Doubling the Performance of Python/NumPy with less than 100 SLOC

Simon A. F. Lund, Kenneth Skovhede, Mads R. B. Kristensen, and Brian Vinter
Niels Bohr Institute, University of Copenhagen, Denmark
{safl/skovhede/madsbk/vinter}@nbi.dk

***Abstract*—A very simple, and outside NumPy, commonly used trick of buffer-reuse is introduced to the NumPy library to speed up the performance of scientific applications in Python/NumPy. The implementation, which we name software victim-caching, is very simple. The code itself consists of less than 100 lines of code, and took less than one day to add to NumPy, though it should be noted that the programmer was very familiar with the inner workings of NumPy. The result is an improvement of as much as 2.29 times speedup, on average 1.32 times speedup across a benchmark suite of 15 applications, and at no time did the modification perform worse than unmodified NumPy.**

I. Introduction

Python/NumPy is gaining momentum in high performance computing, often as a glue language between high performance libraries, but increasingly also with all or parts of the functionality written directly in Python/NumPy. Python/NumPy represents an easy transition from Matlab prototypes, to the extent where we observe scientists working directly in Python/NumPy since their productivity is as high as in Matlab. While Python/NumPy is still not as efficient as C++ or Fortran, which are the more common HPC languages, the productivity of the higher-level language often becomes the choice of the programmer. As a rule of thumb, we expect Python/NumPy to be approximately four to five times slower than C, and the balance in choosing a programming language is thus often a balance between faster programming or faster execution and is stands to reason that, as Python/NumPy solutions close the performance gap to compiled languages, the higher productivity language will gain further traction. In our work to improve the performance of NumPy[1] we came across a behavior which we initially attributed to our work on cache optimizations, turned out to be the effects of a far simpler scheme where by temporary array allocations in NumPy are more efficiently reused.

The amount of memory that is reserved for buffer-space is naturally defined by the user through a standard environment variable. In this work, we experiment with three fixed buffer-sizes 100, 512 and 1024 mega bytes. Programmers can experiment with different buffer-sizes, however, very large buffers rarely make an impact.

The resulting changes to NumPy, less than 100 lines in total, counted using SLOCCount[2], provides advantages over conventional NumPy from none, but never worse, to 2.29 times speedup. Our suite of 15 benchmarks has an average speedup of 1.32, and thus, with no requirement to the application programmer closes the gap to compiled languages a little further.

The rest of this paper is comprised as follows; related work since this is not a new idea outside Python, a section on the implementation details, then the benchmarks are introduced and results are presented.

II. Related Work

In computer architecture, a victim-cache is a small fully-associative cache where any evicted cache-line is stored and thus granted an extra chance for remaining in the cache, before being finally evicted[3]. At the CPU level victim-caching is particularly efficient at masking cache-line tag conflicts. Since NumPy does not have any cache, the victim cache may appear unrelated, but the idea of a fully associative cache that holds buffers a little while until they are fully evicted, is very similar.

In functional languages a similar buffer reuse scheme, copy collections, is found efficient in [4]. In this work, the buffer is very large, and numerous techniques for buffer location and replacement are considered; most of this is similar to page replacement algorithms at the operating system level.

Keeping control of buffers in relational databases is fairly closely related to maintain NumPy buffers, since relational databases also have a high locality of similar sized buffers[5] but dissimilar to NumPy, the space available for buffers is very high, and a more advanced replacement algorithm is needed since databases are multiuser systems, and the buffer patterns is thus less simple than what we can observe in NumPy.

Even though the victim cache technique itself is not related to garbage collection, the idea of memory reuse is very similar. Within a runtime with managed garbage collection, memory allocations are pooled to avoid repeated requests to the operating system[6]. While this is useful for repeated small allocations, most implementations assume that large allocations will stay in memory.

III. Software Victim-Caching

We have dubbed the adopted technique software victim-caching, since the basic functionality is very similar to victim-caching as it is known in computer architecture. The idea is very simple; when NumPy releases an array we do not release the memory immediately, but keep the buffer in a victim-cache, when NumPy issues a new array allocation we first do a lookup in the victim-cache, and if a matching array is found, it is returned rather than a new array allocation.

Different matching and eviction algorithms have been experimented with, see section III-C for further details. Note that only full allocations are returned from the victim-cache, we do not try to use partial arrays or merge arrays to find a match, or in any other way attempt conventional heap management.

The logic behind this very naÃ¯ve approach is fairly simple as well; scientific applications are most often comprised of dense loops any temporary array allocation is due to this very likely to be observed again very soon after being released. In addition, the temporary arrays that are allocated for different operations on the same user defined array are likely to be of identical dimensions as well.

A. Temporary Arrays

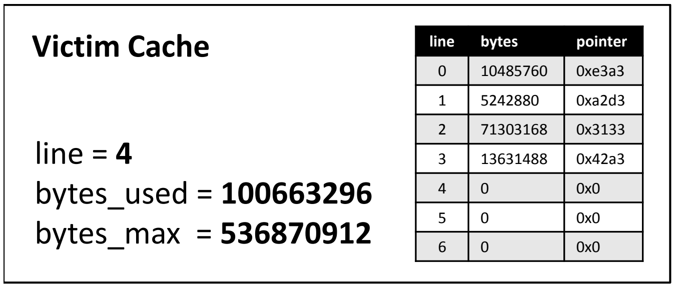
Temporary arrays are instantiated by NumPy whenever an intermediate result is needed. The general case is; an expression consisting of more than a single operator and thereby creating a complex expression. As an example, assume we wish to calculate the distance from $(0, 0)$ for a set of (X, Y) coordinates in NumPy we write:

```
distance = numpy.sqrt(X**2 + Y**2)
```

This operation will create three temporary arrays, plus a non temporary which is returned to distance, the X^2, Y^2, and $+$ operations will each allocate a temporary array which is discarded after the line is executed, the square root operation also allocates an array, which is returned to the distance array. In this case, the first two temporary arrays are released once the fourth allocation is called, and one of the first two could be used since they match the allocation perfectly.

B. Implementation

The implementation is interface-compatible with malloc. This allows for a very low-intrusion integration by only changing 10 lines of code in the NumPy codebase. The implementation of the victim cache itself, including all matching and eviction strategies mention in section III-C, is a total of 237 lines of code, where the simple version with only one strategy is 81 lines. Figure 1 illustrates the data-structures, which are maintained.



Victim Cache

line = **4**
bytes_used = **100663296**
bytes_max = **536870912**

line	bytes	pointer
0	10485760	0xe3a3
1	5242880	0xa2d3
2	71303168	0x3133
3	13631488	0x42a3
4	0	0x0
5	0	0x0
6	0	0x0

Fig. 1: Illustration of the victim-cache data-structures for a victim-cache with seven cache-lines, a maximum size of 512MB and currently populated with four entries consuming 96MB.

The simplest implementation maintains the currently consumed *bytes_used* of the victim-cache, the *bytes_max* maximum number of bytes allowed for consumption, and the currently used *line* in the victim-cache. The following section describes different strategies of using the victim-cache. The implementation is available as a github-fork[1] of NumPy 1.7 on the branches *victim_cache* and *victim_cache_clean*. The branch *victim_cache* contains the implementation featuring the multiple algorithms which are described in the following section. The branch *victim_cache_clean* contains the cleaned up, less than 100 source-lines of code, implementation featuring a single strategy and the possibility of enabling/disabling the victim cache via environment-options.

C. Algorithms

Buffer management algorithms are a well researched area[7]. However, for many scenarios a simple solution is as good, or better, than advanced adaptive algorithms. For that reason, we limit the experiments in this work to six well-known algorithms. Three for matching buffers to requirements and three for selecting a buffer to eliminate when the allocated buffer space is saturated.

For matching buffers, we use three very simple algorithms, Exact, First, and Best. Exact will only return a fit if the requested buffer-size is exactly the same size as the buffer in the victim cache. First will return the first of the tested buffers large enough to hold the requested buffer. Best will search all buffers in the victim-cache and return the buffer that is as large as the requested size, and with as little extra space as possible. If an exact match is found, it is returned immediately.

If the maximum allowed buffer size would be exceeded by adding an allocation, an existing buffer in the cache must be evicted. Choosing one can also be done in many ways including Round-Robin, Second-Chance, and Random. Round-Robin will evict buffers from the victim-cache in the order they are added. If a buffer is selected for reuse, it will be added to the end once it is released again. It could also be described as evicting the oldest cache-line first. Second-chance is well known from demand-paging in operating systems and is aptly named. If a buffer is next to be evicted it will be marked at ready-to-evict, but the algorithm will in-fact move on in the list, only if a buffer is revisited, i.e. when a buffer is marked as ready and is next to be evicted, will it actually be selected for eviction. The worst case scenario is that all buffers must be visited once before one can be chosen for eviction, but in reality is will be more like round-robin, but with a second chance for some buffers. Random selection is extremely simple; a random buffer in the list is selected for eviction, while this may appear as a strange choice this approach has the advantage that it will not fall into a pattern where the same set of buffers are continuously evicted.

As the applications that NumPy is commonly used for, are highly regular in their execution pattern, we expect the simplest algorithms to perform very well, i.e. Exact-fit for matching and round-robin for eviction. If this is the case, there is no reason to keep the more advanced algorithms in a final version and the codebase can be kept very small indeed.

IV. Comparison

To evaluate the performance of the victim-cache, we have chosen 15 different benchmarks, that use a broad range of NumPy functionality. We have chosen some benchmarks that

[1] http://github.com/cphhpc/numpy/

operate on one-dimensional arrays and perform typical Monte Carlo simulations. For two-dimensional benchmarks, we use a selection of classic physics based computational kernels, for higher dimensions, we use a 3D Lattice-Boltzmann simulation and an n-body simulation. To show that the approach is also valid in other scenarios, we also use naïve implementations of FFT, LU, and matrix multiplication. The source-code for the benchmarks are available for closer inspection in the github-fork [2] on the *victim_cache* branch in the *benchmark/Python/* folder.

A. One-dimensional benchmarks

For testing with one-dimensional applications, we have chosen three different Monte Carlo based implementations. The simplest version is the original Monte Carlo Pi simulation that derives the value of π through a series of simulated dart throws. The other two benchmarks are taken from financial analysis domain and attempt to price a set of stock options, using the Black-Scholes model for European pricing and swaptions in the LIBOR market model, respectively. The Monte Carlo Pi simulation generates only a few temporary arrays in each iteration, whereas the Black-Scholes implementation generates as much as 67 temporary arrays in an iteration. The number of temporary arrays generated by the Swaption implementation varies with the input data and the amount of elements in each temporary array is relatively small.

B. Two-dimensional benchmarks

For two-dimensional applications we have chosen a common Jacobi five-point stencil application, a successive over relaxation (SOR), a shallow water simulation, a WireWorld simulation, a Lattice-Boltzmann simulation and a cloth physics simulation. The Jacobi stencil is chosen for its simplicity, where the others are chosen because they are larger applications, which would be hard to optimize by hand. The SOR simulation essentially does the same as the Jacobi stencil, but implemented with a red/black update scheme, and includes a global delta calculation. The Lattice-Boltzmann, shallow water, and cloth simulations all simulate movement in a two-dimensional space with different models for force propagation. The Jacobi stencil code is fairly compact but still generates 9 temporary arrays in each iteration. The other benchmarks generate a larger number of temporary arrays that are candidates for optimization from the victim cache.

C. Higher-dimensional benchmarks

To show the effects of the victim-cache with problems that have multiple dimensions, we have chosen a some classic computational kernels, namely a naïve n-body simulation, a k-nearest-neighbor search, and a Lattice-Boltzmann simulation in 3D space. The k-nearest-neighbors search has a low amount of temporary arrays, and the n-body simulation and Lattice-Boltzmann simulations have a moderate amount of temporary arrays.

Benchmark	Problemsize	Iterations
Black Scholes	$8 \cdot 10^6$	5
Bolzmann 3D	$120 \times 100 \times 100$	5
Bolzmann D2Q9	800×800	5
Cloth	3000×3000	1
FFT	18	N/A
Jacobi Stencil	$10000 \times 4000 \times 10$	10
KNN	$2 \cdot 10^6 \times 10$	3
LU Factor.	500×500	N/A
Matrix Mul	800	N/A
Monte Carlo PI	$2 \cdot 10^7$	10
NBody	3000×1	1
Shallow Water	3000×3000	5
SOR	4000×4000	5
Swaption	1000	N/A
Wire World	5000×5000	5

TABLE I: Overview of benchmarks and problem sizes.

D. Kernel benchmarks

To broaden the experiment we have chosen a set of kernels that are traditionally implemented in external libraries and implemented them in NumPy. The kernels comprise naïve versions of matrix multiplication, LU factorization, and Fast Fourier Transformation (FFT). The FFT kernel generates a low amount of temporary arrays. The where the matrix multiplication and LU kernels generate a large amount of temporary arrays, where the arrays generated by the LU kernel are small, and the ones generated by the matrix multiplication are large.

Table I provides the full list of benchmarks along with the parameters for their execution.

E. Results

As the victim cache mitigates the work related to allocating array memory from the operating system, there is a clear relation between the number of temporary arrays and the gained speedup. Figure 2 shows the speedups obtained from running the same NumPy code with three different sizes of the victim cache. Each benchmark has been set up with parameters that cause the benchmarks to run around 10 seconds with no victim cache. Each benchmark is then executed with the same input data, and varying sizes of the victim cache and the wall-clock times are used to compute the speedup. All benchmarks are executed on a AMD Opteron 6272 CPU with 128 GB of memory, running with Ubuntu 12.04.2 LTS. The execution times were stable with a maximum wall-clock time deviation of 0.08 seconds.

We can see that some experiments gain no speedup at all, but none of the experiments show any slow down from the victim cache. For our problem sizes, a moderate size victim cache of 512 MB is sufficient to gain the maximum performance speed up, except for the Jacobi example, which shows a large speedup when utilizing 1GB of victim cache memory.

The SOR, shallow water, and Jacobi benchmarks show as much as 2.3 times speedup from using the victim cache, which we consider a significant result. The following section provides further analysis of the results.

F. Analysis

The results are quite convincing, while a few benchmarks do not show any improvement in performance most do, and

[2] http://github.com/cphhpc/numpy/

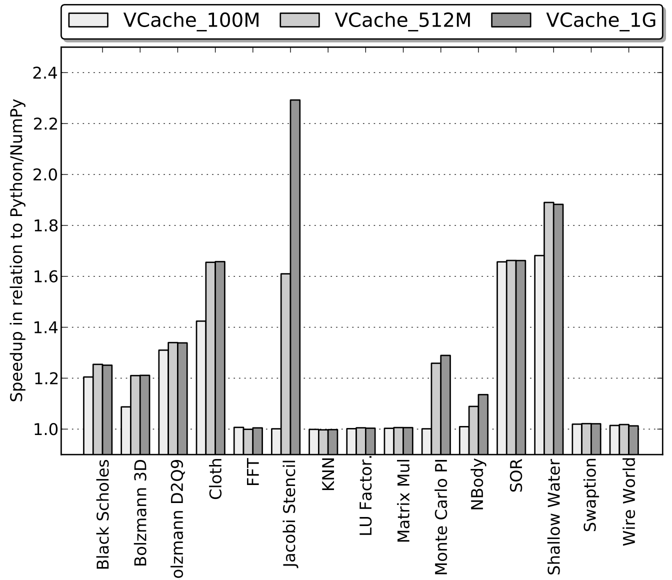


Fig. 2: Speedup of victim-cache in relation to unmodified Python/NumPy of the 15 benchmark applications.

no less than four show a performance increase of more than 50%, two of more then 100%. At a first glance, it is hard to see why a simple victim-cache can improve performance that much. The explanation can be found in the way glibc under Linux handles memory allocations. Any allocation to an area that is larger than 128KB[3], is allocated using mmap rather than sbrk[8]. The consequence is that memory that is allocated with mmap, will be released to the operating system, when free is called. This means that the memory is actually returned to Linux as opposed to memory that is allocated with sbrk, which is kept for reuse by glibc. The consequence of this is that the many, large, temporary array allocations in NumPy are moved back and forth between user space and kernel space. The actual call to the operating system represents an overhead in itself, but the majority of the time is spent zeroing the memory to stop information from leaking between processes. Thus the big advantage of the victim cache model it that we save a write to the temporary memory, which in effect doubles the cost of simple array operations.

To verify that the above description is in fact the reason for the observed performance improvements, the benchmarks were repeated using the time tool in order measure time spend in user-level and kernel-level respectively. Figure 3 show the time spent in kernel-level for each benchmark, for standard NumPy (Native), and the victim-cache implementation for three different cache sizes. In this figure, lower means less time spent in system.

There is an obvious correlation between the benchmark where we observed improvement in the overall runtime, and the drop in time spent in kernel-level. Moreover, going from 512MB to 1GB of victim-cache only shows a significant impact in the Jacobi Stencil benchmark where time spent in kernel-level is reduced by more than half. A manual experiment to increase the victim-cache to 2GB showed no further improvement in runtime.

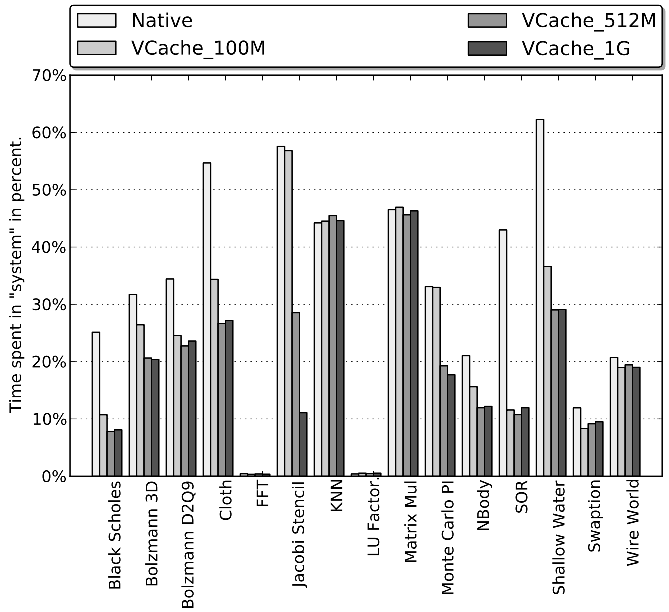


Fig. 3: Time spent in system/kernel-level reported in percentage of total wall-clock time.

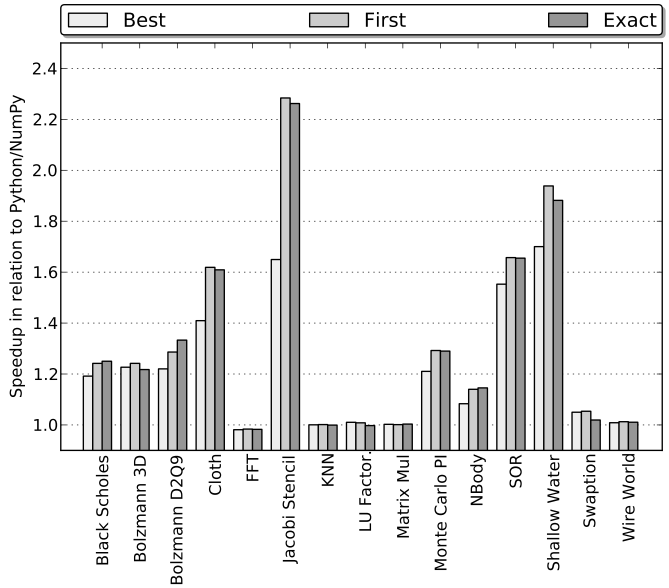


Fig. 4: Comparison of fitting strategies.

We go on experimenting with the three different matching algorithms, best-fit, first-fit, and exact-fit. Best-fit requires an inspection of each element of the cache for each victim-cache lookup, the consequence of which is clearly shown in figure 4. The difference between first-fit and exact-fit is marginal and varies across the benchmark suite.

Figure 5 illustrates the results of experiments with different eviction strategies. The random eviction-scheme, which is known to work well for page-replacement in the operating system, clearly does not apply for the victim-cache. This result was expected since most benchmarks demonstrate a high degree of regularity. Second change and oldest first are

[3] by default but may be changed by the user

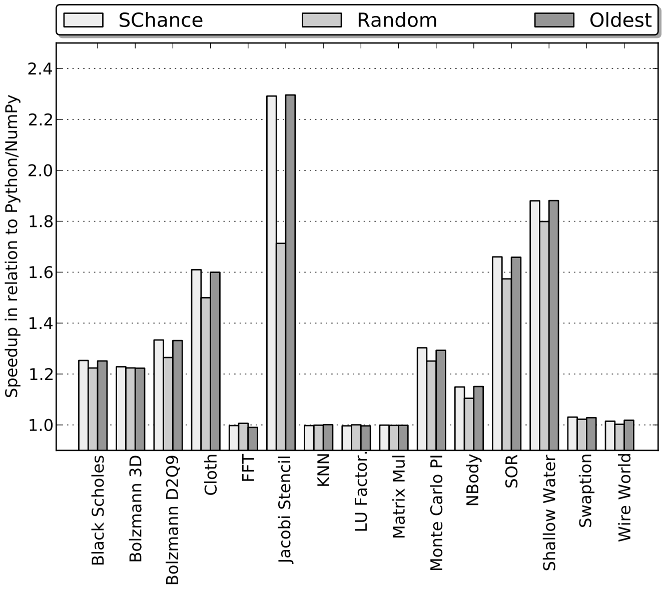


Fig. 5: Comparison of eviction strategies.

indistinguishable.

Given the interface compatibility and the previous description of malloc, a valid question is: ”why not just parameterize malloc correctly?”. Because the zeroing memory only affects allocations above `MMAP_THRESHOLD` bytes, one could consider simply increasing `MMAP_THRESHOLD`. However, glibc malloc does not allow for `MMAP_THRESHOLD` to go beyond `MMAP_THRESHOLD_MAX`, which on 64-bit systems is only 4MB. Changing the maximum value would require recompiling glibc and is thus a fairly intrusive procedure. A more drastic approach would be to re-compile the Linux kernel with `__GFP_ZERO` defined to zero, which would completely eliminate the zeroing of pages across the operating system.

V. Future Work

The solution itself is quite simple, but the implementation may be subjected to further refinements. In the current implementation, a rather simple list-based lookup search is performed, yield a runtime complexity of $O(n)$ over the number of entries. Many better strategies exist, such as tree-based lookups that can reduce this overhead.

The results presented in this article show that there is indeed an overhead involved in the generation of temporary arrays and that a victim cache can reduce the overhead by using some extra memory. However, a more thorough approach is to avoid creating the temporary arrays completely.

However, unlike the victim cache, such a change requires changes in many places within the NumPy libraries. We are actively investigating this approach as part of the Bohrium runtime system[1].

We have produced a cleaned up version of the changes, which only supports exact matching and round-robin eviction. This cleaned up version is reduced to 81 lines of C-code, combined with the ten lines in the NumPy multiarraymodule brings the total lines of code in the victim-cache implementation up to 91. We plan to submit this patch to NumPy upstream developers so NumPy user can reap the benefits discovered.

VI. Conclusion

We have implemented a very simple and non-intrusive victim-cache in NumPy, and evaluated the effects on a variety of different benchmarks. The experiments clearly show that the victim-cache is able to reduce much of the overhead that occurs when NumPy allocates memory from the operating system. In no case did see an actual slowdown. Generally we see an average improvement of 32% accross the benchmark suite, if we cherry-pick only the benchmarks where we see an improvement the average speedup is 52%. The best observed speedup is 230% of the Jacobi Stencil benchmark. A victim-cache of 512MB was sufficient to harvest all the gains of victim-caching in all benchmarks except one, the Jacobi Stencil.

We experimented with three different matching strategies, and three different eviction strategies, however the high degree of regularity in the benchmark suite meant that the simplest algorithms exact-fit and oldest-eviction first performed as good as or better than the more advanced strategies.

Overall we conclude that the victim-cache is a nearly cost-free optimization, that potentially benefits more than half of all NumPy applications, without any requirements towards to programmer. The implementation comprises ten lines of changes to the NumPy multiarraymodule and 81 lines for the victim-cache itself, in total 91 lines of code.

Acknowledgments

The Cloth simulation is ported to Python from a Java based implementation written by Jared Counts[4]. The Black-Scholes simulation is vectorized based on an example from Espen Haug[5]. The 2D Lattice-Boltzmann simulation is based on code by Jonas Latt[6]. The 3D Lattice-Boltzmann simulation is based on code written by Iain Haslam[7]. The Swaption pricing in LIBOR market is written by Lykke Rasmussen[8]. The LU factorization is based on code in the SciMark2 suite.

This research has been partially supported by the Danish Strategic Research Council, Program Committee for Strategic Growth Technologies, for the research center 'HIPERFIT: Functional High Performance Computing for Financial Information Technology' (hiperfit.dk) under contract number 10-092299.

This research has been partially supported by the Danish Strategic Research Council, Program Committee for Strategic Growth Technologies, for the 'High Performance High Productivity' project under contract number 09-067060.

References

[1] M. Kristensen, S. Lund, K. Skovhede, T. Blum, and B. Vinter, “Bohrium: a virtual machine approach to portable parallelism,” *in submission to ICPADS13*, 2013.

[4] http://gamedev.tutsplus.com/tutorials/implementation/simulate-fabric-and-ragdolls-with-simple-verlet-integration/
[5] http://www.espenhaug.com/black_scholes.html
[6] http://wiki.palabos.org/_media/numerics:cylinder.pdf
[7] http://www.exolete.com/images/lbm3d.m
[8] http://quantess.net/

[2] D. A. Wheeler, "Sloccount, a set of tools for counting physical source lines of code (sloc)," *URL http://www. dwheeler. com/sloccount*, 2004.

[3] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Computer Architecture, 1990. Proceedings., 17th Annual International Symposium on*. IEEE, 1990, pp. 364–373.

[4] P. R. Wilson, "Some issues and strategies in heap management and memory hierarchies," *ACM SIGPLAN Notices*, vol. 26, no. 3, pp. 45–52, 1991.

[5] T. Lang, C. Wood, and E. B. Fernández, "Database buffer paging in virtual storage systems," *ACM Transactions on Database Systems (TODS)*, vol. 2, no. 4, pp. 339–351, 1977.

[6] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, "Dynamic storage allocation: A survey and critical review," in *Memory Management*. Springer, 1995, pp. 1–116.

[7] ——, "Dynamic storage allocation: A survey and critical review," in *Memory Management*. Springer, 1995, pp. 1–116.

[8] G. Insolvibile, "Advanced memory allocation," *Linux Journal*, vol. 2003, no. 109, p. 7, 2003.