference on*, pages 667–672. IEEE, 2011.
17. Surendra Gupte, Osama Masoud, Robert FK Martin, and Nikolaos P Papanikolopoulos. Detection and classification of vehicles. *IEEE Transactions on intelligent transportation systems*, 3(1):37–47, 2002.
18. Albert Hartono, Boyana Norris, and Ponnuswamy Sadayappan. Annotation-based empirical performance tuning using orio. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–11. IEEE, 2009.
19. Herbert Jordan, Peter Thoman, Juan J Durillo, Simone Pellegrini, Philipp Gschwandtner, Thomas Fahringer, and Hans Moritsch. A multi-objective auto-tuning framework for parallel codes. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–12. IEEE, 2012.
20. Rudolph Emil Kalman et al. A new approach to linear filtering and prediction problems. *Journal of basic Engineering*, 82(1):35–45, 1960.
21. Robert V Lim, Boyana Norris, and Allen D Malony. Autotuning gpu kernels via static and predictive analysis. *arXiv preprint arXiv:1701.08547*, 2017.
22. Richard Ménard, Stephen E Cohn, Lang-Ping Chang, and Peter M Lyster. Assimilation of stratospheric chemical tracer observations using a kalman filter. part i: Formulation. *Monthly weather review*, 128(8):2654–2671, 2000.
23. Thomas Nelson, Geoffrey Belter, Jeremy G Siek, Elizabeth Jessup, and Boyana Norris. Reliable generation of high-performance matrix algebra. *ACM Transactions on Mathematical Software (TOMS)*, 41(3):18, 2015.
24. Shah Faizur Rahman, Jichi Guo, and Qing Yi. Automated empirical tuning of scientific codes for performance and power consumption. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, pages 107–116. ACM, 2011.
25. See homepage for details. Atlas homepage. http://math-atlas.sourceforge.net/.
26. Mohammad Shokrolah Shirazi and Brendan Morris. Contextual combination of appearance and motion for intersection videos with vehicles and pedestrians. In *International Symposium on Visual Computing*, pages 708–717. Springer, 2014.
27. Mohammad Shokrolah Shirazi and Brendan Morris. A typical video-based framework for counting, behavior and safety analysis at intersections. In *2015 IEEE Intelligent Vehicles Symposium (IV)*, pages 1264–1269. IEEE, 2015.
28. Mohammad Shokrolah Shirazi and Brendan Morris. Vision-based pedestrian monitoring at intersections including behavior & crossing count. In *2016 IEEE Intelligent Vehicles Symposium (IV)*, pages 1022–1027. IEEE, 2016.
29. Mohammad Shokrolah Shirazi and Brendan Tran Morris. Vision-based pedestrian behavior analysis at intersections. *Journal of Electronic Imaging*, 25(5):051203–051203, 2016.
30. Mohammad Shokrolah Shirazi and Brendan Tran Morris. Vision-based turning movement monitoring: count, speed & waiting time estimation. *IEEE Intelligent Transportation Systems Magazine*, 8(1):23–34, 2016.
31. Huan-Sheng Song, Sheng-Nan Lu, Xiang Ma, Yuan Yang, Xue-Qin Liu, and Peng Zhang. Vehicle behavior analysis using target motion trajectories. *IEEE Transactions on Vehicular Technology*, 63(8):3580–3591, 2014.

32. Chris Stauffer and W Eric L Grimson. Adaptive background mixture models for real-time tracking. In *Computer Vision and Pattern Recognition, 1999. IEEE Computer Society Conference on.*, volume 2. IEEE, 1999.
33. Chris Stauffer and W. Eric L. Grimson. Learning patterns of activity using real-time tracking. *IEEE Transactions on pattern analysis and machine intelligence*, 22(8):747–757, 2000.
34. Ananta Tiwari and Jeffrey K Hollingsworth. Online adaptive code generation and tuning. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 879–892. IEEE, 2011.
35. Greg Welch and Gary Bishop. An introduction to the kalman filter. 1995.
36. R. Clint Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005. `http://www.cs.utsa.edu/~whaley/papers/spercw04.ps`.
37. Kamen Yotov, Xiaoming Li, Gang Ren, Michael Cibulskis, Gerald DeJong, Maria Garzaran, David Padua, Keshav Pingali, Paul Stodghill, and Peng Wu. A comparison of empirical and model-driven optimization. *ACM SIGPLAN Notices*, 38(5):63–76, 2003.
38. Zoran Zivkovic and Ferdinand van der Heijden. Efficient adaptive density estimation per image pixel for the task of background subtraction. *Pattern recognition letters*, 27(7):773–780, 2006.