CUDA-level Performance with Python-level Productivity for Gaussian Mixture Model Applications

H. Cook, E. Gonina, S. Kamil, G. Friedland[†], D. Patterson, and A. Fox
Parallel Computing Laboratory, Computer Science Division, University of California at Berkeley
[†]International Computer Science Institute
{hcook, egonina, skamil}@eecs.berkeley.edu, fractor@icsi.berkeley.edu, {pattrsn, fox}@eecs.berkeley.edu

Abstract

Typically, scientists with computational needs prefer to use high-level languages such as Python or MATLAB; however, large computationally-intensive problems must eventually be recoded in a low level language such as C or Fortran by expert programmers in order to achieve sufficient performance. In addition, multiple strategies may exist for mapping a problem onto parallel hardware; unless the hardware geometry and problem dimensions are both taken into account, large factors of performance may be left on the table. We show how to preserve the productivity of high-level languages while obtaining the performance of the best low-level language code variant for a given hardware platform and problem size using SEJITS (Selective Embedded Just-in-Time Specialization), a set of techniques that leverages just-in-time code generation and compilation combined with reflection and metaprogramming. As a case study, we demonstrate our technique for Gaussian Mixture Model training using the EM algorithm. With the addition of one line of code to import our framework, a domain programmer using an existing Python GMM library can run her program unmodified on a GPU-equipped computer and achieve performance that meets or beats GPU code hand-crafted by a human expert. We also show that despite the overhead of allowing the domain expert's program to use Python and the overhead of just-in-time code generation and compilation, our approach still results in performance competitive with hand-crafted GPU code.

1 Introduction

Domain experts coding computationally-intensive programs would prefer to work at a high level of abstraction such as that afforded by scripting languages like Python or MATLAB. However, it is widely accepted that recoding compute-intensive "kernels" in lower-level languages to express explicit parallelism can yield one to three orders of magnitude in performance improvements, creating a tension between programmer productivity and high performance.

The advent of multicore CPUs and manycore GPUs aggravates this tension: the best parallel implementation of a particular algorithm now depends on the target hardware and the specific input data, as evidenced by the fact that autotuning, the automated search of possible implementation variants and tuning parameters, often yields code that surpasses the performance of expert-created low-level code [21]. Yet even when tuning is complete, running the tuned code on a different hardware platform or a new problem size may result in unpredictable performance cliffs [10].

The tedious process of code variant selection and parameter tuning works against domain-programmer productivity (if it is within the domain programmer's expertise at all). Autotuning libraries such as OSKI [19] and Spiral [18] attempt to encapsulate multiple code variants and heuristics for choosing among them, as well as heuristics for selecting tuning parameters for the chosen variant; however, this machinery is specific to each library and not generally repurposable.

In this paper we show that the mechanism and policy for variant selection and tuning can be separated from the application logic in a way that increases productivity for both the application programmer and the performance tuning specialist. Our framework allows the programmer to express her application in a highly productive language (Python). Adding a single import statement pulls in a set of just-in-time code generation mechanisms and hides the variant selection logic from the domain expert programmer, synthesizing the "best" variant at runtime and giving performance comparable to or better than hand-coded implementations by a human expert.

Our case study focuses on a computationally-intensive algorithm for training Gaussian Mixture Models (GMMs), a particular class of statistical models used in speech recognition, image segmentation, document classification, and numerous other areas. The iterative and highly data-parallel algorithm is amenable to execution on GPUs; however, depending on the hardware geometry and dimensionality of the input data (which varies greatly across application domains), different implementations of the algorithm will give the best attainable performance.

We briefly describe our case study problem, present four strategies for parallelizing it onto GPUs, and demonstrate that the selection of the best variant and the optimization parameters to use with that variant is nontrivial. We then describe our framework, ASP, that allows this variant selection and code generation process to be encapsulated in a way that is hidden from the domain expert. Without any variant selection, there is immediate performance gain of three orders of magnitude for realistic problems compared to executing the computation in pure Python. With even a simple variant selection algorithm, an average of 32% further performance improvement relative to always using a single baseline parallel code variant is possible, with the best-performing variant surpassing the performance of human-expert-authored C++/CUDA code. From the domain programmer's point of view, a one-line change to any Python program that uses an existing GMM library suffices to get these performance benefits.

2 Background: GMMs and the EM Algorithm

Suppose we are given audio of a conversation that is known to feature M distinct speakers. We could represent each speaker's speech characteristics with a probabilistic model. We could then attempt to model the conversation as a weighted combination of the M models, without knowing in advance which speaker made which utterances. This is the basic idea of a mixture model. To *train* a mixture model is to determine the parameters of each of the M submodels and the weights corresponding to each model in the mixture, such that we maximize the probability that the observed data (the audio track) corresponds to a prediction of the overall mixture model.

In the case of Gaussian mixture models (GMMs), each submodel is assumed to be represented by a D-dimensional Gaussian distribution of mean μ_i and $D \times D$ covariance matrix Σ_i. Given N observed data points, each a D-dimensional feature vector, we need to learn the parameters μ_i, Σ_i for each submodel and the weight parameters π_i for combining them into the overall mixture model. A common way to learn these parameters is to use the Expectation-Maximization (EM) algorithm [7]: given an initial estimate of the parameters, the E-step computes the expectation of the log-likelihood of the observations given those parameters, and the M-step in turn computes the parameters that maximize the expected log-likelihood of the observation data. These two steps repeat until a convergence criterion is reached.

Our specializer emits parallelized code for all substeps of the EM algorithm. We apply additional effort to the most compute-intensive part of the algorithm, which occurs in the M-step when we compute the $D \times D$ covariance matrix Σ for each of the M clusters. As described in [17], the covariance matrix is the sum of the outer products of the difference between the observation vectors and the cluster's mean vector computed in this iteration:

$$\Sigma_i^{(k+1)} = \frac{\sum_{j=1}^{N}(p_{i,j}(x_j - \mu_i^{(k+1)})(x_j - \mu_i^{(k+1)})^T)}{\sum_{j=1}^{N} p_{i,j}} \quad (1)$$

where $p_{i,j}$ is the probability of point j belonging to cluster i and x_i is the observation vector.

On our GPUs, the covariance matrix computation accounted for 50–60% of the overall runtime of the EM algorithm. There are multiple ways that this computation can be mapped onto parallel hardware.

3 Benefits of Code Variant Selection

The covariance computation exhibits a large amount of parallelism due to the mutual independence of each cluster's covariance matrix, each cell in a covariance matrix, and each observation's contribution to a cell in a covariance matrix. These three possible degrees of freedom in data parallelism suggest different strategies for parallelizing the algorithm on manylane hardware. Indeed, the optimal strategy depends on the problem parameters (N, D, M) as well as certain hardware parameters (e.g. number of cores, SIMD vector width, local memory size). Figure 1 summarizes the four code variants described below for a problem size of $M = 2$, $D = 4$ and $N = 7$.

We use the platform-neutral OpenCL [13] terminology to describe our strategies, which are implemented in NVIDIA's CUDA language [15]. There are two levels of parallelism: workgroups are parallelized across cores on the chip, and a workgroup's work-items are executed on a single core, potentially utilizing that core's SIMD vector unit. Each core has a scratchpad memory, referred to as a local memory.

Code Variant 1 (V1)—baseline: The EM on CUDA implementation from Pangborn [17]. Launches $M \times D \times \frac{D}{2}$ workgroups. Each workgroup is responsible for *one* cell in the covariance matrix for *one* cluster. Work-items correspond to events (N). The mean vector is stored in local memory, however only two values are used (corresponding to the row and column of the cell the group is computing).

Code Variant 2 (V2): Modifies V1 by assigning each workgroup to compute *one* cell for *all* clusters. Work-items correspond to events as in V1.

Code Variant 3 (V3): Makes better use of per-core memory by assigning each workgroup to compute the *entire* covariance matrix for *one* cluster (M). Each work-item in the workgroup is responsible for one cell in the covariance matrix ($D \times \frac{D}{2}$ items). Each work-item loops through all events sequentially.

Code Variant 4 (V4-BXX): Improves upon V3 by making it more resilient to small M by adding blocking across the N dimension. Launches $M \times B$ workgroups, where B is a blocking factor, i.e. the number of desired event blocks. Each workgroup computes the contribution to its entire covariance matrix for its block of events ($\frac{N}{B}$). In this paper we use two values of B (32,128).

We test all the variants on NVIDIA GTX285 and GTX480 GPUs using a regular sampling of problem sizes to gain an understanding of the tradeoffs amongst the variants. The GTX285 has more cores, whereas the GTX480 has longer SIMD vectors and better atomic primitives. Figure 2 shows some example results. Overall, for the space of problem sizes

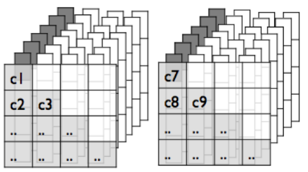
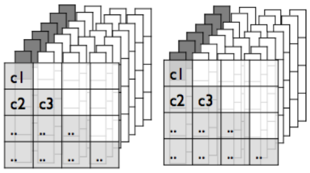
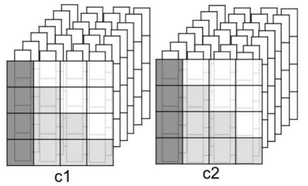
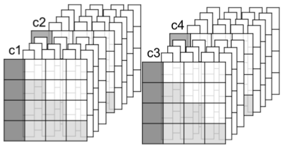
CV:	1	2	3	4
				
Work-items	All events (N total)	All events (N total)	All cells in a covariance matrix (D x D/2 total)	All cells in a covariance matrix (D x D/2 total)
Work-groups	One cell in covariance matrix for one cluster (M x D x D/2 total)	One cell in covariance matrix for all clusters (D x D/2 total)	One covariance matrix for one cluster (M total)	Partial sum for covariance matrix for one cluster (M x B total)
Local Mem	Means for core's cluster	Events (if small), Means for all clusters	Means for core's cluster, Covariance matrix	Means for core's cluster, Partial covariance matrix

Figure 1: Four code variants for computing the covariance matrix during M step. c* refers to different work-groups, darker shading refers to work-items. Each row of the table indicates the units of work (total number of such units) exploited by each level of hardware parallelism for each variant. The last row indicates what objects should be placed in each core's local memory for the variant to be effective.

we examined ($1 \leq D \leq 36, 1 \leq M \leq 128, 10,000 \leq N \leq 150,000$), the best-performing code variant for a given problem instance gave a 32% average performance improvement in covariance matrix calculation time compared to always running the baseline code variant V1. This performance gap increases further with larger problem sizes, e.g. for ($D = 36$, $M = 128$, $N = 500,000$) the difference grows to 75.6%. Figure 2 plots a slice through the 3D space of possible input parameters, allowing the average runtimes of different implementations of the covariance computation to be compared.

V1, V3 and V4 with different B parameters are mutually competitive and show trade-offs in performance when run on various problem sizes and on two GPU architectures. V2 shows consistently poor performance compared to the other code variants. The tradeoff points are different on the two GPUs. While there are general trends leading to separable areas where one variant dominates the others (e.g. V1 is best with small D values), we had difficulty formulating a hierarchy of rules to predetermine the optimal variant because each hardware feature affects each variant's performance differently. This finding suggests that variant selection cannot necessarily be reduced to a compact set of simple rules, even for a specific problem in a restricted domain of problem sizes.

4 SEJITS for GMMs

We have shown that the EM algorithm can be implemented in multiple ways, and that the best implementation is dependent on both the specifics of the target hardware and the problem dimensions. What is needed is a way to encapsulate these variations and dependencies such that the domain application developer does not have to reason about them, and equally importantly, so that the code variant selection (*how* to do the computation) can be kept separate from the actual application (*what* to do), allowing the application to be performance-portable [6].

We chose Selective Embedded Just-in-Time Specialization (SEJITS) [8] as the mechanism to accomplish this separation of concerns. In the SEJITS approach, the domain programmer expresses her application entirely in Python using libraries of domain-appropriate abstractions, in this case objects representing GMMs and functions that can be invoked on them, e.g. training via the EM algorithm. However, when these speciaized functions are called, they do not execute the computations directly; instead, the ASP framework interposes itself and generates CUDA source code, which is then JIT-compiled, cached, dynamically linked, and executed via the GPU's foreign-function interface, with the results eventually returned to Python. From the Python programmer's view, this experience is like calling a Python library, except that performance is potentially several orders of magnitude faster than calling a pure Python library implementation of the same computation.

ASP [1] is a particular implementation of the SEJITS approach for Python[1]. ASP contains facilities to automate the process of determining the best variant to use, emit source code corresponding to that variant, compile and call the optimized code, and pass the results of the computation back to the Python interpreter. A *specializer* is a piece of Python code that uses these facilities in combination with code templates expressing the different code variants hand-coded by a human expert, in our case a CUDA programmer who serves a similar role to a library developer. Our specializer implements the EM algorithm described previously, but ASP's facilities are general and could be used for very different specializers; indeed, it has previously been used to specialize stencil/structured-grid codes on multicore CPUs [1, 8].

Once ASP has been imported into a Python program, it transparently and selectively redirect calls of certain functions (in our case, GMM functions in the *scikit* [5] library API) to the appropriate specializer. From the domain expert's point of view, a one-line change to any of her Python programs that use the existing GMM library suffices to get the performance benefits we reported in the previous section.

[1] Recursive acronym: ASP is SEJITS for Python

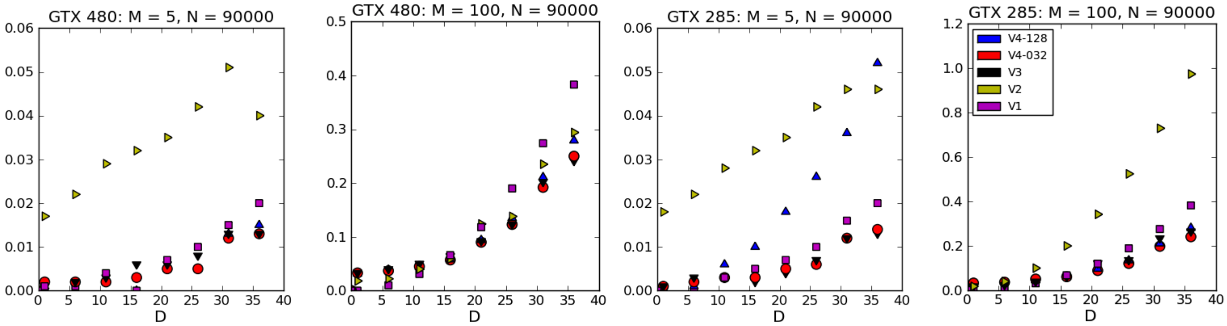


Figure 2: Average runtimes in seconds of covariance computation variants on GTX480 (left) and GTX285 (right) across D, for $N = 90000$ and $M = 5$ or $M = 100$. Different markers signify different variants' runtimes. The best variant depends on both the problem size and the underlying machine. V1 and V2 have a large amount of work-item parallelism while V2 has limited work-group parallelism if D is small. V3 can be limited in both work-group and work-item parallelism if M and D are small. V1 underutilizes the local memory and requires many streaming reads, whereas V2 and V3 utilize local memory more efficiently and V2 requires a factor of M fewer event data streaming reads than V1. V4 mitigates V3's limited work-group parallelism by blocking over N, but requires a global reduction across B work-groups.

Our GMM specializer actually emits two kinds of lower-level source code: C++ host code that implements the outer (convergence) loop of EM training, and CUDA GPU code that implements the different parallel kernels used during each EM substep. Code variant selection occurs for one particular kernel of the training process — covariance matrix creation in the M-step. We hand-coded templates for the variants described previously; ASP transforms these templates into syntactically correct source code via the templating library Mako [3], and compiles, caches and links them using the Python extension CodePy [2]. The templates contain placeholders that are filled in by our specializer, such as blocking factors and limits on the number of main loop iterations.

It should therefore be clear that the specializer has two jobs: variant selection and intra-variant code optimizations.

Variant selection. Part of our specializer is the variant selection logic. As explained earlier, the best variant to use depends on properties of the input problem data (the size of M, N, and D). We accomplish this by telling ASP to examine the values of the parameters passed to the specialized function, and to treat functions with different parameters values as different functions. The current ASP implementation tries a different variant every time a particular function/problem size is specialized, until all variants have been tested. Thereafter, the specializer remembers what the best runtime was for that particular function/problem size, and always reuses the associated variant's cached binary. This variant selection method is naive; future work will attempt to use performance models or machine learning to make decisions about what version of the code to use without exhaustively searching all options. However, the important observation for the present work is that the mechanism and policy for variant selection are well-encapsulated and can be replaced without touching the original application source code or the code variant templates themselves.

Intra-variant performance optimizations. In modern systems, GPUs and CPUs have separate memories, and the programmer is resposible for copying any necessary data and results back and forth between them. Since a common use case is to use the same dataset to train many GMMs (each with a different number of mixtures M), flushing and recopying that data (hundreds of megabytes) would be wasteful. Therefore, our specializer tracks whether the data stored on the GPU by one GMM module is being reused by a second, and only lazily flushes the data and replaces it. This is another example of an optimization that the application programmer need not concern herself with. A similar optimization is the use of the PyUBLAS Python extension [4], which allows numerical data structures to be shared by reference between C++ and Python, eliminating unnecessary value copies.

Note that the logic implementing the above optimizations is itself written in Python. It is not only kept separate from the low-level computation kernels and the high-level application code, but also easier to modify and experiment with since it can exploit the full facilities of the Python language.

5 Results

There are two metrics by which our framework should be evaluated: how productive it is compared to a pure Python library, and how efficient it is compared to a pure C++/CUDA implementation. We measure the former by implementing applications from the scikit.learn [5] library's example code package on top of our specializer: because we implement the same API as the existing library, porting these applications to use our specializer requires only a single line code change.

To measure the overhead relative to a native C++/CUDA application, we implement *Agglomerative Hierarchical Clustering* (AHC), an unsupervised training algorithm for speaker clustering in a speaker diarization application that uses GMMs to represent different speakers. AHC iteratively performs GMM training many times, using a different number of target clusters each time and measuring which clustering is the best. The number of clusters in the "best" GMM corresponds to the number of speakers in the meeting.

We compare the performance overhead of implementing the AHC application loop in Python with specialized GMM

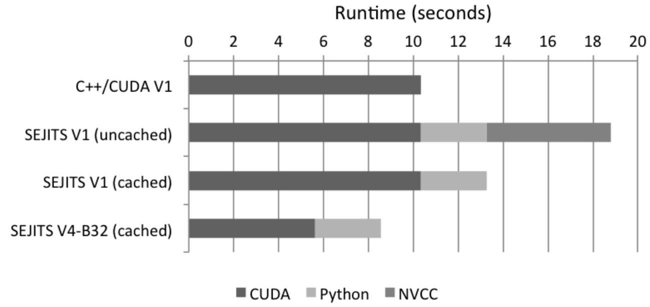


Figure 3: Runtime in seconds of the original C++/CUDA implementation of the AHC application compared to our Python/SEJITS version at various stages of the specialization process.

training, as compared to implementing AHC in C++/CUDA. We determine that for a speaker diarazation data set (D: 19, N: 150000, M: 16 to 1), while there is intially a 81% overhead due to compiler invocations, future runs of the application using the automatically determined optimal code variant actually achieve a 17% performance *improvement* over the original GPU implementation (V1).

Because ASP currently has a naive code variant selection strategy, it invokes the compiler repeatedly to test the performance of different variants on a particular problem size. However, the results of this process are cached, and future calls to the specializer do not have to invoke the compiler.

Figure 3 compares the performance of the original C++/CUDA code to our Python/SEJITS implementation. We measure both the runtime of the SEJITS version when the best version is not cached (which results in compiler invocations) and subsequent calls (which do not). Once ASP has selected the optimal version of the algorithm to use (V4-B32 in this case), its performance actually surpasses the C++/CUDA code. The benefit of using the best code variant outweighs the overhead of implementing the AHC application in Python.

6 Related Work

Our code kernels build directly on previous efforts to accelerate the EM algorithm using GPUs [14, 17], but we view our contribution as the methodology and framework for separating such concerns from the application programmer. The general concept of providing portable, high performance abstractions for this purpose was pioneered by autotuning libraries such as FFTW [11], Spiral [18], and OSKI [19]. These libraries specialize individual function calls, but are not embedded into a host language, and their auto-tuning machinery does not generalize to other use cases. In contrast, ASP allows the domain programmer to stay in the host language (Python) and variant selection can occur at runtime using a general set of just-in-time code generation techniques applicable to a wide variety of computational kernels.

SEJITS inherits standard advantages of JIT compilation, such as the ability to tailor generated code for particular argument values and sizes. While conventional JIT compilers such as HotSpot [16] also make runtime decisions about what to specialize, SEJITS does not require any additional mechanism for dealing with non-specialized code, which is just executed in Python. Even specialized functions can fall back on a Python-only implementation if the specializer is targeted for different hardware than is available at runtime. Another example of the SEJITS approach is Delite [9], which automatically maps Domain-Specific Embedded Languages [12] implemented in Scala onto parallel hardware platforms.

Cython [20] translates annotated Python code into more efficient C code, but has no support for platform retargeting and requires changes to application logic. In contrast, ASP requires a one-line change to a Python application to “hijack” certain Python function calls and redirect them to an appropriate specializer.

7 Conclusion & Future Work

The ASP framework encapsulates and facilitates reuse of two important kinds of resources: handcrafted code *templates* embodying particular parallel execution strategies and *variant-selection heuristics* for choosing the best template at runtime based on hardware configuration and problem instance characteristics. Both kinds of resources are often beyond the expertise of domain-expert programmers to create; encapsulating and hiding the mechanisms necessary to reuse them allows domain experts to focus on productivity without sacrificing the performance advantages of expert-tuned code. In particular, changing a single line in the domain expert’s Python code triggers our specialization framework. This small “productivity footprint” exploits metaprogramming and reflection in Python, and could just as well be adapted to other modern languages possessing these facilities such as Ruby.

Our current prototype uses a naive variant selection process, yet also provides performance competitive with handcrafted CUDA code. Indeed, we found that the performance benefit of specialization outweighed the overhead of Python and the JIT process, allowing amortization of the selection overhead. Since we have separated out variant selection as a first-class concern, the same high-level program can be transparently retargeted to other platforms such as multicore CPUs or the cloud, making the source code performance-portable. We are working on code templates for these cases. We also intend to develop a globally-accessible history database that “remembers” the performance associated with particular problem instances on a given hardware platform and suggests code variant choices and tuning parameters for problems similar to ones previously solved on similar hardware.

As hardware platforms become more varied and the space of tuning parameters becomes more complex, productive programming will require segregating application development, code variant selection and tuning parameter selection, so that experts in the respective areas can address each of the concerns independently. Our ASP framework, as a concrete implementation of SEJITS (Selective Embedded Just-in-Time Specialization) is a first step in that direction.

References

[1] ASP: A SEJITS implementation for python. http://aspsejits.pbworks.com/.

[2] CodePy: a C/C++ metaprogramming toolkit for python. http://mathema.tician.de/software/codepy.

[3] Mako templates for python. http://www.makotemplates.org/.

[4] PyUBLAS: a seamless glue layer between numpy and boost.ublas. http://mathema.tician.de/software/pyublas.

[5] scikits.learn: machine learning in python. http://scikit-learn.sourceforge.net/index.html.

[6] B. Alpern, L. Carter, and J. Ferrante. Space-limited procedures: a methodology for portable high-performance. In *Proceedings of the conference on Programming Models for Massively Parallel Computers*, PMMP '95, pages 10–, Washington, DC, USA, 1995. IEEE Computer Society.

[7] C. M. Bishop. *Neural Networks for Pattern Recognition*. Oxford Univ. Press, New York, 1995.

[8] B. Catanzaro, S. Kamil, Y. Lee, K. Asanović, J. Demmel, K. Keutzer, J. Shalf, K. Yelick, and A. Fox. SEJITS: Getting productivity and performance with selective embedded JIT specialization. In *Workshop on Programming Models for Emerging Architectures (PMEA 2009)*, Raleigh, NC, October 2009.

[9] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun. A domain-specific approach to heterogeneous parallelism. In *Proceedings of the 16th Annual Symposium on Principles and Practice of Parallel Programming*, February 2011.

[10] J. Demmel, J. Dongarra, A. Fox, and S.Williams. Accelerating time-to-solution for computational science and engineering. *SciDAC Review*, (15), 2009.

[11] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[12] P. Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28, December 1996.

[13] Khronos Group. *OpenCL 1.1 Specification*, September 2010. Version 1.1.

[14] N. S. L. P. Kumar, S. Satoor, and I. Buck. Fast parallel expectation maximization for gaussian mixture models on gpus using cuda. In *Proceedings of the 2009 11th IEEE International Conference on High Performance Computing and Communications*, pages 103–109, Washington, DC, USA, 2009. IEEE Computer Society.

[15] NVIDIA Corporation. *NVIDIA CUDA Programming Guide*, March 2010. Version 3.1.

[16] M. Paleczny, C. Vick, and C. Click. The java hotspot(tm) server compiler. In *JVM'01: Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium*, pages 1–1, Berkeley, CA, USA, 2001. USENIX Association.

[17] A. D. Pangborn. Scalable data clustering using gpus. Master's thesis, Rochester Institute of Technology, 2010.

[18] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[19] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16(1):521+, 2005.

[20] I. Wilbers, H. P. Langtangen, and Å. Ødegård. Using cython to speed up numerical python programs. In B. Skallerud and H. I. Andersson, editors, *Proceedings of MekIT'09*, pages 495–512. NTNU, Tapir, 2009.

[21] S. W. Williams. *Auto-tuning Performance on Multicore Computers*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008.