NLAFET
Parallel Numerical Linear Algebra for Future Extreme Scale Systems

H2020–FETHPC–2014: GA 671633

D6.4

An Off-line Autotuning Framework based on Heuristic Search

April 2017

Document information

Scheduled delivery	2017-04-30
Actual delivery	2017-04-27
Version	1.0
Responsible partner	UNIMAN

Dissemination level

PU — Public

Revision history

Date	Editor	Status	Ver.	Changes
2017-04-18	Samuel Relton	Complete	1.0	Feedback from reviewers added.
2017-03-27	Samuel Relton	Draft	0.1	Initial version of document produced.

Author(s)

Negin Bagherpour (UNIMAN)
Jack Dongarra (UNIMAN)
Samuel Relton (UNIMAN)
Mawussi Zounon (UNIMAN)

Internal reviewers

Nicholas Higham (UNIMAN)
Mirko Myllykoski (UMU)
Lars Karlsson (UMU)
Stojce Nakov (STFC)

Copyright

Acknowledgements

This project has received funding from the *European Union's Horizon 2020 research and innovation programme* under the grant agreement number 671633.

Table of Contents

1 Introduction 3

2 Review of existing software 4

3 Applying OpenTuner within NLAFET 5

4 Curve fitting 6

5 Performance results 9

6 Conclusions 10

List of Figures

1 The optimal tile size for DGEMM is independent of the matrix size, so a constant model is most appropriate. On this architecture the constant is 304. 7

2 The optimal tile size for DGETRF appears to depend linearly on the matrix size. In this model the intercept is 92.25 with a gradient of 0.008. 7

3 The optimal tile size for SPOTRF appears to depend logarithmically on the matrix size. The least-squares fit provides the tile size function $y = -898 + 135 \log(x)$. 8

4 The optimal tile size for DPOTRF appears to be a step function. The three constants in this step function are 124, 291, and 487. 8

5 Lua code implementing the curves fitted to the single and double precision Cholesky factorizations. The arguments to the function are `dtype`, a character which determines the precisions of the computation, and `n`, the size of the matrix. 9

6 Tuned and untuned performance for the GEMM kernel. Single precision is on the left and double precision on the right. 10

7 Tuned and untuned performance for Cholesky factorization. Single precision is on the left and double precision on the right. 10

8 Tuned and untuned performance for *LU* factorization. Single precision is on the left and double precision on the right. 11

9 Tuned and untuned performance for *QR* factorization. Single precision is on the left and double precision on the right. 11

1 Introduction

The *Description of Action* (DoA) document states for deliverable D6.4:

> "*D6.4: An off-line auto-tuning framework based on heuristic search*
>
> Review of techniques for pruning the search space in the context of autotuning, resulting in prototypes for an offline auto-tuning framework based on heuristic search. Includes reporting on optimal circumstances for switching scheduling approaches at runtime."

This deliverable is in the context of Task 6.2 (Auto-Tuning).

In most pieces of scientific software there are a number of parameters that can be tweaked to obtain better performance. These can take a variety of forms including continuous variables (e.g. a parameter in a model of some data), integer variables (e.g. the number of threads to spawn in a parallel section of code), or even categorical variables (e.g. on/off switches for compiler flags).

In this document we discuss, with a specific focus on the needs of the NLAFET project, how to optimize such parameters via autotuning. We propose a methodology that can be applied to the various pieces of software produced by the NLAFET consortium. In particular we concentrate on off-line autotuning; meaning that the optimisation is performed before the user runs the software on their own problems. An alternative approach is on-line autotuning, where the parameters are allowed to vary during the execution of a large problem. This latter approach will be covered in other NLAFET deliverables.

To begin, let us explain the specific needs of the NLAFET project with regards to off-line autotuning. The NLAFET project aims to deliver software for a large number of linear algebra problems, on both shared and distributed memory architectures, making use of runtime systems such as OpenMP [4], StarPU [3], and PaRSEC [5].

Many of the algorithms employed to solve these linear algebra problems involve decomposing a large matrix into smaller "blocks". These blocks are then dealt with by separate cores (or nodes in the distributed memory setting) and are later combined to give the final solution to the problem. This approach is the key idea behind the PLASMA and MAGMA projects [1] and has been incorporated into many other libraries such as Intel MKL. In terms of autotuning, this means that positive integers controlling the size of the blocks must be chosen to maximize performance. Note that the optimal tile size may be different for each linear algebra routine. For example, performing a matrix multiplication may require a different tile size to an LU factorization.

Other algorithms, especially for sparse matrices, tend to be iterative in nature. In particular some algorithms have two layers of iteration (an inner and outer iteration) and the convergence tolerance of each iteration can differ. Increasing the error tolerance of the iterations leads to a faster solution, sometimes at the expense of accuracy, making these values continuous tunable parameters.

Finally, all these algorithms need to be compiled across a number of different architectures with different compilers, various number of cores, and various number of GPUs etc. The compiler flags used during the compilation can have a dramatic effect on the overall performance. For example, whether or not vectorization is enabled when compiling with gcc will have a large impact on performance. Similarly using the flags -O1, -O2, or -O3 during the compilation can have a dramatic impact on the overall runtime. Further examples include parameters to select the runtime scheduling strategy in StarPU etc. These categorical parameters also need to be optimized to achieve the best possible performance.

The large number of parameters to optimize leads to a combinatorial explosion of possible options. Furthermore, due to the variety of parameter types that need to be optimized (integer, continuous, and categorical) a very general optimization routine is required. We believe that genetic algorithms are well-suited to optimize such high-dimensional problems with various parameter types. Currently, many pieces of software attempt to perform a "grid-sweep" which tries all (or a representative sample) of the parameter combinations before selecting the best. The grid-sweep will act as a baseline for us to test the efficiency of other optimisation strategies considered.

To test our choice of autotuning strategy, we will show the results when this procedure is employed to tune the PLASMA library for dense linear algebra. For this particular library we will be optimizing, and fitting models to, a single parameter. Extensions to multiple parameters are discussed in the conclusions.

The rest of this report is organised as follows. In section 2 we compare a number of existing off-line autotuning approaches, aiming to determine which one is most suitable for the NLAFET project. In section 3 we give more details on how the chosen strategy can be applied to linear algebra software by combining it with a Lua interface [7]. Then in section 4 we analyze some typical behaviours exhibited by the tile size and show how fitting curves to the tile size data allows us to select near-optimal tile sizes for all sizes of input matrix. In section 5 we show how our tuning strategy improves the performance of PLASMA for a number of different routines before giving some concluding remarks in section 6.

2 Review of existing software

There are a number of off-line autotuning frameworks already available from the HPC community, and they are largely split into two categories:

- those which require modification to the source code, and
- those which interface with the source code via, for example, a configuration file.

The primary advantages of frameworks that modify the source code directly (such as the Periscope Tuning Framework [6], [8]) are that the developer has fine-grain control over specifically what is being optimized and that different sections of the code can be optimized independently, possibly speeding up the autotuning process. On the other hand, the overall code must be linked up to the tuning library, which increases the number of dependencies of the software. Issues can also arise in the future if the chosen autotuning framework becomes unsupported due to lack of funding, or fails to compile on new architectures.

Performing autotuning through some intermediary interface has one distinct advantage over the previous approach: it is more modular and flexible. This separation between the computational routines and the autotuning means that, if our chosen optimizer fails to run on a certain architecture it can easily be replaced by another. One disadvantage of this approach is that different parts of the software cannot be tuned independently: the entire algorithm must be re-run to try a new set of parameters.

Since the NLAFET project targets multiple architectures, including upcoming ARM-based HPC machines [9] that are not readily available at present, we prefer to take the second approach and perform autotuning through a configuration interface. In order to maximize the number of architectures where our autotuning can be performed, we would like our optimizer to be written in languages such as C or Python, which are supported in all major operating systems.

Although there are a large number of autotuning frameworks available for download, the only one we have found that fits all of our desired criteria is OpenTuner [2]. OpenTuner is a Python module with minimal dependencies that can be used to optimize all the quantities mentioned previously.

A particularly interesting feature of OpenTuner is its use of ensemble optimisation: a large number of algorithms (including genetic optimisation and simulated annealing etc.) are fed into a multi-armed bandit model which detects the algorithms that are performing well on the current problem and gives them a larger share of the optimisation time. More detail on the specifics can be found in the reference above.

3 Applying OpenTuner within NLAFET

Now that we have decided on a tuning approach, this section describes in more detail how we plan to perform tuning withing NLAFET, using the PLASMA project (for dense linear algebra) as an example.

To give an overview of our approach, we aim to use optimization software such as OpenTuner (or even a full grid-sweep) to determine optimal values of the tile size for a range of different matrix shapes and sizes. Once these values have been found, we need to inform PLASMA which tile size is appropriate for each matrix via an intermediary interface. However, instead of using a simple configuration file we opt to use a Lua script.

Lua is a lightweight, portable, embeddable, and open-source scripting language that is ideal for making small extensions to larger software projects. Its only dependency is the availability of a C compiler so it does not hamper the portability of the overall software. In this scenario, we can use Lua to return an "optimal" tile size for matrices with sizes that we haven't tested by interpolating the values seen during our autotuning runs. Also, if we can fit a smooth curve (or surface) to our optimal parameters (which will be explored later) then this curve can be coded into Lua to provide parameters for matrices with various sizes. This provides a much more powerful and flexible interface than a simple configuration file allows.

Therefore, we have two separate problems to tackle. First, we need to use OpenTuner, or some other method such as a grid-sweep, to find optimal parameters for a range of matrix sizes. Second, we would like to fit a function to these points (so that a good value of the tile size can be returned for any matrix size) before writing Lua code to implement this function. From here, the software can simply call the Lua script to obtain the parameters for any input matrix. This second issue is rather difficult to automate: there are a variety of different functions needed to fit the various behaviours we observe the tile size showing. However, it would be easy to automate this latter part by using spline interpolation or kernel density estimation for example, at the expense of possibly overfitting the observed parameters.

For the first issue, performing a grid-sweep over all tile sizes for a variety of matrix sizes is very simple and the optimal parameters can easily be stored in a csv file. The downside is that grid sweeps are extremely expensive. In order to use OpenTuner, to reduce the time for the sweep, there is a small amount of extra work required: we must define a Python class with the following 3 functions

- `manipulator` – to define the search space,

- `run` – to run our routine with the current parameters, and,
- `save_final_config` – to save the final parameters.

In `manipulator` we simply define the tile size as a multiple of 8 (the width of the vector units on most CPUs) between 80 and 520. Within `run` we simply run the routine we are currently investigating with the current tile size and take the average time over 5 runs (where 5 is actually a user-chosen parameter). Finally in `save_final_config` we output the “optimal” tile size in JSON format. A wrapper script collates the JSON files for various matrix sizes into a single CSV file.

The actual optimization is taken care of by OpenTuner itself, meaning that there is nothing else to do but to limit the number of tests performed (i.e. function calls) using the command-line argument `--test-limit=x`. This was set to 30 for our experiments whereas doing a full grid sweep over all tile sizes requires 55 runs for each matrix size; we can save almost half the sweep time by using OpenTuner.

We can also define multiple initial guesses for the optimal tile size using the command-line argument `--seed-configuration=f` where `f` is a JSON file containing, for example, `{"blocksize": 256}`. We used this latter option to define a few initial guesses for the powers of 2 etc. We found that OpenTuner was able to identify maxima close to those provided by a grid sweep with half the number of runs per matrix size: reducing the overall optimization time (for all the routines we considered) from 12 to 6 hours. Note that we have only considered 4 routines in this report (in both single and double precision) from over 50 routines in the current version of PLASMA, so using OpenTuner could reduce the optimization time by a number of days in a larger experiment.

4 Curve fitting

Once we have found the optimal tile sizes for a variety of matrix sizes we would like to fit a curve to the resulting data, which can then be coded in Lua. Using this fitted curve we can then return near-optimal parameters for any input matrix, regardless of its size. The following experiments were performed on a 2 socket NUMA node with 20 Haswell cores (2 x Xeon(R) CPU E5-2650 v3, 2.30GHz).

In this section we will show the variety of behaviours that the optimal tile size can take as the matrix size changes, propose a number of models to fit the resulting data, and describe how the parameters of such models can be estimated. We found that the following four models can cover all of the routines we investigated.

- Constant models - tile size does not depend on the input matrix size.
- Linear models - tile size increases linearly with the input matrix size.
- Logarithmic models - tile size increases logarithmically with the input matrix size.
- Piecewise models - A combination of the other three models is required, for example a step function is a combination of constant models.

First we investigate the performance of DGEMM for matrix multiplication. As we can see from Figure 1, the optimal tile size is largely independent of the matrix size. Therefore, we use the mean of all the values to return a constant tile size: in this case the constant is 304. The reason for the lack of dependence on the matrix size is fairly straightforwards,

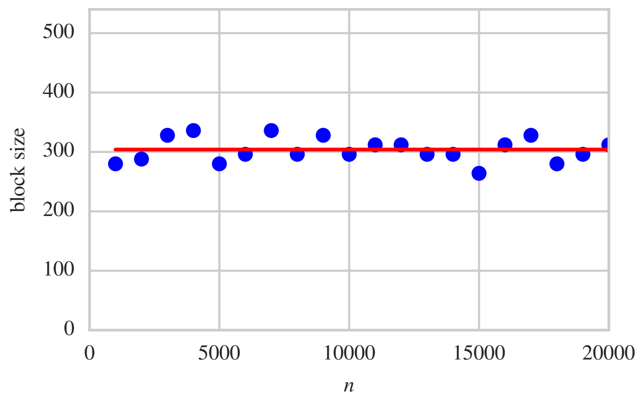


Figure 1: The optimal tile size for DGEMM is independent of the matrix size, so a constant model is most appropriate. On this architecture the constant is 304.

since DGEMM is very arithmetic intensive, involving only fused multiply-adds, the tile size is affected only by the memory hierarchy and cache size.

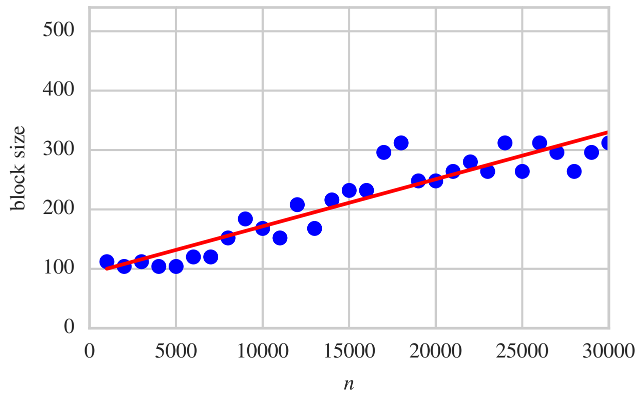


Figure 2: The optimal tile size for DGETRF appears to depend linearly on the matrix size. In this model the intercept is 92.25 with a gradient of 0.008.

Next we investigate DGETRF for *LU* factorization with partial pivoting in Figure 2. In this experiment it is clear that a linear model provides a much better fit for the data. Doing a least-squares fit gives an intercept of 92.25 and a gradient of 0.008.

In Figure 3 we observe that the tile size for SPOTRF (Cholesky factorization in single precision) increases logarithmically with the matrix size. The least-squares fit of a model $y = a + b\log(x)$ on a set of k data points (x_i, y_i) gives the parameters

$$\begin{aligned} b &= \frac{k\sum y_i \log(x_i) - \sum y_i \sum \log(x_i)}{k\sum \log(x_i)^2 - (\sum \log(x_i))^2}, \\ a &= \frac{\sum y_i - b\sum \log(x_i)}{k}. \end{aligned}$$

Fitting this model to our data gives the model $y = -898 + 135\log(x)$.

Finally, by looking at the performance of DPOTRF (Cholesky factorization in double precision) in Figure 4, we see an example of a routine where a piecewise function is the best choice to model the data. In this particular case a step function is an appropriate

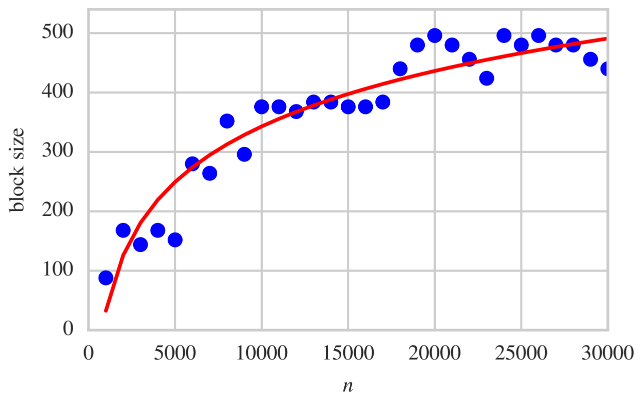


Figure 3: The optimal tile size for SPOTRF appears to depend logarithmically on the matrix size. The least-squares fit provides the tile size function $y = -898 + 135\log(x)$.

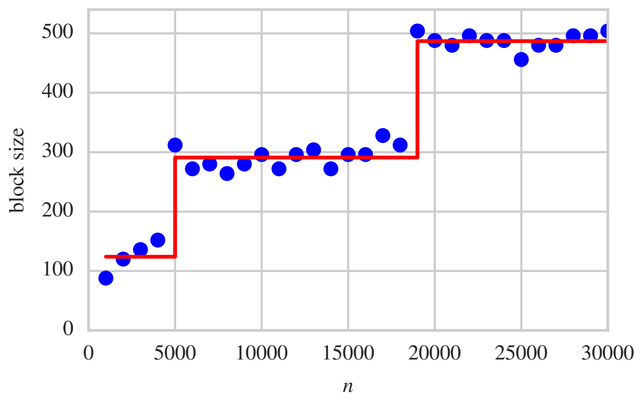


Figure 4: The optimal tile size for DPOTRF appears to be a step function. The three constants in this step function are 124, 291, and 487.

model, though there are other routines where initially the function is logarithmic before settling at some constant value. In this particular step function the value changes when the matrix sizes are 5000 and 19000, whilst the three constants are 124, 291, and 487.

In summary there are a variety of different behaviours that the tile size can exhibit depending upon the routine in question. Note that even performing the same routine in different precisions can lead to drastically differing behaviours (compare Figures 3 and 4). This is likely due to the complex interaction between the memory hierarchy and the width of the SIMD units within each core.

As such, we can only recommend that each routine is treated independently and no initial assumptions are made about their behaviour. Furthermore, these results are only measured on a Haswell NUMA node. Moving to more recent architectures, or the Intel Xeon Phi (codenamed Knights Landing) may lead to drastically different behaviours. Clearly, once these trends have been found, they need to be coded within Lua functions so that the software can find the appropriate tile sizes at runtime. As an example, the Lua code for the single and double precision variants of the Cholesky factorization is given in Figure 5.

```
function potrf_nb (dtype, n)
    if dtype == 'S' then
        if n <= 1000 then
            return 128
        else
            return math.min(512,
            math.floor(-898.372659 + 134.77263 * math.log(n)))
        end
    elseif dtype == 'D' then
        if n < 5000 then
            return 124
        elseif n < 19000 then
            return 291
        else
            return 487
        end
    else
        return 256
    end
end
```

Figure 5: Lua code implementing the curves fitted to the single and double precision Cholesky factorizations. The arguments to the function are `dtype`, a character which determines the precisions of the computation, and `n`, the size of the matrix.

5 Performance results

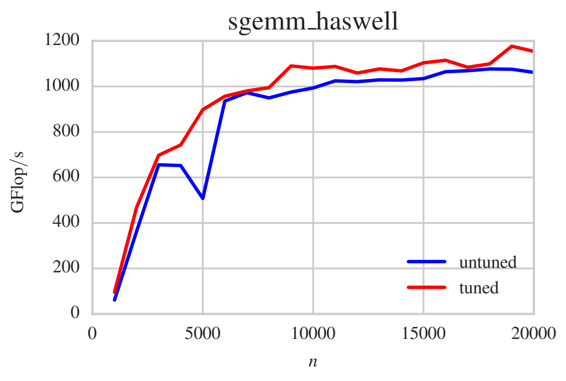
In this section we aim to demonstrate the improvements in performance that can be gained by applying the procedure described in the previous sections. Using a 2 socket NUMA node with 20 Haswell cores (2 x Xeon(R) CPU E5-2650 v3, 2.30GHz) we ran the OpenTuner procedure to obtain the best tile sizes for various matrix sizes and fit various curves to the resulting data—as described in section 4—before encoding these curves in Lua functions.

We will compare the performance of OpenMP PLASMA using the default parameters (i.e. taking a tile size of 256) against our tuned version for four different routines: matrix multiplication, Cholesky factorization, *LU* factorization, and *QR* factorization. In each case we will show the performance using square matrices of various sizes in both single and double precision.

First, in Figure 6, we look at the performance of GEMM (matrix multiplication) before and after tuning. Single precision is on the left and double precision on the right. As we can see, the tuned version always performs at least as well as the untuned version and is often superior. Both versions converge to a similar level of performance, with the tuned version being marginally better.

Next we consider POTRF (the Cholesky decomposition) in Figure 7. As before we see that the tuned version is almost always superior to the untuned version and the two versions appear to converge to similar GFlop rates.

In Figure 8 we see the results of GETRF (the *LU* factorization). In this case we see that the tuned version is always preferable and the gap between the two versions has



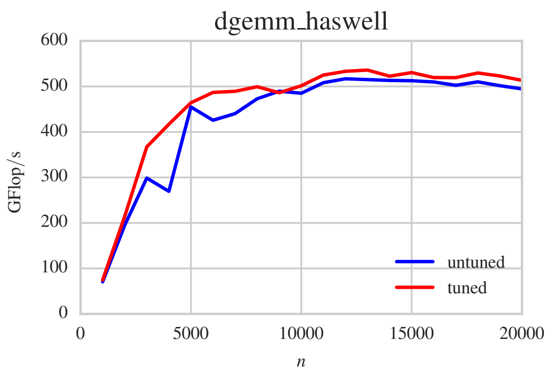
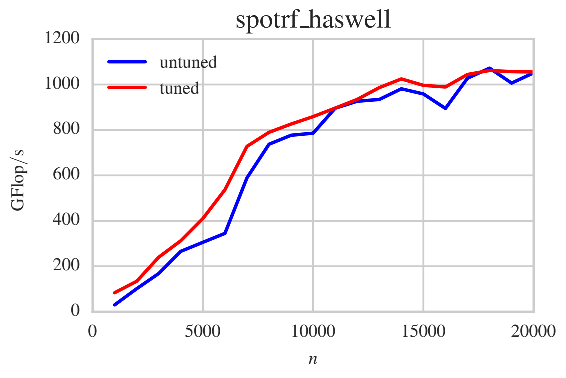


Figure 6: Tuned and untuned performance for the GEMM kernel. Single precision is on the left and double precision on the right.



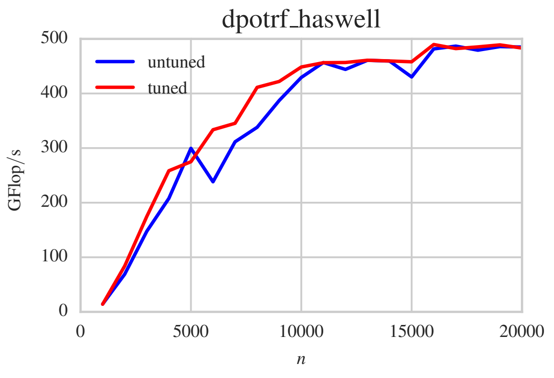


Figure 7: Tuned and untuned performance for Cholesky factorization. Single precision is on the left and double precision on the right.

widened significantly. Indeed for the larger matrices in our tests we see a performance gap of over 100 GFlop/s between them, in both precisions.

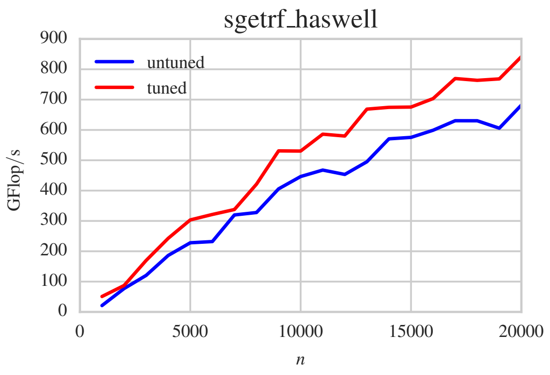
Finally, Figure 9 shows the results for GEQRF (the QR factorization). In single precision arithmetic the tuned version is only slightly better than the untuned version. However, in double precision, the tuned version is far superior and is around 150 GFlop/s faster at one point. In both cases the two versions converge to a similar performance as the matrix size increases, though the tuned version is slightly faster.

6 Conclusions

To conclude, we have described a process to move from a piece of software with no tuning whatsoever towards software which is highly tuned for a given architecture. Our process makes use of the OpenTuner optimization software to find optimal parameters, combined with data analysis and curve fitting techniques which can then be translated to the Lua scripting language.

As seen from our experiments with the PLASMA library for dense linear algebra, we can obtain results that are superior to untuned versions in almost every test case.

Future work in this area could focus on functions with multiple tuning paramers. In such functions it may be more difficult to fit a surface to the data found by OpenTuner.



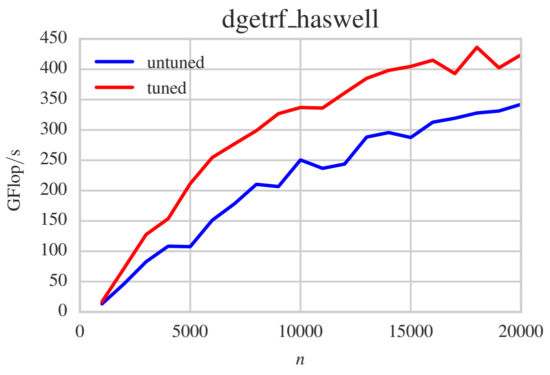
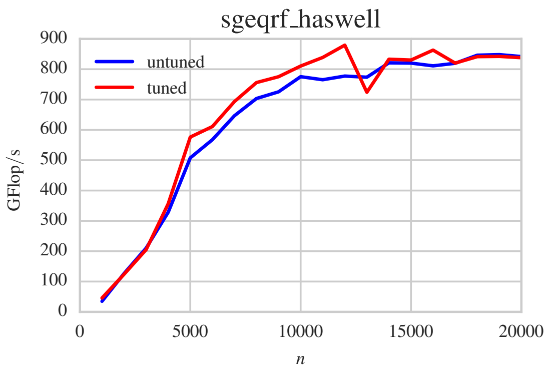


Figure 8: Tuned and untuned performance for LU factorization. Single precision is on the left and double precision on the right.



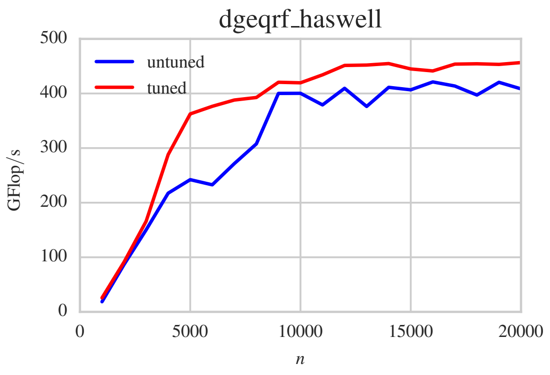


Figure 9: Tuned and untuned performance for QR factorization. Single precision is on the left and double precision on the right.

Some possible ways to tackle this may be to fit each parameter separately and use a copula (an idea borrowed from statistics), or to apply Gaussian process regression in this multi-dimensional setting. We could also use spline interpolation or kernel density estimation as mentioned previously.

Further research could also look at simulating performance results as opposed to measuring them directly. This would significantly reduce the time needed to optimize the parameters, but reliability may suffer as a result. Simulators such as SimGrid (tied to StarPU) have shown promising results in this area.

References

[1] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, 180(1), 2009.

[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 *International Conference on Parallel Architectures and Compilation Techniques*, Edmonton, Canada, August 2014.

[3] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, February 2011.

[4] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of openmp tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, 2009.

[5] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J Dongarra. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering*, 15(6):36–45, 2013.

[6] R. Miceli et. al. AutoTune: a plugin-driven approach to the automatic tuning of parallel applications. *Applied Parallel and Scientific Computing. PARA 2012. Lecture Notes in Computer Science*, 7782:328–342, 2013.

[7] Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes. Lua 5.1 reference manual, 2006.

[8] Y. Oleynik, R. Mijakobić, I. A. Comprés Ureña, M. Firbach, and M. Gerndt. *Tools for High Performance Computing*, chapter Recent Advances in Periscope for Performance Analysis and Tuning, pages 39–51. Springer, 2014.

[9] J. Weloli, S. Bilavarn, S. Derradji, C. Belleudy, and S. Lesmanne. Efficiency modeling and analysis of 64-bit ARM clusters for HPC. In *Digital System Design (DSD)*, 2016.