Measuring the Haskell Gap

Leaf Petersen Todd A. Anderson Hai Liu Neal Glew
Intel Labs
{leaf.petersen,todd.a.anderson,hai.liu}@intel.com, aglew@acm.org

Abstract

Papers on functional language implementations frequently set the goal of achieving performance "comparable to C", and sometimes report results comparing benchmark results to concrete C implementations of the same problem. A key pair of questions for such comparisons is: what C program to compare to, and what C compiler to compare with? In a 2012 paper, Satish et al [9] compare naive serial C implementations of a range of throughput-oriented benchmarks to best-optimized implementations parallelized on a six-core machine and demonstrate an average $23\times$ (up to $53\times$) speedup. Further, they demonstrate that most of this so-called "Ninja-gap" between naive C and best-optimized code can be eliminated using relatively straightforward C programming techniques. Even accounting for thread parallel speedup, these results demonstrate a substantial performance gap between naive and tuned C code. In this paper we choose a subset of the benchmarks studied by Satish et al to port to Haskell. We measure performance of these Haskell benchmarks compiled with the standard Glasgow Haskell Compiler and with our experimental Intel Labs Haskell Research Compiler and report results as compared to our best reconstructions of the algorithms used by Satish et al. Results are reported as measured both on an Intel Xeon E5-4650 32-core machine, and on an Intel Xeon Phi co-processor. We hope that this study provides valuable data on the concrete performance of Haskell relative to C.

1. Introduction

It is often claimed that high-level languages provide greater programmer productivity at the expense of some performance; functional languages have been touted as providing reasonable parallel-scalability without huge programmer effort at the loss of some sequential performance. Assessing these claims in general is difficult: instead, in this paper we provide a careful study of six benchmarks in Haskell reporting the sequential, parallel, and SIMD-vector performance in comparison to C.

We use C, as is often done in the literature, to represent what is possible with low-level high-performance languages. But this choice immediately raises the question "What C?". In particular, how much programmer optimization is to be applied to the C program? And what compiler should the C code be compiled with? These concerns may seem minor, but they are not. In a 2012 paper [9] (henceforth "the ninja gap paper"), Satish et al study a range of throughput oriented benchmarks and demonstrate dramatic performance differences between naive implementations and the best known hand-tuned implementations (an average of $23\times$ speedup and up to $53\times$). Moreover, they show that performance comparable to the best-in-class implementations can generally be achieved using relatively-straightforward language-level optimizations applied to the naive versions, combined with appropriate use of a good C compiler. These results suggest that the question "What C" is in fact critical to any such comparison.

We attempt to provide a very careful analysis of the relative performance of C and Haskell on six of these benchmarks, using both the standard Glasgow Haskell Compiler (GHC) and our experimental whole-program optimizing Haskell compiler, the Intel Labs Haskell Research Compiler (HRC). We do not claim that this analysis is the definitive study, nor that this analysis is the only way to do such a study. It is possible that better C programmers could improve on our reconstructions of the C benchmarks, and it is possible that better Haskell programmers could improve on our versions of the Haskell benchmarks. We have had to make choices as to how far outside of the space of idiomatic Haskell programs to go, and to what extent to use unsafe, non-standard, or experimental constructs (such as strictness annotations, explicit strictness (seq), unboxed vectors, and unsafe array subscripting). And of course, our resources for exploring these spaces are finite. Our goal then is not to be definitive, but to be *transparent*.

We believe that our results provide insights into Haskell performance. We show that Haskell can sometimes achieve performance comparable to best-in-class C, that Haskell can often achieve good parallel scalability with little programmer effort, and that Haskell can benefit from SIMD vectorization on some benchmarks. However, we also observe that Haskell performs badly on many benchmarks, and that good scalability is not enough to make up for poor sequential performance. Specifically, we observe that compiled with GHC and run on a 32 core machine, our selected benchmarks run between $1.37\times$ and $90\times$ slower (peak performance) than the best performing implementations. We call this gap *the Haskell Gap*. We show that compiling with HRC reduces the Haskell Gap to between $0.56\times$ and $10.11\times$. We also give measurements on a pre-production Intel Xeon Phi board, demonstrating a measured Haskell Gap with HRC compiled code of between $0.62\times$ and $6.84\times$ of our best C versions. To the best of our knowledge, these are the first performance results for Haskell on the Xeon Phi processor.

1.1 Methodology: C

We are grateful to the authors of the ninja gap paper for graciously providing us with access to archived versions of their C implementations, and for answering our questions in our attempts to reproduce their results. It is important to make clear that this cannot be considered a full reproduction of their work. Architectures and compilers have changed significantly since the original code was written. New issues (such as, very notably, non-uniform memory-

access) arise on the larger and newer machines which we are targeting. These are issues which the original code was not designed to address, and which we lack the expertise and time to address ourselves. As we will discuss further in Section 2, in places compiler technology has substantially narrowed the gap between naive and optimized algorithms. We have also, for various reasons including the need to run on Microsoft Windows, needed to modify some of their original C code. Any mistakes, anomalies or inconsistencies with the original work are most likely due to us.

1.2 Methodology: Floating Point

Floating point semantics are a notoriously difficult issue. Floating point arithmetic is generally not associative, and many identities which hold over the real numbers do hold for floating point numbers in the sense that they may change the precision of the result. Whether this matters is very application dependent. For certain numeric applications in which numeric stability is a critical property of algorithms, predictable rounding semantics may be critical. On the other hand, in many graphics applications performance is critical and details of rounding is more or less irrelevant. For our measurements, we have chosen to give the compilers maximal freedom to optimize floating point operations (the methodology used in the original ninja gap paper). Other choices are reasonable. We discuss this further in Section 2.

1.3 Methodology: Haskell

Porting of the Haskell benchmarks was done by three of the authors, one of whom is a very experienced Haskell programmer, one of whom is very experienced with functional languages but less so with Haskell, and one of whom is a relative novice to functional programming. In all cases, mutual assistance in writing and tuning the benchmarks was provided. A broad goal of the porting effort was to remain more or less in the space of reasonably idiomatic Haskell. It is likely possible to achieve better performance on some of these benchmarks by essentially writing the C code in Haskell using IORefs and unsafe malloc-ed byte arrays. We do not feel this style of programming is an interesting use of Haskell, and it does not reflect well on the goal of high-level programming in general. Where exactly the boundaries of idiomatic programming begin and end are entirely a matter of judgment. For example, we do make extensive use of strictness annotations and other non-standard GHC extensions to Haskell. We discuss particular choices in this regard in the discussion of each benchmark in Section 2.

In tuning the benchmarks, we profited significantly from being our own compiler developers. In particular, we were able to study the generated code of both GHC and our own compiler to better understand weaknesses in the generated code. These weaknesses could often be addressed by small changes in the source code. This technique would be much less accessible to other Haskell developers. We also made extensive use of the Intel VTune™tool to understand and tune performance.

In an ideal world, we would have kept our compiler fixed over the course of this study. However, since our larger goal is the development of the compiler itself, this work necessarily was used to drive compiler development in the sense that weaknesses in the compiler revealed by the benchmarks were sometimes addressed in the compiler. The changes in the compiler were not done in an ad hoc manner simply to address one particular benchmark, but rather were generally beneficial optimizations. Nonetheless, this is a weak point from the standpoint of viewing this as a scientific performance study.

Our focus as a compiler development team is on our own compiler. We made reasonable efforts to select good optimization flags for GHC and to provide fair measurements. However, it is very possible that someone more familiar with the strengths and weaknesses of the GHC compiler might be able to improve upon the relative performance of GHC vs our compiler. The style of benchmark measured here is also particularly favorable to our compiler. We hope that these results will not be taken as a criticism of the GHC compiler, especially given that we rely essentially on GHC as a high-level optimizing front-end for our compiler.

As of this writing, we have only ported a subset of the benchmarks from the ninja-gap paper. This selection is not entirely random - the easier to port benchmarks were chosen for porting first, and the last remaining un-ported benchmarks seem likely to be the most difficult to get good performance on. The selection of benchmarks should if anything therefore be viewed as skewed in favor of Haskell.

1.4 HRC

HRC is discussed in detail elsewhere [6] and we only briefly describe it here. The compiler uses GHC as a front-end and intercepts the Core intermediate language before generation of spinelesstagless G-machine code. Core code for the entire program (including all libraries) is aggregated by our compiler in order to perform whole-program optimization. Some initial high-level optimization and transformation are performed before translation to a strict SSA-style internal representation in which most optimization is done. The backend of our compiler generates code in an extension of C called Pillar [1], which is then transformed to standard C code and compiled with the Intel C compiler or GCC. Our compiler performs a number of loop-based and representation-style optimizations as described elsewhere [6, 7]. In addition, SIMD vectorization is performed where applicable as described by Petersen et al [8].

HRC implements most of the functionality of the GHC system, and can correctly compile and run most of the nofib benchmark suite. The most notable known deficiencies in functionality are that:

1. Re-evaluating a thunk which exited with an exception will produce an error instead of re-raising the exception.
2. Asynchronous exceptions are not supported.

The first issue can be addressed but has not been a priority. Addressing it will likely have some adverse effect on performance of thunk intensive code, but is irrelevant to these benchmarks, which have no laziness in performance critical sections. The second issue seems likely to be impossible to address given our language-agnostic runtime representation.

2. The Benchmarks

We begin by describing qualitatively the benchmarks which we have chosen to port, and the manner in which we have chosen to port them. For each benchmark, the ninja-gap paper studied three implementations: a naive C implementation (“naive C”), a best-optimized implementation (“ninja C”), and an algorithmically tuned C version (“optimized C”). The naive C code for a given benchmark generally consisted of the “obvious” C implementation for that benchmark, with little thought given to performance tuning. The ninja code on the other hand consisted of deeply and carefully optimized code, using compiler intrinsics and pragmas as appropriate, validated to match the performance of the best published results for that problem. Finally, the optimized C code was developed by taking the naive C code and performing small, low-effort algorithmic improvements to produce C code comparable in performance to that of the ninja code. For more details about the algorithms and the C implementations upon which our C code is based, we refer the reader to the ninja gap paper [9].

For our work, we have where possible reconstructed each of the three C versions for each of the selected benchmarks, starting from versions of the code used in the original ninja-gap paper. It is

important to note that since the ninja code was written for previous architectures using (at times) hand-coded assembly or intrinsics, it was explicitly not designed to be "future-proof". That is, unlike the optimized C code which could be successfully retargeted to a new architecture simply by passing different flags to the C compiler, the ninja code would require hand re-coding and re-tuning. While we have for some of these benchmarks attempted this re-coding, we cannot claim to be ninja-programmers, and so it is likely that the ninja code that we measure no longer truly represents the best-optimized code. Similarly, for the optimized C code, it is likely that a small tuning effort comparable to that described in the ninja-gap paper might give further performance improvements on machines with non-uniform memory access (NUMA) behaviors such as those on which we perform our measurements.

Given all this, we emphasize that the reader should interpret our results not as situating Haskell relative to the absolute best C versions, but rather as situating Haskell relative to a range of C versions, from the fairly ordinary, to the very good, to the possibly quite excellent. Nonetheless, for clarity and for consistency with the ninja-gap paper, we continue to use the naive/optimized/ninja terminology throughout the rest of the paper. In the rest of this section, we describe each of the benchmarks that we have chosen to port to Haskell in this manner.

2.1 NBody

The NBody benchmark is an implementation of the naive quadratic algorithm for computing the aggregate forces between N point masses. Given an array of bodies described as a coordinate in space with a mass, the sum of the forces induced by the pair wise interactions between each body and all of the other bodies is computed and placed in an output array.

The translation of this benchmark to Haskell was entirely straightforward using the Repa libraries [2, 4, 5]. Bodies are represented using quadruples of floating point numbers containing x, y, and z coordinates along with a mass. Computed forces are represented as triples of floating point numbers. The vector of bodies is represented using an unboxed dimension one vector of points, and the result vector is represented using an unboxed dimension one vector of forces. The main computational kernel then consists of a parallel map over the points. At each point, another map is performed to compute an intermediate vector containing the pair-wise interactions between the given point and all other points. Finally, a fold is performed over this intermediate vector summing the forces computed by each pair-wise interaction. The computation of the pair wise interaction between two points is computed as follows:

```
pointForce :: Point -> Point -> Force
pointForce (xi, yi, zi, _) (xj, yj, zj, mj) =
  let
    dx = xj-xi
    dy = yj-yi
    dz = zj-zi
    eps_sqr = 0.01
    gamma_0 = dx*dx + dy*dy + dz*dz + eps_sqr
    s_0 = mj/(gamma_0 * sqrt(gamma_0))
    r0 = s_0 * dx
    r1 = s_0 * dy
    r2 = s_0 * dz
    r = (r0, r1, r2)
  in r
```

Strictness annotations are required on the parameters to the helper function which performs the summation in the fold (and nowhere else). The Repa library provides a clean unboxed struct of array (SOA) representation for the data structures, and avoids unnecessary subscript checks. The GHC compiler successfully eliminates all of the intermediate data structures implied by the idiomatic code as written, and the result after further optimization is clean tight inner loop which is easily amenable to SIMD vectorization.

The naive C version of this benchmark uses a struct of array (SOA) representation to represent the input vector of bodies and the output vector of forces, and performs the computation using a simple nested loop. The C compiler is unable to vectorize this version, most likely because it is unable to prove that the array reads and writes in the loop do not interfere.

The optimized C version of this benchmark uses a struct of array representation, keeps all of the data structures in static globals, and accumulates into temporary variables instead of directly into the output array. As a result, the C compiler is able to vectorize this very successfully. We implemented two different versions of this optimizer version, one which unrolls the outer loop four times, and second which does blocking to increase cache locality.

The inner loop for this benchmark is extremely tight, and a substantial speedup is obtained by the C compiler's ability to eliminate a division and square root instruction in favor of a single reciprocal square root instruction. This is done essentially by rewriting the formula $s_0 = m_j/(\gamma_0 * \sqrt{\gamma_0})$ as $s_0 = m_j * (1/\sqrt{\gamma_0})^3)$. The latter formula can be computed using only a single reciprocal square root instruction and series of multiplies. Since the reciprocal square root instruction and each multiply cost only a few cycles (both latency and throughput) as opposed to the division and square root instructions which each cost tens of cycles, this optimization gives a more than 2X speedup. The C compiler is not able to perform this optimization for us on the already vectorized code that we emit, and so we do not get the benefit of this optimization.

The ninja version of this benchmark was originally implemented using SSE intrinsics, and modified for this paper to use AVX intrinsics. All arrays are aligned on 256-bit boundaries allowing the use of aligned loads. The outer loop is hand unrolled four times. The algebraic re-arrangement described above is performed by hand.

2.2 1D convolution

The 1D convolution benchmark performs a one-dimensional convolution on a large array of complex numbers (the real and imaginary components of which are represented as single precision floating point numbers). The kernel for the convolution contains 8192 floating point elements. The inner loop of the simple naive C version consists of the following code:

```
for (i=0; i<rep; i++) {
  for (p=0; p<(end-start); p++) {
    out[p].r = 0.0;
    out[p].i = 0.0;
    for (f=0; f<FSIZE; f++) {
      out[p].r += in[p+f].r * coef[f] -
                  in[p+f].i * coef[f];
      out[p].i += in[p+f].r * coef[f] +
                  in[p+f].i * coef[f];
    }
  }
}
```

As shown, the naive version uses an array of struct (AOS) representation, passes arrays as function arguments, and accumulates directly into the output array. Note that the input arrays are padded to avoid the need for conditionals to deal with boundary conditions. The optimized version passes arrays through globals, aligns all arrays, uses a struct of array representation, and accumulates into temporary variables. The C compiler is able to vectorize this code very successfully.

The ninja version of this code uses the same basic representations as the optimized C version. The original version was implemented using SSE intrinsics, and modified to use AVX intrinsics for this paper. The inner loop in this version is hand-unrolled four times.

Translating this code to Haskell presented a somewhat interesting challenge. While the Repa libraries include support specifically for stencils [4], this support is somewhat preliminary and is limited to two dimensional arrays. Only small fixed size stencils are fully optimized. However, after some brief experimentation a performant implementation was obtained by using the Repa extract function to obtain a slice of the input array which was then zipped together with the stencil array using the convolution function, and then reduced with a fold. It was somewhat surprising to us that GHC was able to successfully eliminate the implied intermediate arrays with this, but combined with our backend optimizations we were indeed able to obtain excellent code. The code of the stencil computation is as follows:

```
convolve0 :: Complex -> Float -> Complex
convolve0 (r, i) s = (r*s - i*s, r*s + i*s)

convolve :: Int -> Stencil -> Data -> Data
convolve size stencil input = output
where
 genOne tag =
  let
   elements = R.extract tag stencilShape input
   partials = R.zipWith convolve0 elements stencil
  in R.foldAllS complexPlus (0.0, 0.0) partials
 shape =  R.Z  R.:. size :: R.DIM1
 [output] = R.computeUnboxedP
            $ R.fromFunction shape genOne
```

Surprisingly, GHC itself performs extremely poorly on this benchmark despite successfully eliminating the intermediate arrays. We have not as of this writing been able to diagnose this, other than to observe that it appears to be unable to elimimate some remaining allocation from the inner loop.

2.3 2D Convolution

The 2D convolution benchmark performs a convolution over a two-dimensional array of floating point numbers using a 5x5 stencil.

The naive C version passes arrays as function arguments, but uses a temporary variable as an accumulator in the inner loop. The C compiler is able to successfully vectorize this code, yielding good speedups. The ninja gap paper reports using a preliminary version of the Cilk++ array notation to produce an optimized version of this code vectorized on the second-from-outer loop. The original version of this code that we obtained was not compatible with the current C compiler and we were not able to reconstruct this code in the time available, and so we do not present results for this configuration, comparing instead only the naive vectorized and the ninja versions of the program.

The ninja version of the code is implemented using SSE intrinsics, which we have not yet updated to the wider AVX instructions. The ninja code performs the outer-loop vectorization described in the original ninja-gap paper, fully eliminating the inner loop in favor of straightline SSE code.

Producing a Haskell version of this code was entirely straightforward, since the Repa stencil libraries provide direct support for this style of stencil operation. Interestingly, the problem as originally written used a stencil of all ones, with the result that the compiler stack was able to eliminate all of the stencil multiplies entirely. While indicative of the greater optimization flexibility available in a functional language, it was felt that this was not indicative of the performance of the code on general stencil problems, and so the stencil was changed in both the C and the Haskell code to consist of all twos. With this change we are still able to constant propagate the value, but the multiply can no longer be eliminated. The generated code is overall quite good, and our compiler is able to vectorize the inner loop. However, due to unrolling performed by the Repa libraries, our vector code must use strided load instructions which are not supported in the AVX instruction set and instead must be emulated with some loss of speedup. Since there is overlap in the values loaded in the unrolled loop, lowering the strided loads to scalar loads in our compiler (rather than emulating them in the code generator) allowed the compiler to eliminate a number of the loads using common sub-expression elimination.

A difficult issue arose in the translation and measurement of the 2D convolution benchmark, as well as several other of the ported benchmarks. In order to measure large enough problem sizes to obtain good timing results (particularly at larger number of processors), the C programs were written to iteratively re-compute the result an arbitrary number of times as specified on the command line. The re-computation is performed after distribution of the work to the worker threads: that is, each worker thread receives a portion of an array to convolve, and an iteration count telling it how many times to perform the convolution. We saw no way to implement this directly in the Haskell code, and were forced instead to iterate notionally outside of the worker threads by repeatedly doing the entire convolution. This is problematic for comparison purposes for a number of reasons: firstly in that it potentially introduces additional synchronization and communication overhead; secondly in that introduced garbage collection into the equation since new result arrays must be allocated and collected; and thirdly that it produces somewhat different cache behavior than in the C program. All of the C benchmarks were written in this style, and we do not have a general solution to this problem. Where possible, we have chosen to increase the problem size to the point where a single iteration suffices. For some programs, such as the 2d convolution, this was not possible. We modified the C code to do a barrier between iterations and to swap the source and destination arrays between iterations. Garbage collection time does not account for a large portion of the 2D convolution time; however, there are almost certainly mutator effects as a result of the garbage collections and the fresh allocations of the result arrays.

2.4 Black Scholes

The Black Scholes benchmark computes put and call options. The computational kernel is a loop computing a cumulative normal distribution function. The inner loop of the computation contains conditionals.

The naive C version uses an array of struct representation. The C compiler does not choose to auto-vectorize the loop. Annotating the loop with "#pragma simd" allows the loop to be vectorized: however, the AOS representation is poorly suited to vector code generation.

The optimized C version uses a struct of array representation, but the auto-vectorizer still chooses not to vectorize the loop. Once again annotating the loop with "#pragma simd" suffices to cause vector code to be generated. The SOA format is better suited to vector code generation.

The ninja version use the same representation as the optimized C version but is written using AVX intrinsics directly.

The Haskell port of this code was subject to fairly extensive performance tuning. The core of the kernel is a relatively straightforward translation of the C code, with some re-arrangement to eliminate some conditionals. One strictness annotation is used on a helper function.

The option data is represented as a tuple, and the input array of options is represented as a one-dimensional Repa unboxed array of options. The Repa library performs the AOS-SOA conversion on the input data. The iteration over the option array to produce the result is performed using the Repa "map" function over the input array.

2.5 Volume Rendering

The volume rendering benchmark projects two-dimensional images into a three-dimensional volume. The computation is irregular, with conditionals and a data-dependent while-loop.

The naive C version of this benchmark passes all arguments in global variables, and is sufficiently well-optimized by the C compiler that no optimized version was written.

The ninja version of the benchmark uses handwritten AVX intrinsics. Mask registers are simulated using standard AVX registers.

Writing a performance version of this benchmark in Haskell was somewhat challenging, since the structure of the kernel computation does not fit naturally into any of the patterns supported by existing array libraries. The final tuned implementation implements the inner loop of the computation as recursive function which traverses a section of the opacity and visibility arrays from the input data. Unsafe indexing is used in this function, and strictness annotations were required on the arguments. The function contains multiple recursive calls, all in tail positions.

The outer iteration of the computation is performed using the Repa "traverse" array generator to traverse the input array of rays, obtaining the index of each element in the process for use in the call to the recursive function. The use of traverse incurs an extra array bounds check which is not eliminated in this code. Strictness annotations are also used on these iteration functions.

2.6 Tree Search

The tree search benchmark does a search on a structured tree to find data by index. The code is consequently quite control dependent.

The naive C version is implemented using the obvious binary search algorithm over a static index structure laid out breadth first in an array.

The optimized C version implements a fast algorithm by Kim et al [3], performing multi-level search over a tree re-organized into hierarchical blocks. The blocking structure helps to improve page and cache memory locality, and even permits a SIMD vector implementation after algorithmic change to avoid early loop exits. Further optimizations such as loop unrolling make the code suitable for compiler auto vectorization.

The Ninja C version implements the same algorithm as the optimized C with hand written SSE intrinsics. It also implements SIMD-level blocking as well as pipelining, neither of which are used by the optimized C version. These optimizations require the use of gather instructions which must be emulated on CPU.

Both the ninja C and the optimized C programs can only deal with a specific fixed tree depth in order to completely unroll the inner loops and get rid of all early loop exits. In contrast, the naive C can deal with arbitrary tree size.

The Haskell version of the code represents the search tree using a Repa unboxed array, and implements the same optimized binary search algorithm as the optimized C version. Rather than manually unrolling the loop as in optimized C, the Haskell program represents a single step of tree traversal as a function, composes that into a traversal for a block, and composes multiple block traversals into a traversal for the entire tree. GHC is able to inline all the intermediate function compositions and thus achieves the same effect as loop unrolling. HRC is then able to vectorize the resulting program.

Just like the ninja C and optimized C programs, the Haskell version can only handle a specific fixed tree depth. One may argue that statically composing traversal functions to get a single fused search function for a fixed tree depth is beyond the scope of idiomatic Haskell. But since we are implementing the same algorithm as the optimized C, we feel it is fair to compare fused Haskell functions with loop-unrolled C functions, especially when both versions are able to vectorize.

3. Performance Comparison

Generally we analyze five different configurations, three C configurations and two Haskell configurations, but sometimes some configurations are not available or not reportable for reasons noted below. The three C configurations are called naive, optimized, and ninja. They correspond to the three types of C in the ninja gap paper. C naive is straightforwardly written C that solves the problem; C ninja is hand optimized and hand tuned, often using compiler intrinsics and pragmas, and often hand implementing optimizations that the C compiler cannot or will not perform; C optimized is an intermediate point where algorithm transformation and compiler pragmas and/or flags are used to tune the performance. Note that very few programmers have the skill to produce C ninja; many more programmers have or can be taught the skills to produce C optimized.

We analyze the performance of the benchmarks in terms of both sequential performance and parallel speedup. For each benchmark we present two charts: a chart show sequential speedup relative to the C naive configuration, and a chart showing parallel speedup relative to the best-performing sequential algorithm.

For the sequential performance chart, all numbers are normalized to C naive: that is, for a given configuration, the height of the bar on the Y axis is computed by dividing its run time by the run time of the C naive configuration. Hence, lower is better on these charts.

For the parallel speedup charts, we show speedup relative to the best sequential configuration, whichever that is: generally, but not always this is the C ninja configuration. For a given configuration, the X axis indicates the number of parallel threads run, and the value on the Y axis is computed by dividing the run time of that configuration (on that number of threads) by the run time of the best sequential configuration on one thread. Higher is better on these graphs.

For each benchmark, we also calculate the *Haskell Gap* between the best performing C code and the best performing Haskell code. This number is calculated by dividing the fastest runtime for the Haskell version (at any number of cores) by the fast runtime for any of the C versions (at any number of cores). This number gives the slowdown (or speedup) factor of the Haskell code relative to the best C, allowing for both sequential and parallel speedup.

All of the C programs measured in this paper were compiled using the Intel C++ Compiler, version 13.1.2.190. All of the GHC compiled Haskell programs were compiled with GHC version 7.6.1. When compiled with the LLVM backend, LLVM version 2.9 was used. For all GHC configurations, the "-optlc-enable-unsafe-fp-math" option was passed to GHC to permit unsafe floating point optimizations to be performed.

3.1 CPU

We first report results as measured on an Intel Xeon CPU with 128 GB of RAM running Microsoft Windows Server 2008. The machine contains 4 Intel Xeon E5-4650 (codename Sandy Bridge) processors, each of which has 8 cores for a total of 32 execution cores. Each core has 32KB L1 data and instruction caches and a 256KB L2 cache. Each processor has a 20MB L3 cache shared among 8 cores. All runs were performed with hyperthreading off.

3.1.1 1D Convolution

For the 1D convolution benchmark, we measured four configurations: the naive C, the optimized C, the ninja (AVX) C, and Haskell compiled with our compiler. Haskell compiled with GHC runs several orders of magnitude slower. We do not include GHC numbers since running the GHC compiled code on a sufficiently large problem size is impractical for this benchmark. All numbers were taken by convolving 3,000,000 elements.

The relative sequential performance is given in Figure 1. For this benchmark, the optimized C code runs in approximately 16% of the time taken by the naive C program. This is a good example of the hazards of comparison to C that we wish to highlight in this paper. Even keeping the C compiler fixed, we observe an 85% reduction in run time simply through the use of a few quite small changes to the source code. The ninja code (written using AVX intrinsics) does not perform as well as the code optimized by the C compiler. This may reflect either improvements in the C compiler since the original ninja gap paper was written, or changes in the underlying architecture, or both. On the latter point, it is important to note that the ninja versions of these benchmarks, written using intrinsics, are explicitly not "future-proof": that is, they are highly tuned to an explicit architecture and instruction set.

The Haskell code compiled with HRC runs in 18% of the time taken by the naive C code. This is a substantial speedup over the naive code, and only a little slower than the best sequential code. We are very pleased with the performance of this benchmark: the code is written in a natural and idiomatic style, the GHC frontend fuses away the intermediate data structures effectively, and our compiler is able optimize and vectorize the key loops very effectively.

The speedup graph is given in Figure 2. All of the configurations exhibit good scalability and are able to use effectively all of the processors. However, since the optimized C version scales quite well and is substantially faster at one processors, all of the configurations exhibit a significant performance gap at 32 processors. The drop off at 32 processors is repeatable, and likely reflects a weakness in our support for parallel Haskell code, which is a recent addition. The final result is an overall Haskell Gap of $1.24\times$.

3.1.2 Black Scholes

For the Black Scholes benchmark, we took measurements using a range of configurations. The three primary C configurations are the naive C configuration, the optimized C configuration, and the ninja (AVX) configuration. Since the C compiler was unable to (or chose not to) perform auto-vectorization, each of the first two configurations was measured with and without the use of a simd pragma. Hence the "C Naive NOSIMD" configuration is the C Naive code without a simd pragma, whereas the "C Naive" configuration is the C Naive code with a simd pragma on the inner loop, and similarly for the optimized C configuration. The ninja configuration uses AVX intrinsics. We also present measurements for the Haskell code compiled both with our compiler (HRC) and the GHC compiler (GHC, GHC LLVM). The benchmarks were run on 40,000,000 options.

The relative sequential performance is given in Figure 3. The optimized C code (SIMD) runs in 20% of the time of the naive (not SIMD), outperforming the Ninja code by 40%, again possibly reflecting changes in architectures of compiler improvements.

The Haskell code with our compiler runs slightly faster than the naive C code, but is a factor of $4.7\times$ slower than the optimized C code. The GHC LLVM compiled code is slower than the naive C code by a factor of $3.85\times$, and is slower than the optimized C SIMD code by almost a factor of $20\times$.

Figure 4 shows the speedup of the main configurations relative to the best sequential version (the optimized C code). We leave off

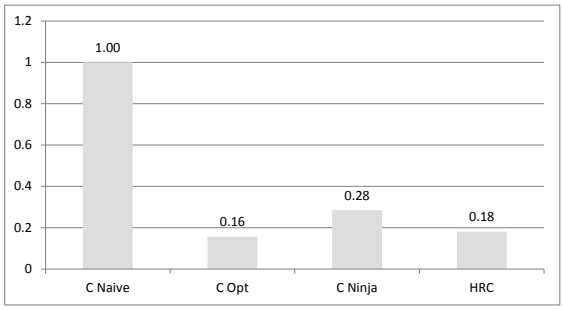

Figure 1. 1D convolution run time normalized to C Naive

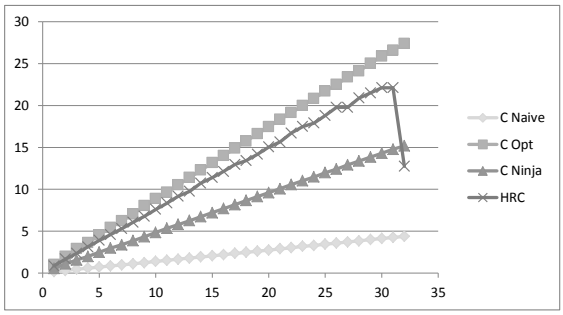

Figure 2. 1D convolution speedup relative to best sequential

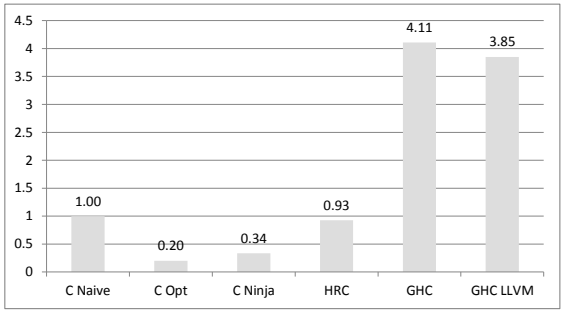

Figure 3. Black Scholes run time normalized to C Naive

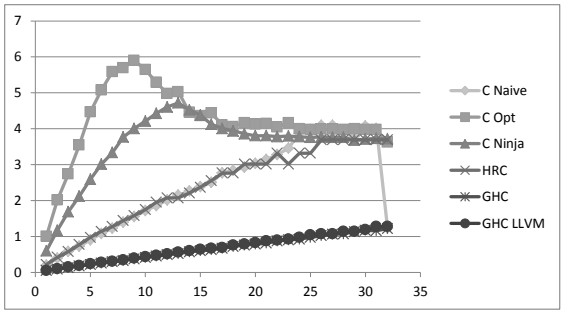

Figure 4. Black Scholes speedup relative to best sequential

the non-SIMD C configurations to avoid cluttering the chart - their scalability essentially mirrors that of the SIMD versions. The scalability results for this benchmark are significantly mixed. The ninja code scales reasonably well up to 8 or 9 processors, and then flattens out. It is likely based on our preliminary investigations that this is related to migration of data between sockets. The server machine on which these measurements were taken contains 4 separate physical processors, each with 8 cores. The phase transition at 8 threads is suggestive, but not definitive.

Interestingly, the Haskell programs for this benchmark scale relatively well. The GHC compiled code scales cleanly, but cannot overcome the sequential performance deficit, resulting in a final Haskell Gap of $4.62\times$. The code compiled with our compiler comes surprisingly close to matching the peak performance of the optimized code, albeit using 4 times as many processors. The final Haskell Gap for the HRC code is $1.60\times$.

3.1.3 2D Convolution

For the 2D convolution benchmark, we were only able to reconstruct naive C and ninja C versions of the code, the latter using AVX intrinsics. We also measured the Haskell code compiled with both the Intel compiler and the GHC compiler. All measurements were taken by convolving a 4000 by 4000 image repeatedly for 32 iterations.

The relative sequential performance is given in Figure 5. The ninja code is substantially faster than the naive C code as is to be expected given the lack of vectorization. The GHC compiled Haskell code is almost $2.7\times$ slower than the naive C code and hence almost $20\times$ slower than the ninja code. The GHC LLVM backend gives substantial benefit on this benchmark, exhibiting only a $1.53\times$ slowdown over the naive C, or a $11\times$ slowdown over the ninja C. However, the Intel compiled Haskell code is substantially faster than the naive C code, and only around $1.4\times$ slower than the ninja code. The original ninja gap paper reported reaching within 1.3X of the ninja code with their optimized C code [9].

Figure 6 shows the parallel speedup of the various configurations relative to the sequential performance of the ninja configuration. The ninja code scales well up to 9 or 10 processors, again suggestive of issues arising with the multi-socket architecture. However, none of the other configurations is able to make up the substantial head-start provided by the significantly better sequential performance. The GHC compiled code remains the slowest throughout the full range, with a final Haskell Gap of $6.18\times$. The Intel compiled Haskell code suffers a somewhat similar trajectory to the ninja code, flattening out and even dropping a bit after 10 processors. Interestingly, the naive C code scales quite cleanly, and eventually beats the best performance of the Haskell code by making better use of higher number of cores. The final Haskell Gap for the HRC compiled code is $2.55\times$.

3.1.4 N Body

For the N Body benchmark, we measure six different configurations. The C versions include a naive C version, an optimized C version (hand unrolled), and a ninja C version written using AVX intrinsics. We measure the Haskell code as compiled with HRC, and with GHC and GHC LLVM. The blocking version of the C code that we implemented showed no additional performance or scalability on this architecture, and so we elide those results. The benchmarks were run simulating 150,000 bodies.

The relative sequential performance is given in Figure 7. The C compiler is able to do an excellent job of vectorizing and optimizing this benchmark. The optimized C version run in approximately 13% of the time of the naive C versions, and is only 18% slower than the ninja code. The GHC compiled code is slower than the

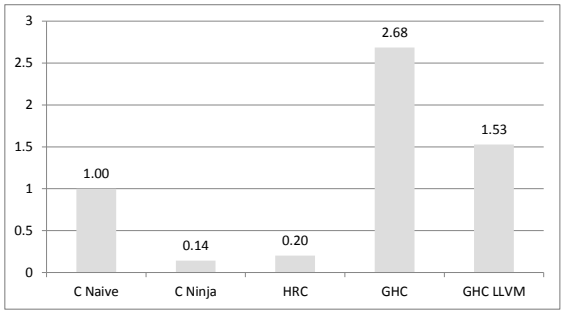

Figure 5. 2D convolution run time normalized to C Naive

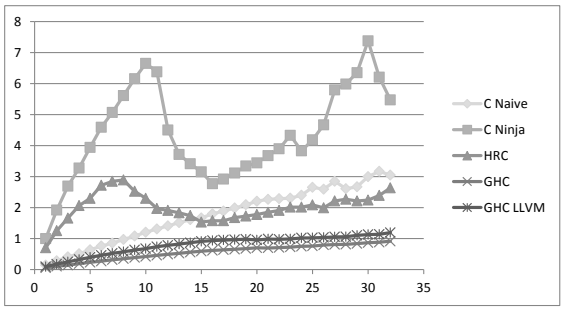

Figure 6. 2D convolution speedup relative to best sequential

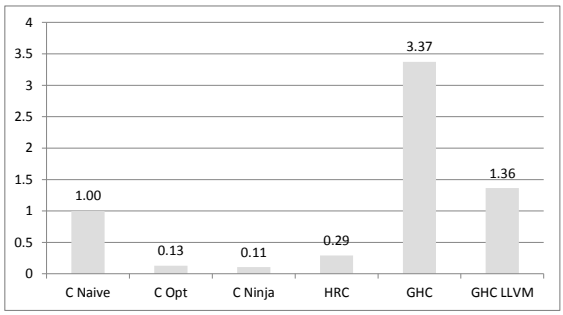

Figure 7. N body run time normalized to C Naive

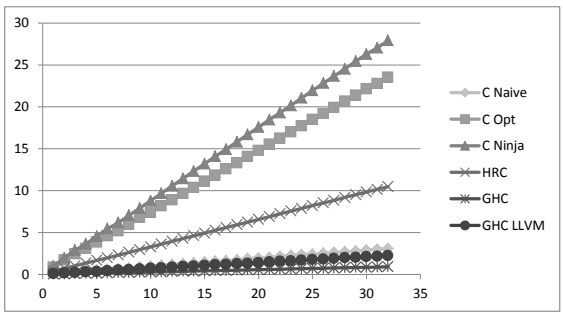

Figure 8. N Body speedup relative to best sequential

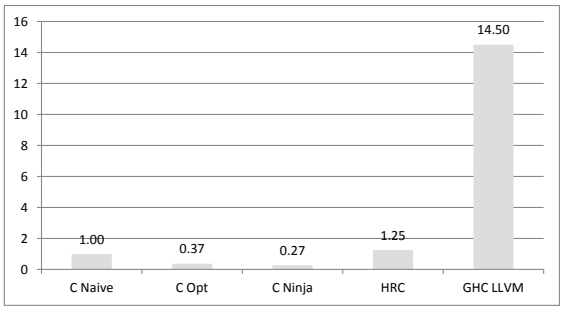

Figure 9. TreeSearch run time normalized to C Naive

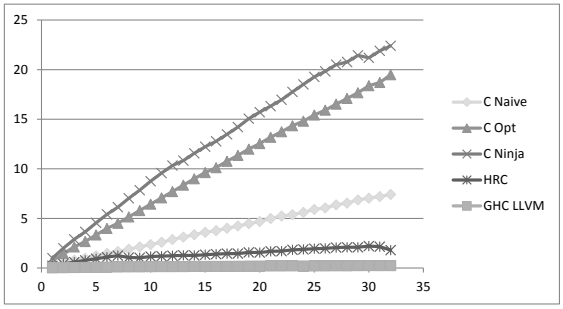

Figure 10. TreeSearch speedup relative to best sequential

naive C code by a factor of $3.37\times$, and is $31\times$ slower than the ninja code. The GHC LLVM code is only $1.36\times$ slower than the naive code, or $12.45\times$ slower than the ninja code.

HRC produces code that is significantly faster than the naive C code, primarily because of its ability to vectorize the code. It remains a factor of $2.6\times$ slower than the optimized C versions however, due to the C compiler's ability to eliminate the division and square root instructions from the inner loop as discussed in Section 2. On the one hand, this is a disappointing result in that there is a substantial gap between the Haskell code and the C code. However, we find it encouraging that Haskell code can be optimized to the point that machine level peephole optimizations can make this substantial a difference.

It is worth emphasizing here that the relative performance here is highly dependent not just on the choice of compilers and algorithms, but even on the flags passed to the compiler. Passing options requiring the C compiler to maintain the source level precision of the division and square root operations results in this optimization being disabled, reducing the performance of the optimized C code to almost exactly that of the Intel compiled Haskell code. Moreover, in order to vectorize the code both the C compiler and HRC must re-associate floating point operations which does not preserve source level semantics. Consequently, forcing fully strict floating point semantics reduces performance even further for both the C and the Haskell code.

Figure 8 shows the speedup of all of the configurations relative to the ninja sequential performance. All of the benchmarks scale extremely well, and maintain approximately the same relative performance throughout the full range of processor counts. The final Haskell Gap for GHC LLVM is $12.5\times$, and for HRC is $2.67\times$.

3.1.5 TreeSearch

For the TreeSearch benchmark, we measure five different configurations: a naive C version, an optimized C version, ninja C version with SIMD blocking and pipelining, a Haskell version compiled with HRC, and the same Haskell code compiled with GHC LLVM. The ninja C code has not been ported to AVX architcture, so we use a SSE version only. On the other hand, both the optimized C version and the Haskell version compiled with HRC have been vectorized to use AVX intrinsics. All programs were run with 95 million queries over a binary tree of depth 24.

The relative sequential performance for these configurations is given in Figure 9. For this code, the optimized C is 3 times faster than the naive C, mostly because it is blocked, vectorized, and specialized to handle fixed tree depth of 24. The ninja version using SSE intrinsics is a little bit faster still than the optimized C. The Haskell code compiled with HRC is slower than the naive C code by 25%. We are still investigating what is the real cause for this, but for a smaller tree depth of 16, the same Haskell code compiled with HRC runs twice faster than naive C code, and slightly slower than optimized C code. GHC LLVM is $14.50\times$ slower than naive C.

Figure 10 shows the speedup of all of the configurations relative to the naive C sequential performance. All versions of the C code scale quite well, but the Haskell versions are a bit worse. The final Haskell Gap for GHC is $90.38\times$ and for HRC is $10.11\times$.

3.1.6 Volume Rendering

For the Volume Rendering benchmark, we measure five different configurations: a naive C version, a ninja version written with SSE intrinsics, a Haskell version compiled with the HRC Compiler, and the Haskell version compiled with GHC and GHC LLVM. All runs were performed by rendering 1,000,000 voxels, repeating the computation 1000 times. The requirement to repeat the computation is somewhat unfortunate, not only because of the difficulties of expressing this effectively in Haskell, but also because of the difficulties in preventing GHC and HRC from eliminating the unecessary repetitions. Our initial implementation performed far better than any C version—but only because it was able to remove the redundant main calculation from the iteration loop. Working around this is a finicky process, and where possible we have avoided using iteration in the benchmarks.

The relative sequential performance for these configurations is given in Figure 11. The ninja code runs in 62% of the time of the naive C code. The GHC compiled Haskell code takes a factor of $2.67\times$ times slower than the naive C code. The HRC compiled code runs $1.25\times$ slower than the naive C.

The speedup of all of the configurations relative to the ninja code is given in Figure 12. The C configurations scale well to 8–10 processors and then flatten out; whereas the Haskell configurations scale well, except again for the HRC drop off at 32 processors reflecting a likely weakness in our threading runtime. Eventually HRC scaling overcomes its lack of sequential performance and out performs the best C. However, we should be cautious in concluding too much here as the C code may not be properly optimised for multiple sockets. The final Haskell Gap for this benchmark is $0.56\times$ for HRC, and $1.37\times$ for GHC.

3.1.7 The CPU Haskell Gap

Figure 13 summarizes the Haskell Gap for these benchmarks, using HRC and GHC with the LLVM backend. For 1D convolution, we do not present GHC numbers—they are likely to be very high as we are unable to run GHC on these benchmarks in reasonable time. Our use of the Haskell Gap measurement in these benchmarks is intended to capture the overall potential peak performance achievable using Haskell, relative to well optimized C versions, accounting for both sequential performance, SIMD parallelism, and thread parallelism. We believe that this emphasizes the point that achieving performance parity with low-level languages necessarily requires

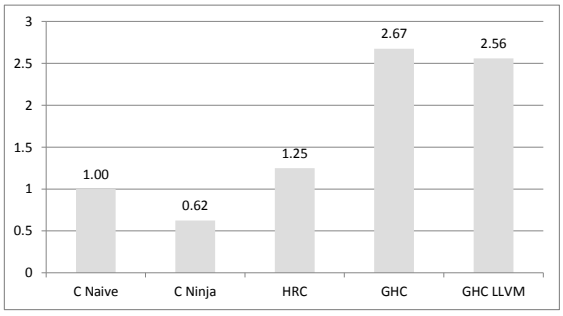

Figure 11. Volume Rendering run time normalized to C Naive

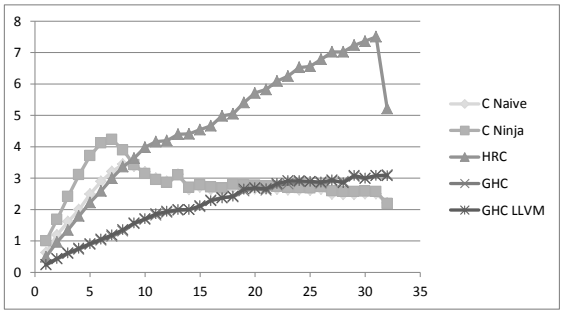

Figure 12. Volume Rendering speedup relative to best sequential

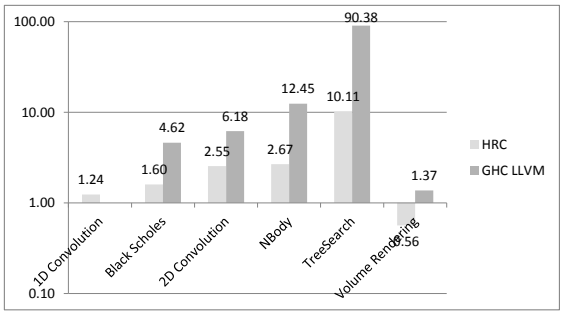

Figure 13. The Haskell Gap on CPU (log scale)

both good sequential performance and good scalability. For certain of these benchmarks, generally ones in which we are able to effectively leverage SIMD parallelism and provide good baseline sequential performance, the Haskell Gap is encouragingly small. For others however, the gap remains wide.

3.2 Xeon Phi Performance

We have also measured a subset of the benchmarks on a Xeon Phi 57 core co-processor. The board we have access to is not a final production chip, and may have additional idiosyncracies in addition to the odd core count. Our support for the Xeon Phi is very preliminary and very little performance tuning has been performed. The vector support in particular is a very recent addition, and contains a number of performance compromises for the sake of achieving initial functionality. Nonetheless, we believe that these numbers are interesting, measuring as they do the scalability of a Haskell implementation on a machine with a very large number of cores. To the best of our knowledge, these are the first Haskell

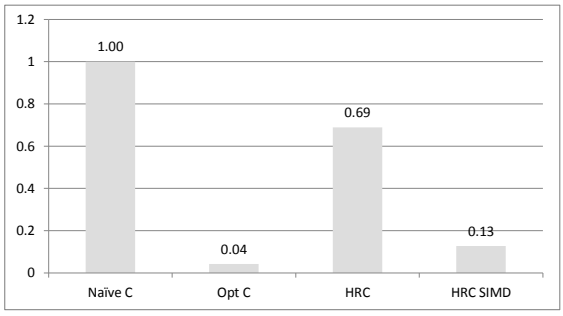

Figure 14. 1D convolution Xeon Phi run time normalized to C Naive

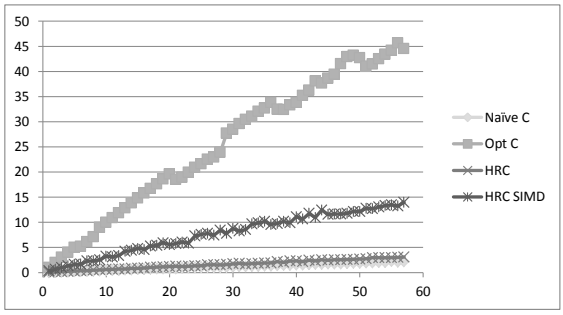

Figure 15. 1D convolution Xeon Phi speedup relative to best sequential

performance numbers reported for the Xeon Phi. There is no GHC version available targeting Xeon Phi, and so we report Haskell numbers using HRC only.

3.2.1 1D Convolution

For the 1D convolution benchmark, the naive and optimized C configurations were compiled for the Xeon Phi. Since the ninja code uses AVX intrinsics, it is not portable to the Xeon Phi which uses a different vector ISA. In addition to the C configurations, we present measurements for Haskell code compiled by HRC with our vectorization pass turned on (HRC SIMD) and off (HRC).

Figure 14 shows the sequential runtime relative to the C Naive configuration. The optimized C configuration is able to take good advantage of the wide vector instruction set, running in 4% of the time of the naive C configuration. The Haskell code is also able to beat the naive C configuration, running 31% faster without vectorisation and 87% faster with; however, it is still $3\times$ slower than the optimised C.

Figure 15 shows the speedup of the configurations at 1 to 57 threads over the optimized C sequential runtime. All of these configuration scale cleanly up to 57 threads. The final measured Haskell Gap is $3.27\times$.

3.2.2 2D Convolution

For the 2D convolution benchmark the naive C configuration was compiled for the Xeon Phi, and a version of the ninja code was produced by modifying the implementation used in the original ninja gap paper to measure results on an earlier software development platform sharing the same ISA. Haskell code was compiled with HRC, again with and without vectorization.

Figure 16 shows the sequential runtime relative to the C Naive configuration. The ninja C configuration again gets substantial

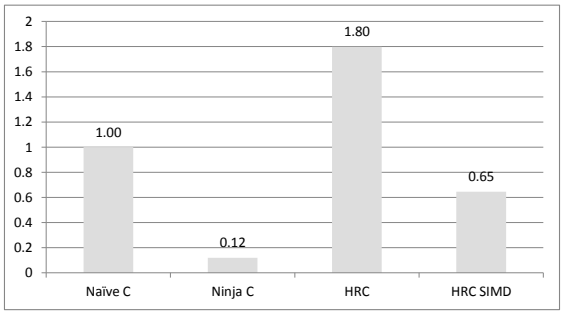

Figure 16. 2D convolution Xeon Phi run time normalized to C Naive

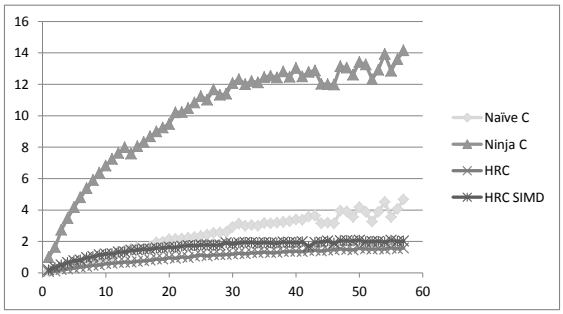

Figure 17. 2D convolution Xeon Phi speedup relative to best sequential

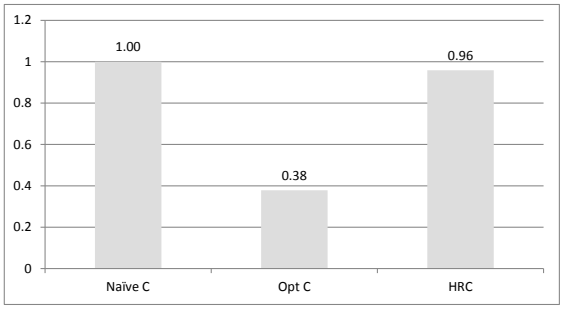

Figure 18. Black Scholes Xeon Phi run time normalized to C Naive

speedups from the wide vector units, running in 12% of the time of the naive C configuration. The Haskell code is 80% slower than naive C without vectorization, but vectorization boosts the results by $2.8\times$ to a speedup of 35%. However, it is still over $5\times$ slower than ninja C. We have not yet investigated why the speedup is not larger given the wide vector widths.

Figure 17 shows the speedup of the configurations at 1 to 57 threads over the ninja sequential runtime. The naive C code scales fairly linearly, but the ninja C and HRC configurations taper off somewhat. The final Haskell Gap for this benchmark is $6.84\times$.

3.2.3 Black Scholes

For the Black Scholes benchmark the naive C and optimized C configurations were compiled for the Xeon Phi, and the Haskell code was compiled with the HRC. No vectorization results are

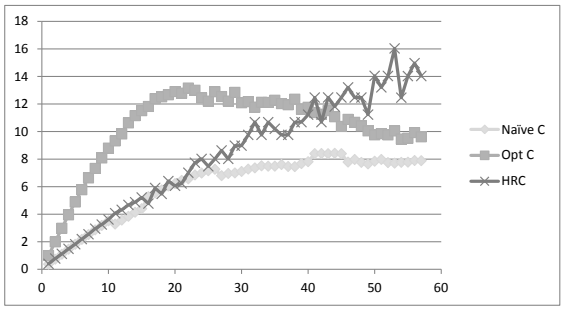

Figure 19. Black Scholes Xeon Phi speedup relative to best sequential

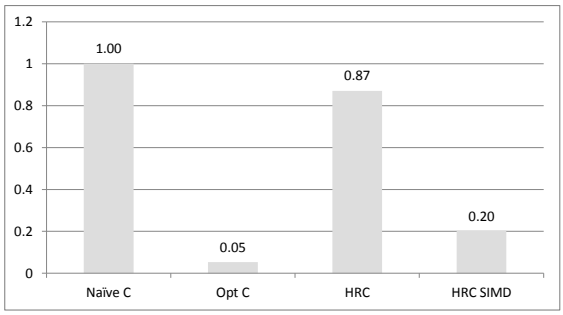

Figure 20. N body Xeon Phi run time normalized to C Naive

reported for HRC, since we are unable to vectorize the branches in the inner loop at this time.

Figure 18 shows the sequential runtime relative to the C Naive configuration. The optimized C configuration runs in 38% of the the time of the naive C version, possibly reflecting the cost of the required masking operations to handle the conditionals internal to the loop. The HRC compiled code is very slightly faster than the naive C code, but substantially slower than the optimized C code.

Figure 19 shows the speedup of the configurations at 1 to 57 threads over the optimized C sequential runtime. For this benchmark, the optimized C code scales poorly past 20 processors, with performance dropping off significantly at higher numbers of processors. Similarly, the naive C code ceases to scale at around 25 processors. The Haskell code on the other hand scales very well, with the result that at high numbers of processors the Haskell code edges out the peak performance of the optimized C code. The final Haskell Gap for this benchmark is $0.88\times$.

3.2.4 N Body

For the N Body benchmark the naive C, optimized C, and blocked C configurations were compiled for the Xeon Phi, and the Haskell code was compiled with HRC, again with and without vectorization.

Figure 20 shows the sequential runtime relative to the C Naive configuration. The optimized C configuration runs in 5% of the the time of the naive C version—a substantial increase in performance. The blocked version of this code again gives no significant benefits over the unrolled version. Our compiler is able to vectorize this code for reasonably good speedup, albeit not as significant a drop as with the C code.

Figure 21 shows the speedup of the configurations at 1 to 57 threads over the optimized C sequential runtime. Much as with the Black Scholes code, the optimized C code scales poorly past 27

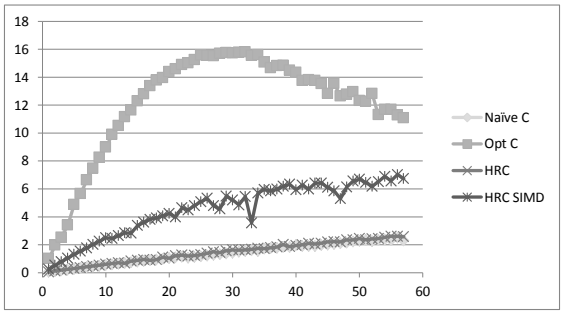

Figure 21. N Body Xeon Phi speedup relative to best sequential

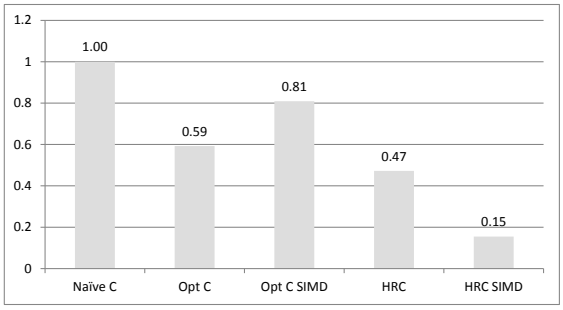

Figure 22. TreeSearch Xeon Phi run time normalized to C Naive

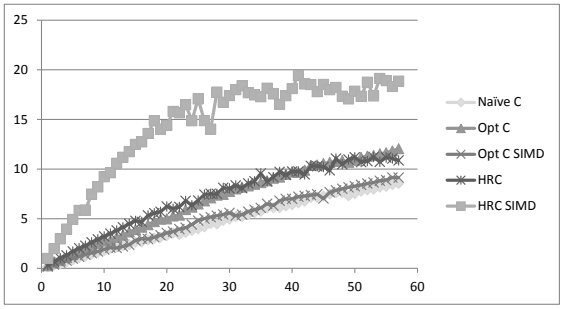

Figure 23. TreeSearch Xeon Phi speedup relative to best sequential

processors, with performance dropping off significantly at higher numbers of processors. The naive C code however scales fairly linearly. The HRC SIMD configuration continues to scale up to 57 processors, but with a decreasing slope at higher core counts. Despite the superior scalability at higher core counts, the Haskell code is not able to overcome the sequential performance deficit. The final Haskell Gap for this benchmark is $2.25\times$.

3.2.5 TreeSearch

For the TreeSearch benchmark, since we have yet to re-produce a ninja version using the Xeon Phi vector ISA and its naive gather support, we only report the performance for the naive C, optimized C compiled with Intel C compiler for the Xeon Phi. The Haskell program were compiled and vectorized with HRC. We report both vectorized and non-vectorized versions for optimized C and Haskell compiled with HRC. All benchmarks were run with 10 million queries over a binary tree of depth 24.

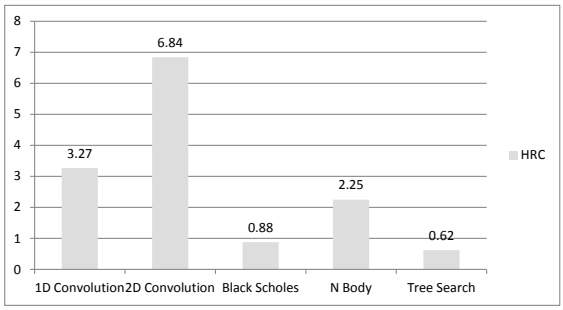

Figure 24. The Haskell Gap on Xeon Phi

The relative sequential performance for these configurations is given in Figure 22. Again, we observe poor performance with the standard optimized C code. The C code optimized for a depth 24 tree (Opt C) is 40% faster than the naive C, but surprisingly the vectorized version (Opt C SIMD) is about 30% slower than the non-vectorized version. We have yet to confirm the exact cause for this slowdown, but suspect it was not able to use the native gathering intrinsic available on the Xeon Phi architecture. The vectorized Haskell version is the fastest of all, about 85% faster than naive C. This result provides an interesting comparison to the performance numbers on CPU for the same benchmark, contrasting the difference in hardware architectures.

Figure 23 shows the speedup of all the configurations relative to the HRC SIMD sequential runtime. All versions scale relatively linearly except for HRC SIMD, which scales poorly beyond around 30 threads. We have good reasons to believe a native ninja C version if available could beat Haskell, but from the results presented here, the final measured Haskell Gap for this benchmark is $0.62\times$ when compared to the best C performance available for now.

3.2.6 The Xeon Phi Haskell Gap

Figure 13 summarizes the Haskell Gap on Xeon Phi for the subset of the benchmarks ported to that architecture. We are greatly encouraged to achieve an overall improvement in peak performance over the best C version on Black Scholes and Tree Search. Our performance on the 1D convolution, and 2D convolution, benchmarks are disappointing when compared to our performance on the CPU. This almost certainly reflects the preliminary nature of our vectorization support on this architecture, and in particular some new issues to be resolved in dealing with a 64-bit architecture.

4. Conclusions

We strongly believe that empirical performance comparisons to C and other high-performance languages serve as a valuable reference point and sanity check for work on optimizing functional languages in general, and Haskell in particular. However, we hope that this paper makes the point that such comparisons are extremely difficult to do well. There are always, at some point, judgment calls to be made—among them the crucial questions "What C?", and "What C compiler?". A benefit of programming in C is that there are substantial opportunities for hand-optimization - as we show in this paper, relatively simple code transformations can make dramatic changes in performance. Therefore, exactly what C code is compared to is critical. Similarly, the choice of C compiler and the options passed to it can significantly change the result of the comparison. Finally, there is always the question of what is "fair" to use in the C code. Is the use of pragmas to induce vectorization where the compiler otherwise would not "fair"? What about intrinsics? What about inline assembly code? To what extent should we allow

the C compiler to re-arrange floating point computations in ways that may change the precision of the computed result?

And on the other side of the equation, what Haskell code should be used for a comparison? One can, with sufficient effort, essentially write C code in Haskell using various unsafe primitives. We would argue that this is not true to the spirit and goals of Haskell, and we have attempted in this paper to remain within the space of "reasonably idiomatic" Haskell. However, we have made abundant use of strictness annotations, explicit strictness, and unboxed vectors. We have, more controversially perhaps, used unsafe array subscripting in places. Are our choices reasonable?

We do not believe that there are definitive answers to these questions. We have tried, in this paper, to explore very carefully a space of answers to these questions that we feel is reasonable. We have shown that for our notion of "reasonable" Haskell, using the compiler technology we have developed, there are reasonable C programs which are significantly out-performed by our reasonable Haskell programs; and that there are other, equally reasonable C programs which in turn significantly out-perform our reasonable Haskell. We have also tried, as best as possible, to leverage previous work [9] to situate our choices of "reasonable" programs relative to the best published algorithms. We hope that this work provides a valuable set of data points for programmers and implementers wishing to understand better how certain classes of Haskell programs stack up against "equivalent" C programs. We also hope that this work encourages a practice of taking comparisons seriously, and presenting them transparently, with the understanding that every such comparison inevitably relies on making choices and is hence only meaningful insofar as those choices can be seen and understood by the reader.

Acknowledgments

We are grateful to the authors of the ninja gap paper [9] for providing us with access to their source code, answering our questions, and most of all for providing the inspiration for this work with their careful analysis.

References

[1] T. A. Anderson, N. Glew, P. Guo, B. T. Lewis, W. Liu, Z. Liu, L. Petersen, M. Rajagopalan, J. M. Stichnoth, G. Wu, and D. Zhang. Pillar: A parallel implementation language. In *LCPC*, pages 141–155, 2007.

[2] G. Keller, M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 261–272, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3.

[3] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. Fast: fast architecture sensitive tree search on modern CPUs and GPUs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 339–350. ACM, 2010.

[4] B. Lippmeier and G. Keller. Efficient parallel stencil convolution in Haskell. In *Proceedings of the 4th ACM symposium on Haskell*, Haskell '11, pages 59–70, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0860-1.

[5] B. Lippmeier, M. Chakravarty, G. Keller, and S. Peyton Jones. Guiding parallel array fusion with indexed types. In *Proceedings of the 2012 symposium on Haskell symposium*, Haskell '12, pages 25–36, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1574-6.

[6] H. Liu, N. Glew, L. Petersen, and T. Anderson. The Intel Labs Haskell research compiler. Submitted for publication, June 2013.

[7] L. Petersen and N. Glew. GC-safe interprocedural unboxing. In *Proceedings of the 21st international conference on Compiler Construction*, CC'12, pages 165–184, Berlin, Heidelberg, 2012. Springer-Verlag.

[8] L. Petersen, D. Orchard, and N. Glew. Automatic SIMD vectorization for Haskell. Submitted for publication, April 2013.

[9] N. Satish, C. Kim, J. Chhugani, H. Saito, R. Krishnaiyer, M. Smelyanskiy, M. Girkar, and P. Dubey. Can traditional programming bridge the ninja performance gap for parallel computing applications? In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 440–451, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-1-4503-1642-2.