SANDIA REPORT
SAND2009-6093
Unlimited Release
Printed September 2009

Parallelism of the SANDstorm Hash Algorithm

Tim Draelos, Rich Schroeppel, and Mark Torgerson

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd.
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2009-6093
Unlimited Release
Printed September 2009

Parallelism of the SANDstorm Hash Algorithm

Timothy Draelos, Richard Schroeppel, and Mark Torgerson
Analytics and Cryptography Department

Sandia National Laboratories
P. O. Box 5800
Albuquerque, NM 87185-0672

Abstract

Mainstream cryptographic hashing algorithms are not parallelizable. This limits their speed and they are not able to take advantage of the current trend of being run on multi-core platforms. Being limited in speed limits their usefulness as an authentication mechanism in secure communications. Sandia researchers have created a new cryptographic hashing algorithm, SANDstorm, which was specifically designed to take advantage of multi-core processing and be parallelizable on a wide range of platforms. This report describes a late-start LDRD effort to verify the parallelizability claims of the SANDstorm designers. We have shown, with operating code and bench testing, that the SANDstorm algorithm may be trivially parallelized on a wide range of hardware platforms. Implementations using OpenMP demonstrates a linear speedup with multiple cores. We have also shown significant performance gains with optimized C code and the use of assembly instructions to exploit particular platform capabilities.

Acknowledgements

We would like to thank Kevin Pedretti (1423) for his advice and expertise with OpenMP, for porting our test code to his hardware platforms, and for running performance tests.

Contents

Figures

Tables

1 Introduction

Cryptographic hashing is the workhorse of authentication services provided by cryptography and is required in almost all secure data transactions. The speed in which data authentication can be done plays a significant role in the viability of high-speed, secure communications. Since the clock speed of microprocessors has leveled off in recent years, the speed in which the authentication mechanisms can be accomplished is also limited. The path to increased computing performance is with multiple processors (cores) running simultaneously. Unfortunately, current mainstream cryptographic hashing algorithms do not support parallelization and thus are not able to take advantage of the current hardware trends. The Sandia-designed cryptographic hashing algorithm, SANDstorm [1], was designed to take advantage of current and planned multi-core hardware. The algorithm has several different novel features, each of which allows significant parallelization.

The purpose of this LDRD project was three-fold. The first effort was to verify empirically the claims of the SANDstorm designers (i.e., to show that the algorithm may be parallelized). The second was to measure the difficulty and the performance of parallelization. Lastly, the LDRD project focused on efforts to speed the algorithm by high performance implementations of core components in optimized C and assembly languages.

SANDstorm takes as input a message of arbitrary length, cryptographically hashes the message, and outputs a message digest of size 224, 256, 384, or 512 bits in length. SANDstorm-224 and SANDstorm-256 perform operations on 64-bit words while SANDstorm-384 and SANDstorm-512 perform the same operations using 128-bit words. Software using 64-bit words will perform well on 64-bit hardware if compiled with a 64-bit compiler. However, software using 128-bit words will require the use of special, extended instructions (e.g. SSEx) to perform optimally. The SANDstorm algorithm also supports pipelining of the round function due to the data chaining. Verifying the pipelinability was beyond the small amount of funds provided in this small late start LDRD, so we did not verify those claims.

This LDRD project has shown that, by far, the largest speed payoff comes from taking advantage of the parallelism designed into the algorithm. In summary, there is a linear increase in speed with the addition of computing resources. The difficulty of attaining those speed increases is minimal with the instructions found in the OpenMP parallel processing library.

Our optimizations using standard serial programming have been shown to be substantial, but of course they cannot compete with the linear speed-ups associated with parallelization. Our best implementations use both optimized C and/or assembly language with parallelization.

The report is structured as follows: Section 2 gives a very brief description of the SANDstorm algorithm needed to explain the various optimization efforts. Section 3 then discusses programming details, Section 4 the results, and Section 5 concludes the report.

2 The SANDstorm Algorithm

Figures 1 and 2 give enough of the SANDstorm details to indicate where our programming efforts have been applied. Figure 1 is a representation of the mode, which is the structure from the highest level in which the message data is processed. The data message to be hashed is processed by Levels 0 and 1. Levels 2-4 take as input the hash outputs from the previous level. Figure 2 is a closer look at the compression function. In particular, it focuses on the chaining of data blocks, the input of the message schedule, and the round functions.

2.1 The mode

The mode (Figure 1) was designed so that different levels of the mode and different sections within levels can be processed independently. The final amount of parallelization available in the mode is a function of the length of the message. For very long messages one may see a maximum of 1000-times speedup over serial processing. This speedup would require about 1100 times the serial processing hardware resources. For message less than one block in length, only Levels 0 and 4 are required. For messages less than 11 blocks in length, only Levels 0, 1, and 4 are required. A message of at least 21 blocks in length is necessary to take advantage of Mode parallelization.

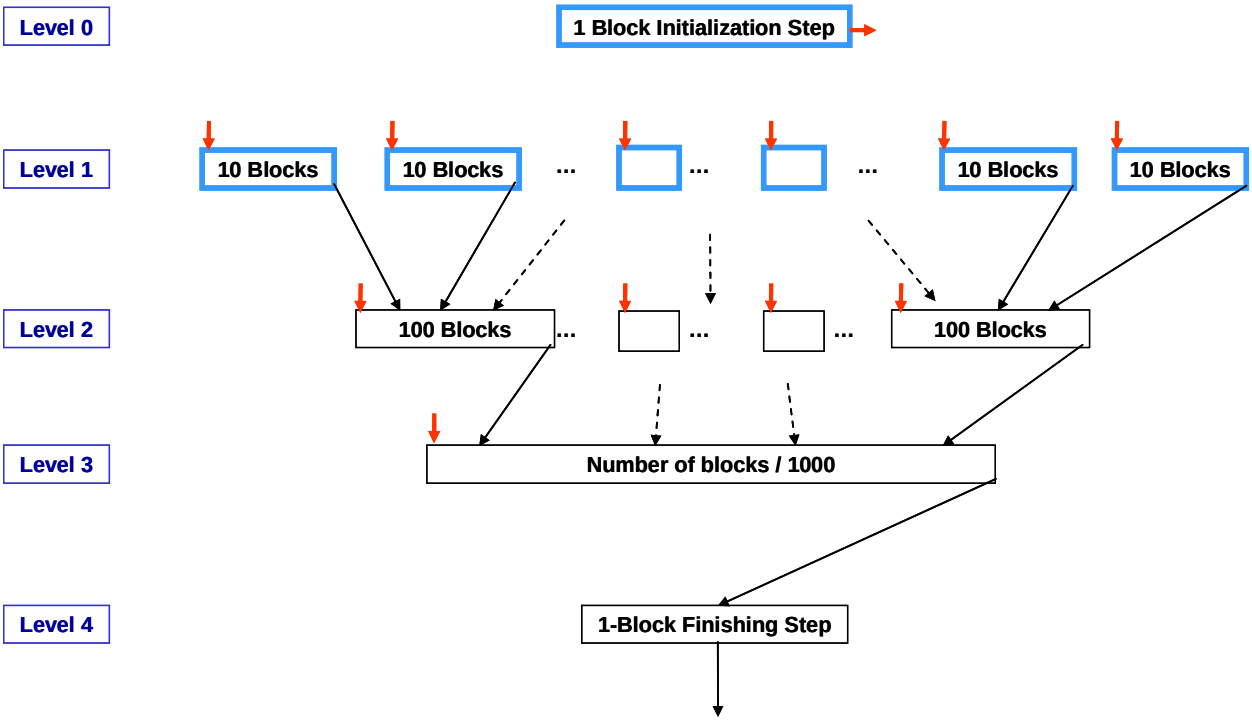

Figure 1 The SANDstorm hash algorithm Mode.

Once Level 0 has been completed, each rectangle in Levels 1 and 2 can be processed in parallel. Level 3 is fully serial and so caps the amount of parallelization that is available. (However, this truncated tree mode also bounds the latency over a full tree). Level 0 and Level 4 are the initialization and finishing step, respectively. For very long messages, these computations may be amortized away. In shorter messages, they do have a cost.

The speedup factor of 1000 comes from the fact that each block that comes into Level 3 represents 1000 message blocks. It is not that those 1000 message blocks may be processed independently – indeed they cannot because each Level 1 superblock processes 10 blocks at a time. However, 1000 Level 1 superblocks may be processed independently and fed into 100 Level 2 superblocks. If they are output in the right sequence, when the first Level 2 superblock finishes it may begin processing the 101st superblock, and so on. Enough hardware must be in place to keep this going. We need to process 1000 Level 1 superblocks, 100 Level 2 superblocks and 1 block at Level 3. Thus, a total of 1101 times the resources needed to process a single block of data.

2.2 Chaining, the Compression Function

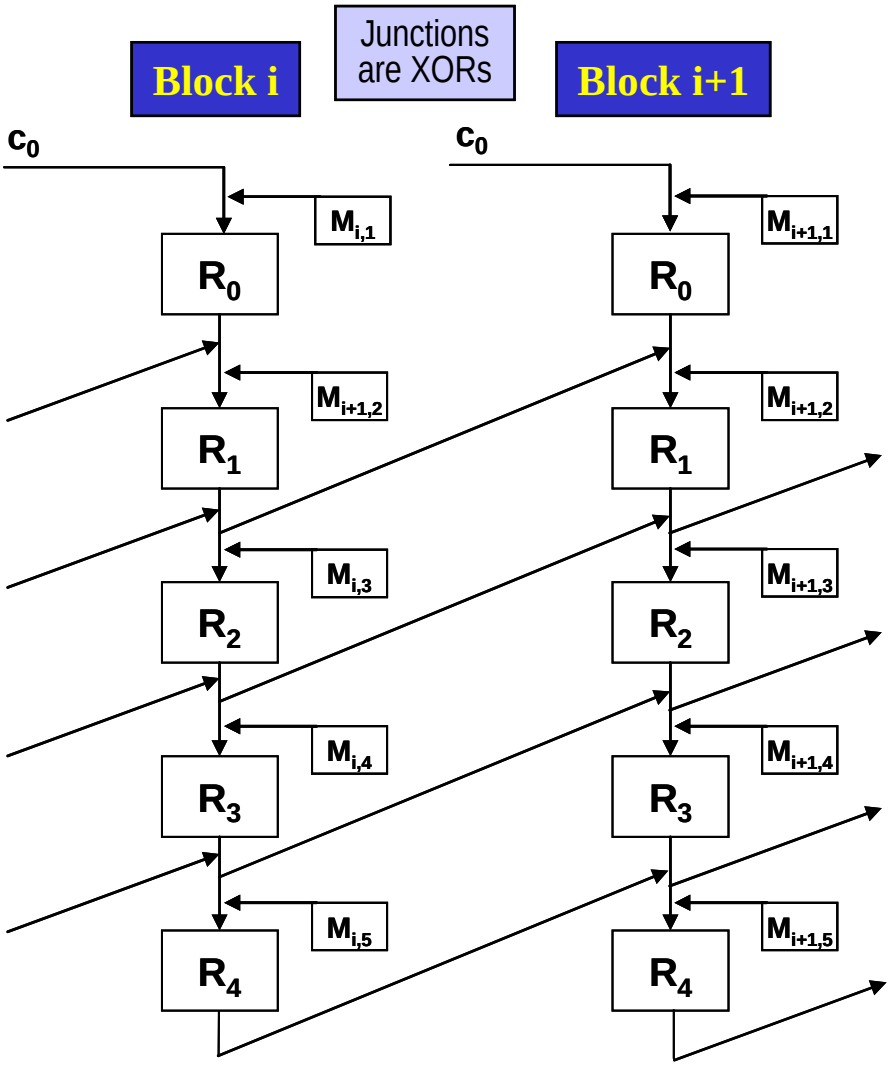

Figure 2 The SANDstorm hash algorithm Compression Function and Chaining mechanism.

The first thing to note in the compression function is that the message schedule (outputs represented by M in Figure 2) and the round functions (represented by R in Figure 2) may be computed independently (c_0 is a constant). The SANDstorm design was such that the cost to compute the round function and the message schedule is about equal. Strictly speaking, there is about 45:55 ratio between the two, but there is more data organization/manipulation in the round processing, so the actual work is split fairly evenly between the two. If there are two independent processors available, the message schedule and the round functions may be computed at the same time, giving a factor of two speedup.

There are four chaining variables. During the processing of a single data block, they are processed sequentially. They are chained forward and used during the computation of the next block and so on. This setup allows for pipelining between data blocks and may add a speed factor of up to four. Pipelining in this fashion is reasonable to do in hardware, but is not amenable to multicore processing in software, and so we did not attempt to verify this feature.

Not shown here is a more detailed description of the computations in the round function and the message schedule. They were designed to allow certain operations to be completed simultaneously. This is where the optimized C and assembly methods are best brought to bear.

3 Programming details

There are multiple ways to increase the performance of the SANDstorm algorithm relative to the original functional implementation. The biggest payoff with large messages is leveraging the parallel processing features of the algorithm. For a message of any size, improving the performance of the compression function (message schedule and round function) is crucial. This improvement is possible with optimized C and even more with assembly language targeting certain microprocessor instructions not available in the standard C language.

3.1 Parallelism

Parallel processing is dependent on a suitable algorithm, suitable hardware, and suitable software for computing elements of the computational work simultaneously. As noted in Section 2, the SANDstorm algorithm offers several opportunities for parallel process of hashing operations. It is difficult to buy desktop computers that do not have multiple processing cores. Finally, there is compiler and operating system support for programming an algorithm to perform parallel processing.

We utilized OpenMP [2], possibly the simplest avenue to achieve parallel processing of SANDstorm on common desktop computers running Linux and Microsoft Windows operating systems. OpenMP is an application programming interface (API) specification for parallel processing in the C and C++ languages. It is supported by the Microsoft Visual Studio and GNU compilers and is relatively easy to use since it relies on the underlying operating system to interact with the hardware and manage threads. The most amount of work involved in taking advantage of OpenMP in our case was restructuring our software to utilize loops. Since the length of the message to be hashed is not necessarily known when hashing begins, many implementations do not use a loop structure. Once this structure exists the amount of additional OpenMP code is trivial. Figure 1 shows C code employing OpenMP to parallel process the SANDstorm mode (Level 1 followed by Level 2).

The following list of activities were performed to restructure the original C implementation of the SANDstorm algorithm.

- Include omp.h in any C file utilizing OpenMP functions.
- Structure the hashing of input data to use *for* loops.
- Minimize the use of shared data. Each processor (core) needs its own set of variables and must properly index the shared input buffer and output buffer.
- Add the following OpenMP pragma statement prior to the *for* loop that is to be parallel processed.

  ```
  #pragma omp parallel for
  ```

The following actions are necessary to utilize OpenMP in the two listed C compilers.

Visual Studio: Enable OpenMP with the following integrated development environment setting.
Configuration Properties → C/C++ → Language → OpenMP Support = Yes

GCC: Add the following switch to the compiler command line.

```
-fopenmp
```

```
#include <omp.h>

// Check for enough data to parallelize L1 superblock processing (>= 20 blocks)
if (…) {
      …
#pragma omp parallel for
      for (k = 0; k < nL1superblocks; k++) {
            int h, i;
            u64 dataIndex;
            u64 MSd[33];                        // Message schedule buffer
            u64 prevBlock[5][4];                // Hash state
            …
            for (h = 0; h < 10; h++) {
                  PREPARE_DATA (MSd, data, dataIndex);
                  Compress (MSd, prevBlock);    // Compress message block
                  dataIndex += MAXBYTES;
            }
            …
      }
      …
#pragma omp parallel for
      for (k = 0; k < nL2superblocks; k++) {
            …
            for (h = 0; h < 100; h++) {     …     }
            …
      }
}
```

Figure 3 C code for parallel processing of the SANDstorm mode using OpenMP.

3.2 C

The primary implementation technique used to improve the speed of the C implementation of SANDstorm was the elimination of function calls. Each call to a C function involves saving the state (registers) of the processor, establishing new variables for use by the function, and restoring the original state of the processor. If a function is called many times, the expense of these operations can be significant. An alternative approach is to use macros instead of functions. Macros may look like functions from the standpoint of readability, but the compiler treats them as direct substitutions of C statements, resulting in inline code.

Another opportunity for speed enhancements of C code is to use intrinsic functions. These function lie somewhere between C and assembly language and are not part of the standard C language. They are callable from C and allow the program to take advantage of hardware attributes of specific processors. In the case of SANDstorm, there is an intrinsic function that takes two 64-bit operands and returns the 128-bit product of them, which greatly improves the speed of SANDstorm-512. The C programming language has no rotate operator and generally requires two shift operations and a bit-wise OR operation instead. However, most microprocessors have left and right rotate instructions. SANDstorm makes use of the rotate operation and benefits from the rotate intrinsic function.

3.3 Assembly Language

Using x86 assembly language is the lowest level of implementation that we utilized for parallelism in SANDstorm. The SANDstorm design allows for parallelism at many levels. One part of the design is that the Message Schedule processes the input message into intermediate results that are passed to the Round Function. About 60% of the work of SANDstorm takes place in the Message Schedule. This work can be overlapped with the work of the Round Function, and leads to roughly a factor of two speedup.

The Intel x86 architecture has evolved considerably over the years. In the current incarnation (early 2009), each core of the processor has three sets of registers for working with data. These registers are of different lengths (some 64 bits, some 128 bits) and use different instructions for processing. The core contains several processing engines for working with the different register sets, so several operations can be accomplished simultaneously, if they are using different register sets. (The processor also contains a lot of logic to make sure that the simultaneous operations are non-interfering, and can be serialized to match the programmer's implied serial intent.)

We have organized the assembly language code to use one set of registers for the Message Schedule processing, and a second set for the Round Function. The third set of registers is used for interactions between the Message Schedule and the Round Function. The SANDstorm Message Schedule produces five inputs to the Round Function, one input for each iteration of the Round Function. The first input is relatively simple, and is produced quickly, allowing the Round Function to get started on Round 0. After this, the Message Schedule and Round Function compute simultaneously.

The instructions for the Message Schedule and Round Function are interleaved (in the instruction stream in memory), and the processor sends them to the different arithmetic engines that work with each register set. This case is practically a poster child for flexible design: The Intel x86 architecture is an ad hoc collection of registers and instructions, evolved over the years for whatever tasks were important for each year's marketing effort. And yet, the SANDstorm design is able to make effective use of the hodgepodge, getting a speedup of roughly a factor of two over serial processing, and using a large fraction of the available raw computing power.

4 Results

For the international hash competition, the National Institute of Standards and Technology established a reference platform on which to test candidate algorithms. This platform consists of an Intel Core 2 Duo 64-bit microprocessor running at 2.6 GHz and the Windows Vista (32-bit and 64-bit) operating system (OS). Table 1 shows performance results of the SANDstorm algorithm hashing large messages on this machine.

Software Implementation	Speedups	Compiler	Speed (Cycles/byte)
SANDstorm-256 compression function	Assembly	64-bit	13
SANDstorm-256 hash (serial)	Assembly	64-bit	15
SANDstorm-256 hash (parallel)	Assembly	64-bit	8
Original SANDstorm-512 compression function		64-bit	90
SANDstorm-512 compression function	Macros	64-bit	49
SANDstorm-512 compression function	Macros Intrinsics	64-bit	24
SANDstorm-512 hash (serial)	Macros Intrinsics	64-bit	27
SANDstorm-512 hash (parallel)	Macros Intrinsics	64-bit	14

Table 1 Performance of SANDstorm on dual-core NIST reference platform.

Table 1 illustrates the speedup afforded by the parallelized implementation with and without assembly language. Both the serial and parallel implementations incorporated optimized C code. The hardware consisted of 64-bit dual quad-core PCs running 64-bit Windows Vista and 64-bit Ubuntu Linux. The parallel code is nearly eight times as fast as the serial code. On a dual-core process, the parallel code is twice as fast. As the number cores increases, the size of the message must increase as well in order to take full advantage of the hardware.

Software Implementation	Speed (Cycles/byte)
Serial	18.5
OpenMP Parallel	2.4
OpenMP Parallel with 64-bit Assembly Code	2.1

Table 2 Performance of SANDstorm-256 on dual quad-core microprocessors.

Figure 4 illustrates the relationship of message size and speedup on dual quad-core hardware. It is clear that it takes a message of approximately 100,000 blocks (64 Mbytes) to achieve an 8-times speedup.

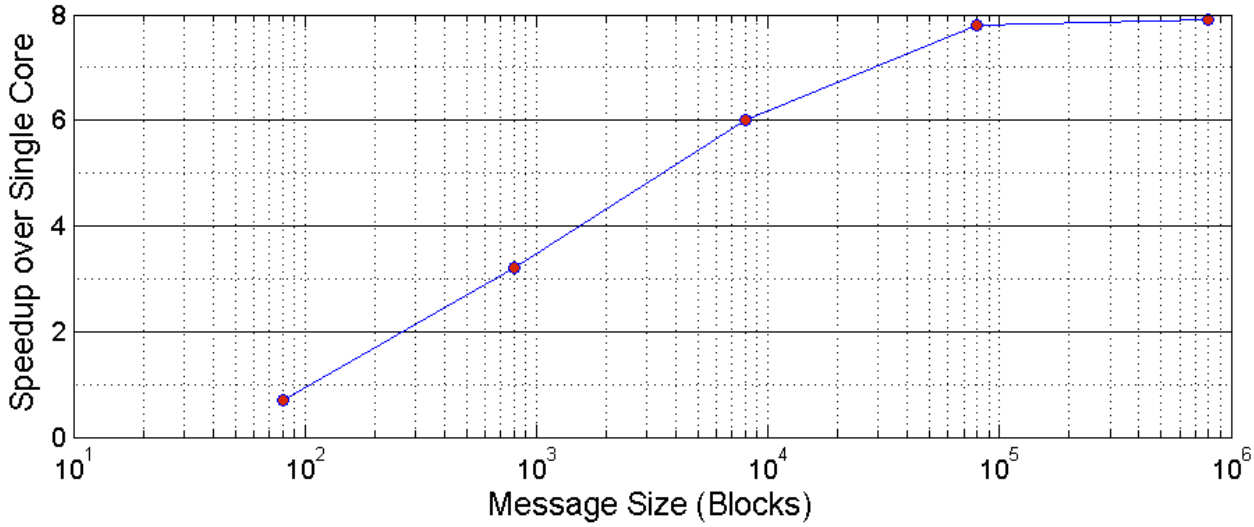

Figure 4 SANDstorm-256 parallel processing speedup as a function of message size on eight cores.

Empirical results prove that the SANDstorm algorithm allows parallel hashing to be performed eight times faster with eight cores than hashing sequentially. Figures 5-8 show how the algorithm performs on processors with more cores. The UltraSPARC T2 microprocessor is Sun Microsystems' 2nd generation of the Niagara platform. It is a multithreading, multicore microprocessor with up to 8 cores per chip, up to 64 threads per CPU, and huge memory capacity. SANDstorm performance was tested on a 64-core Sun Sparc system with 2 8-core Niagara processors. The Intel Xeon 5400 series microprocessors (also known as Nehalem), supports up to 4 cores and two threads per core with hyperthreading enabled. SANDstorm performance was tested on hardware with 2 4-core Nehalem processors. The key difference between these two multicore processors is that the Niagara contains more cores and allows more threads, but each core is relatively simple and slow whereas the Nehalem processor contains fewer, but larger and more powerful cores.

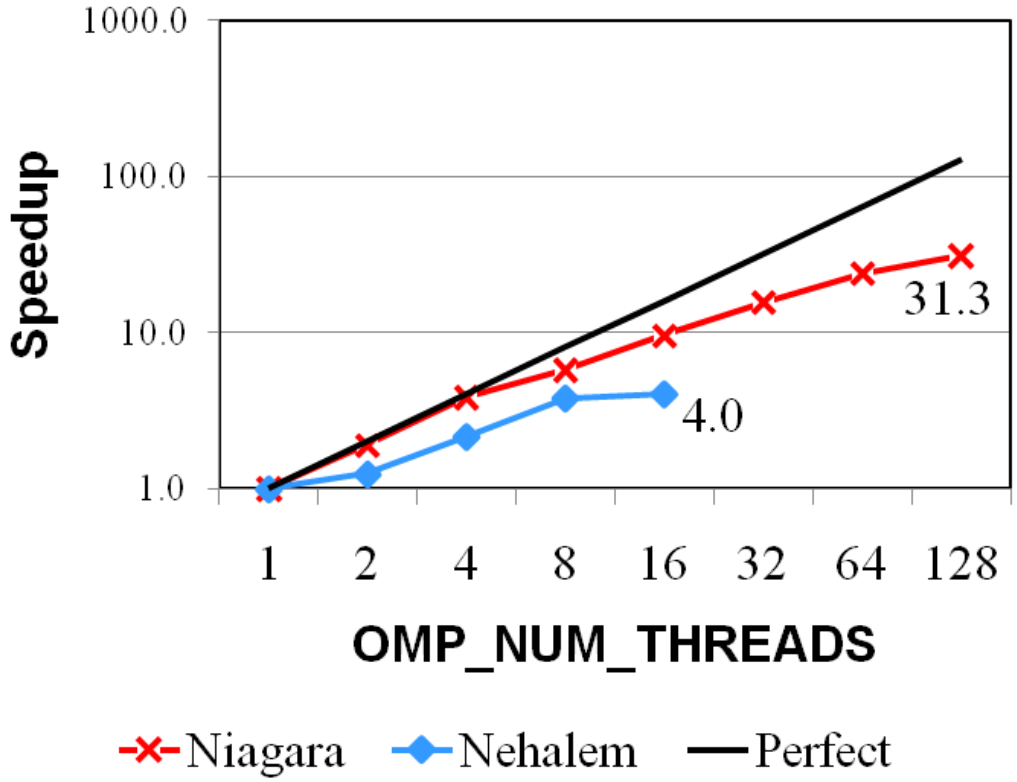

Figure 5 SANDstorm-256 speedup vs. Threads using OpenMP on a 50 MB message.

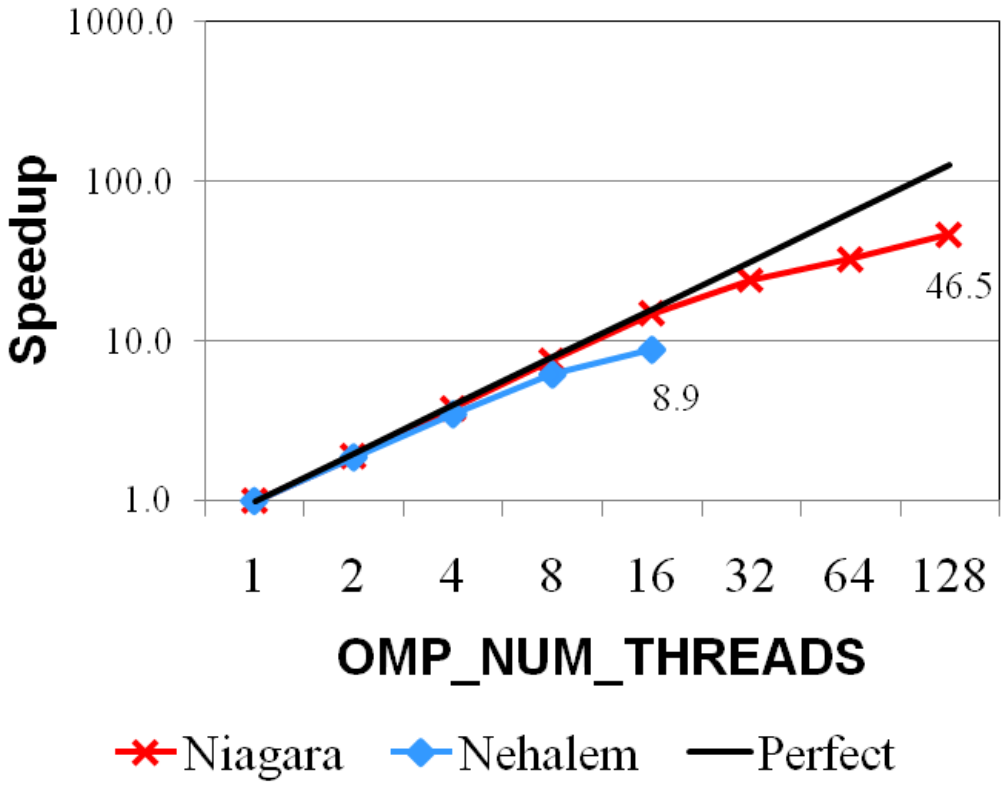

Figure 6 SANDstorm-256 speedup vs. Threads using OpenMP on a 500 MB message.

Figure 6 shows that scaling improves with a larger message. Figures 7 and 8 shows that the Nehalem processor has better absolute performance (13-47%) than the Niagara processor, although the Niagara scales better (Figures 5 and 6).

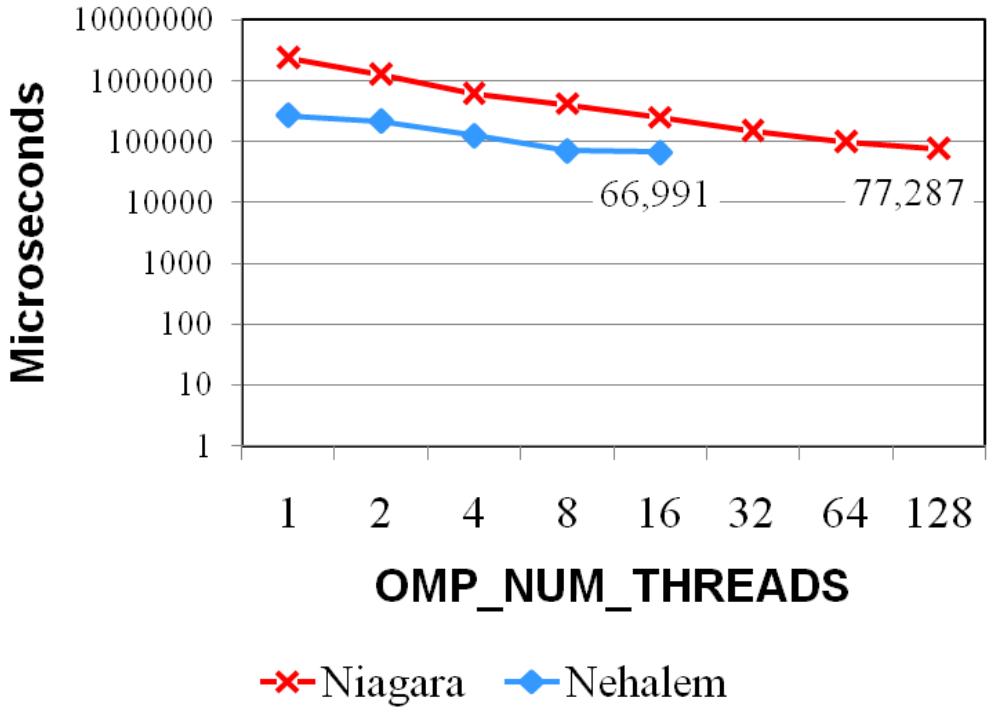

Figure 7 SANDstorm-256 absolute speed vs. Threads using OpenMP on a 50 MB message.

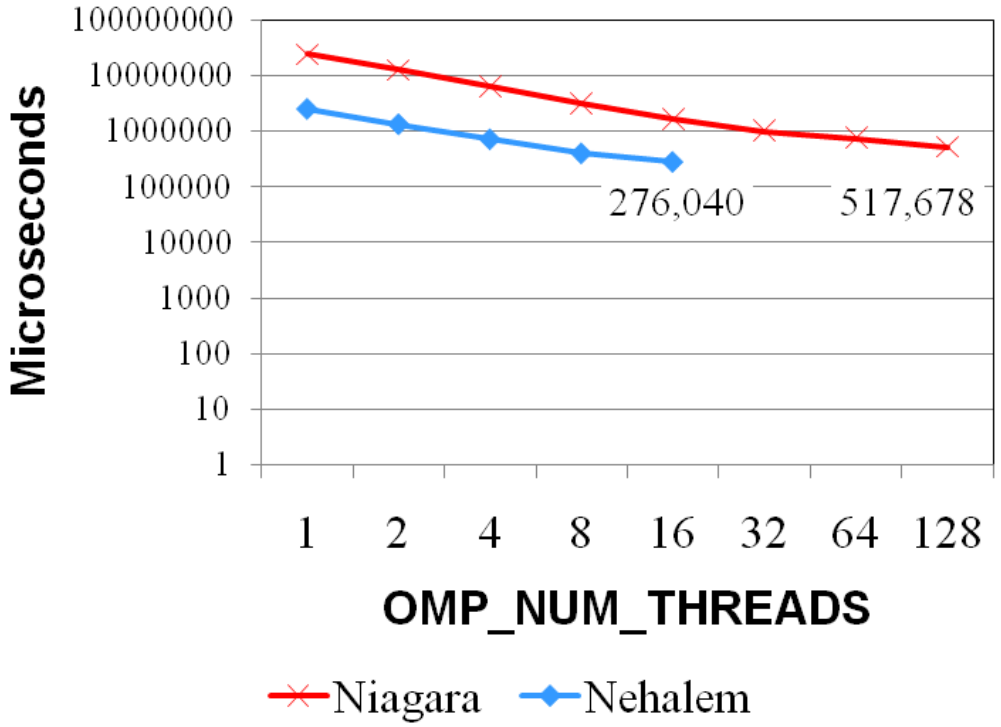

Figure 8 SANDstorm-256 absolute speed vs. Threads using OpenMP on a 500 MB message.

5 Conclusions

The SANDstorm cryptographic hashing algorithm was designed to allow a high degree of freedom for various kinds of parallelism. The intent was to provide multiple avenues that were not tied to any particular hardware architecture. The work in this LDRD has confirmed that many of the design goals were met.

We have shown that the mode may be parallelized by producing actual code and comparing the timings with equivalent serial implementations on several different hardware platforms. We have also shown that with the OpenMP library, exploiting the parallelization in the mode is simple and straight forward, requiring only a few lines of code, and results demonstrating a linear speedup with multiple cores.

We have shown that in the compression function, the round function work and the message schedule work can be separated. Using SSE instructions and redundant registers we were able to parallelize those two operations.

We have shown a number of assembly and C coding techniques targeting the computational engine of SANDstorm, the compression function. Together with code optimizations and parallel implementations, we have shown that the SANDstorm algorithm is very efficient and may be parallelized up to an amount beyond currently reasonable hardware bounds.

6 References

[1] M. Torgerson, R. Schroeppel, T. Draelos, N. Dautenhahn, S. Malone, A. Walker, M. Collins, H. Orman, The SANDstorm Hash , NIST Hash Submission, October 2008.

[2] The OpenMP API specification for parallel programming, http://openmp.org/wp/.

DISTRIBUTION:

1 MS 0899 Technical Library (electronic copy)