PeachPy: A Python Framework for Developing High-Performance Assembly Kernels

Marat Dukhan
School of Computational Science & Engineering
Georgia Institute of Technology, Atlanta, GA, USA

***Abstract*—We introduce PeachPy, a Python framework which aids the development of assembly kernels for high-performance computing. PeachPy automates several routine tasks in assembly programming such as allocating registers and adapting functions to different calling conventions. By representing assembly instructions and registers as Python objects, PeachPy enables developers to use Python for assembly metaprogramming, and thus provides a modern alternative to traditional macro processors in assembly programming. The current version of PeachPy supports x86-64 and ARM architectures.**

I. Introduction

We consider the problem of how to enable *productive* assembly language programming. The use of assembly still plays an important role in developing performance-critical computational kernels in high-performance computing. For instance, recent studies have shown how the performance of many computations in dense linear algebra depend critically on a relatively small number of highly tuned implementations of microkernel code [1], for which high-level compilers produce only suboptimal implementations [2]. In cases like these, manual low-level programming may be the only option. Unfortunately, existing mainstream tools for assembly-level programming are still very primitive, being tedious and time-consuming to use, compared to higher-level programming models and languages.

Our goal is to ease assembly programming. In particular, we wish to enable an assembly programmer to build high-performing code for a variety of operations (i.e., not just for linear algebra), data types, and processors, and to do so using a relatively small amount of assembly code, combined with easy-to-use metaprogramming facilities and nominal automation for routine tasks *provided* they do not hurt performance. Toward this end, we are developing PeachPy, a new Python-based framework that aids the development of high-performance assembly kernels.

PeachPy joins a vast pool of tools that blend Python and code generation. Code-generation is used by Python programs for varying reasons and use-cases. Some projects, such as PyPy [3] and Cython [4], use code-generation to improve the performance of Python code itself. Cython achieves this goal by statically compiling Python sources to machine codes (via C compiler) and providing a syntax to specify types of Python variables. PyPy instead relies on JIT-compilation and type inference. Another group of code generation tools is comprised of Python bindings to widely used general-purpose code-generation targets. This groups includes LLVM-Py [5], PyCUDA and PyOpenCL [6] projects. In essence, these examples focus on accelerating Python through low-level code generation.

By contrast, we are interested primarily in the task of generating assembly code using Python. This style of Python-assisted assembly programming was pioneered by CorePy [7], the works of Malas et al. [8], Dongarra and Luszczek [9], and several other projects [10, 11]. CorePy [7] enabled developers a programmer to write an assembly program in Python and compile it from the Python interpreter. The authors suggested using CorePy to optimize performance-sensitive parts of Python application. Malas et al. [8] used Python to auto-tune assembly for PowerPC 450 processors powering Blue Gene/P supercomputers. Their assembly framework could simulate the processor pipeline and reschedule instructions to avoid data hazards and improve performance. The assembly code-generator of Dongarra and Luszczek [9] focuses on an assembly programmer productivity, and is the most similar in spirit to PeachPy. The primary use-case for their code-generator is cross-compilation to ARM, but they also supported a debug mode, where the code-generator outputs equivalent C code.

The audience for PeachPy is an optimization expert writing multiple but similar kernels in as-

PyHPC 2013, November 18, 2013, Denver, Colorado, USA

sembly, as might happen when implementing the same operation in both double and single precision or for multiple architectures. PeachPy further aids the programmer by automating a number of essential assembly programming tasks; however, it does so only in instances when there will be no unintended losses in performance. To achieve these design goals, PeachPy has a unique set of features:

- Like prior efforts [7–9], PeachPy represents assembly instructions, registers, and other operands as first-class Python objects.
- PeachPy's syntax closely resembles traditional assembly language.[1] Most instructions can be converted to calls to equivalent PeachPy functions by just enclosing their operands in parentheses.
- PeachPy enriches assembly programming with some features of compilers for higher-level languages. PeachPy performs liveness analysis on assembly functions, does fully automatic register allocation,[2] adapts program to different calling conventions, and coalesces equal constants in memory.
- PeachPy collects information about the instruction sets used in the program. This information enables the implementation of dynamic dispatching between several code versions based on instruction extensions available on a given host processor.
- PeachPy aims to replace traditional assembly language programming models. Furthermore, PeachPy-generated code does not impose any requirements on runtime libraries and does not need Python to run. PeachPy supports a wide variety of instruction set architectures, including various versions of SSE, AVX and AVX2, FMA3 and FMA4 on x86 and NEON on ARM.

A. Contributions

PeachPy brings two improvements to the process of developing assembly kernels:

1) **Advancing Python-based assembly metaprogramming** (§ III): PeachPy represents assembly instructions, registers, and constants as first-class Python functions and objects. As such, PeachPy enables Python to serve as a modern alternative to traditional assembly macro programming. The use of Python-based metaprogramming lets an optimization expert generate compute kernels for different data types, operations, instruction sets, microarchitectures, and tuning parameters from a single source. While earlier works pioneered the use of Python for assembly metaprogramming [7–9], PeachPy goes further with new tools and support for a wider range of use-cases. For instance, PeachPy makes it easier to write software pipelined code and to emulate newer instruction sets using older instructions. Another important PeachPy-enabled use-case is creating fat binaries with versions for multiple instruction sets or microarchitectures.
2) **Automation of routine tasks in assembly** (§ II): PeachPy fully automates some tasks in assembly programming, but only where such automation is not likely to impact performance. For example, PeachPy handles register allocation because on modern out-of-order processors the choice of physical registers to use for virtual registers does not affect performance. However, unlike compilers for high-level languages, if PeachPy finds that it cannot fit all virtual registers onto physical registers without spilling, it will not silently spill registers on local variables, which might degrade performance; rather, it will alert the programmer by generating a Python exception.

B. PeachPy DSL

PeachPy contains two Python modules, `peachpy.x64` and `peachpy.arm`, which implement assembly instructions and registers for x86-64 and ARM architectures, respectively, as Python objects. These modules are intended to be used with the `from peachpy.arch import *` statement. When imported this way, PeachPy objects let the programmer write assembly in Python with syntax similar to traditional assembly, effectively implementing an assembly-like domain-specific language (DSL) on the top of Python..

Assembly functions are represented by a `Function` class in peachpy modules. The function constructor accepts four parameters that specify the assembler object (storage for functions), function name, tuple of function arguments, and the string name of a target microarchitecture. The latter restricts the set of instructions which can be used: an attempt to use an instruction unsupported on

[1] Malas et al [8] followed the same principle in their PowerPC assembler

[2] Most previous research for assembly code generation used some kind of register management mechanism. However, it required the programmer to explicitly allocate and release registers. PeachPy allocates registers automatically based on liveness analysis.

the target microarchitecture will cause throw an exception. Once a function is created, it can be made active by the `with` statement. Instructions generated in the scope of `with` statement are added to the active function. When the execution of the Python script leaves the scope of the `with` statement, PeachPy will run the post-processing analysis passes and generate final assembly code.

Listing 1 Minimal PeachPy example

```
from peachpy.x64 import *

abi = peachpy.c.ABI('x64-sysv')
assembler = Assembler(abi)
x_argument = peachpy.c.Parameter("x",
    peachpy.c.Type("uint32_t"))
arguments = (x_argument)
function_name = "f"
microarchitecture = "SandyBridge"

with Function(assembler, function_name,
    arguments, microarchitecture):
    MOV( eax, 0 )
    RETURN()

print assembler
```

II. Automation of routine tasks

By design, PeachPy only includes automation of assembly programming tasks that will not adversely affect the efficiency of the generated code. Any automating choices which might affect performance are left to the programmer. This design differs from the philosophy of higher-level programming models, where the compiler must correctly handle all situations, even if doing so might result in suboptimal performance. PeachPy opts for relatively less automation, in part because we expect it will be used for codes where the high-automation approach of high-level compilers does not deliver good performance.

A. Register allocation

While PeachPy allows referencing of registers using their standard names (e.g., rcx or xmm0 on x86), it also provides a *virtual* register abstraction. The programmer creates a virtual register by calling constructors without parameters for the register classes. Virtual registers can be used as instruction operands in the same way as physical registers. After PeachPy receives all instructions that constitute an assembly function, it performs a liveness analysis for virtual registers and uses it to bind virtual registers to physical registers. If it cannot perform this binding without spilling, it will generate a Python exception to alert the programmer about the situation. The programmer may then rewrite the function to use fewer virtual registers or manually spill virtual registers to local variables.

B. Constant allocation

Code and data are physically stored in different sections of the executable, and in assembly programming they are defined and implemented in different parts of the source file. The code where an in-memory constant is used might be far from the line where the constant is defined. Thus, the programmer has to always keep in mind the names of all constants used in a function. PeachPy solves this problem by allowing constants to be defined exactly where they are used. When PeachPy finalizes an assembly function, it scans all instructions for such constants, coalesces equal constants (it can coalesce integer and floating-point constants with the same binary representation), and generates a constants section.

C. Adaptation to calling conventions

Both x86-64 and ARM architectures can be used with several variants of function calling conventions. They might differ in the how the parameters are passed to a function or which registers must be saved in the function's prolog and restored in the epilog. For assembly programmers, supporting multiple calling conventions requires having several versions of assembly code. While these versions are mostly similar, maintaining them separately can quickly become a significant burden.

To assist adaptation of function to different calling conventions, PeachPy provides a pseudo-instruction, `LOAD.PARAMETER` which loads a function parameter into a register. If the parameter is passed in a register and the destination operand of the `LOAD.PARAMETER` pseudo-instruction is a virtual register, PeachPy will bind this virtual register to the physical register where the parameter is passed, and the `LOAD.PARAMETER` instruction will become a no-op.

III. Metaprogramming

One of the goals of PeachPy project was to simplify writing multiple similar compute kernels in assembly. To achieve this goal, PeachPy leverages flexibility of Python to replace macro preprocessors in traditional assemblers.

A. Custom Instructions

PeachPy users can define Python functions that will be used interchangeably with real assembly instructions, just like macros in traditional assembly can be used similar to instructions. Unlike macros, Python-based PeachPy functions that implement an instruction can use virtual registers to hold temporaries without making them part of the interface. Listing 2 shows how the SSE3 instruction `HADDPD` can be simulated using SSE2 instructions in PeachPy. The interface of the simulated `HADDPD` instruction exactly matches the interface of the real `HADDPD`, and can be transparently used by compute kernels which need this instruction.

Listing 2 Simulation of HADDPD with x86 SSE2 instructions

```
def HADDPD(xmm_dst, xmm_src):
    xmm_tmp = SSERegister()
    MOVAPD( xmm_tmp, xmm_dst )
    UNPACKLPD( xmm_dst, xmm_src )
    UNPACKHPD( xmm_tmp, xmm_src )
    ADDPD( xmm_tmp, xmm_dst )
```

B. Parameterized Code Generation

With PeachPy, a programmer can use Python's control flow statements and expressions to parametrize the generation of assembly code. Then, an optimization expert might tune the parameters to maximize the performance. Common examples of such parameters in HPC kernels are loop unrolling factors and prefetch distances. An example of parameterized code generation for an array summation kernel appears in Listing 3.

Listing 3 Array summation kernel parametrized by loop unroll factor and prefetching distance using x86 AVX instructions

```
def VECTOR_SUM(unroll_regs, prefetch_distance):
    ymm_acc = [AVXRegister()
        for _ in range(unroll_regs)]
    for i in range(unroll_regs):
        VXORPD( ymm_acc[i], ymm_acc[i] )
    LABEL( "next_batch" )
    PREFETCHNTA( [arrayPtr + prefetch_distance] )
    for i in range(unroll_regs):
        VADDPD( ymm_acc[i], [arrayPtr + i * 32] )
    ADD( arrayPtr, unroll_regs * 32 )
    SUB( arrayLen, unroll_regs * 8 )
    JNZ( "next_batch" )
    # Sums all ymm_acc to scalar in ymm_acc[0]
    REDUCE.SUM( ymm_acc, peachpy.c.type("float") )
    VMOVSS( [sumPtr], ymm_acc[0].get_oword() )
```

C. Generalized Kernels

Since instructions in PeachPy are first-class Python objects, it is easy to parametrize the operation or data type of a compute kernel to enable generating multiple similar kernels from a single generalized kernel. Listing 4 gives an example of one such generalized kernel.

Listing 4 Generalized kernel which generates addition and subtraction for 32-bit and 64-bit integer arrays

```
def VECTOR_OP(operation, data_size):
    SIMD_OP = {('Add', 4): VPADDD,
               ('Add', 8): VPADDQ,
               ('Sub', 4): VPSUBD,
               ('Sub', 8): VPSUBQ}
               [(operation, data_size)]
    LABEL( "next_batch" )
    xmm_x = SSERegister()
    xmm_y = SSERegister()
    VMOVDQU( xmm_x, [xPtr] )
    VMOVDQU( xmm_y, [yPtr] )
    SIMD_OP( xmm_x, xmm_x, xmm_y )
    VMOVDQU( [zPtr], xmm_x )
    for ptr in [xPtr, yPtr, zPtr]:
        ADD( ptr, 16 )
    SUB( length, 16 / data_size )
    JNZ( "next_batch" )
```

D. ISA-Specific Code Generation

A PeachPy user must specify the target microarchitecture when creating a Function object. This information is provided back to PeachPy kernels via static methods of the `Target` class. `Target.has_<isa-extension>` methods indicate if the target microarchitecture supports various ISA extensions, such as SSE4.2, AVX, FMA4, or ARM NEON. HPC kernels may use this information to benefit from new instructions without rewriting the whole kernel for each ISA level. Listing 5 shows how to make a dot product kernel use AVX, FMA3, or FMA4 instructions, depending on the target microarchitecture.

E. Instruction Streams

By default, PeachPy adds each generated instruction to the active assembly function. However, it may also make sense to generate similar instruction streams and then combine them together. One example is optimizing the kernel for the ARM Cortex-A9 microarchitecture. Cortex-A9 processors can decode two instructions per

Listing 5 Dot product kernel which can use AVX, FMA3 or FMA4 instructions

```
def DOT_PRODUCT():
    if Target.has_fma4() or Target.has_fma3():
        MULADD = VFMADDPS if Target.has_fma4()
            else VFMADD231PS
    else:
        def MULADD(ymm_x, ymm_a, ymm_b, ymm_c):
            ymm_t = AVXRegister()
            VMULPS( ymm_t, ymm_a, ymm_b )
            VADDPS( ymm_x, ymm_t, ymm_c )

    ymm_acc = AVXRegister()
    VXORPS( ymm_acc, ymm_acc )
    LABEL( "next_batch" )
    ymm_tmp = AVXRegister()
    VMOVAPS( ymm_tmp, [xPtr] )
    MULADD( ymm_acc, ymm_tmp, [yPtr], ymm_acc )
    ADD( xPtr, 32 )
    ADD( yPtr, 32 )
    SUB( length, 8 )
    JNZ( "processBatch" )
    REDUCE.SUM( [ymm_acc],
        peachpy.c.type("float") )
    VMOVSS( [resultPtr], ymm_acc.get_oword() )
```

cycle, but only one of them can be a SIMD instruction. Thus, performance may be improved by mixing scalar and vector operations. Bernstein and Schwabe [12] found that such instruction blending improves performance of Salsa20 stream cipher on the ARM Cortex-A8 microarchitecture, which has a similar limitation on instruction decoding. PeachPy instruction streams provide a mechanism to redirect generated instructions to a different sequence, and later merge sequences of instructions. When the `InstructionStream` object is used with the Python `with` statement all instructions generated in the `with` scope are added to the instruction stream instead of active function. Instructions stored in `InstructionStream` then can be added one-by-one to the active function (or active InstructionStream object) by calling the `issue` method. Listing 6 outlines the use of instruction stream object to mix scalar and vector variants of the kernel.

F. Software Pipelining

Instruction streams can also assist in developing software-pipelined versions of compute kernels. Using instruction streams, the programmer can separate similar instructions from different unrolled loop iterations into different instruction streams. Listing 7 applies this technique to a kernel which adds constant to a vector of integers.

After similar instructions are collected in instruction streams, it is often possible to shift these sequences relative to each other so that by the time any instruction is decoded its inputs are already computed, so the instruction can issue immediately. Figure 1 illustrates this principle. Making a software pipelined version of a kernel typically involves duplicating the kernel code twice, skewing the instruction columns relative to each other, and looping on the middle part of the resulting sequence.

Listing 6 Use of instruction streams to interleave scalar and vector instructions

```
scalar_stream = InstructionStream()
with scalar_stream:
    # Scalar kernel
    ...

vector_stream = InstructionStream()
with vector_stream:
    # SIMD kernel
    ...

while scalar_stream or vector_stream:
    # Mix scalar and vector instructions
    scalar_stream.issue()
    vector_stream.issue()
```

Listing 7 Use of instruction streams to separate similar instructions for software pipelining

```
instruction_columns = [InstructionStream()
    for _ in range(3)]
ymm_x = [AVXRegister for _ in range(unroll_regs)]
for i in range(unroll_regs):
    with instruction_columns[0]:
        VMOVDQU( ymm_x[i], [xPtr + i * 32] )
    with instruction_columns[1]:
        VPADDD( ymm_x[i], ymm_y )
    with instruction_columns[2]:
        VMOVDQU( [zPtr + i * 32], ymm_x[i] )
with instruction_columns[0]:
    ADD( xPtr, unroll_regs * 32 )
with instruction_columns[2]:
    ADD( zPtr, unroll_regs * 32 )
```

IV. Performance Study

Although PeachPy improves the programmability of traditional assembly, PeachPy code is harder to develop and maintain than code in C or FORTRAN. As such, we might want to check that optimization at the assembly level can at least improve performance compared to using an optimizing C compiler. Previous research provides ambiguous results: Malas et al. [8] found that, on PowerPC 450 processor, optimized assembly can deliver up to twice the performance of the best autotuned C

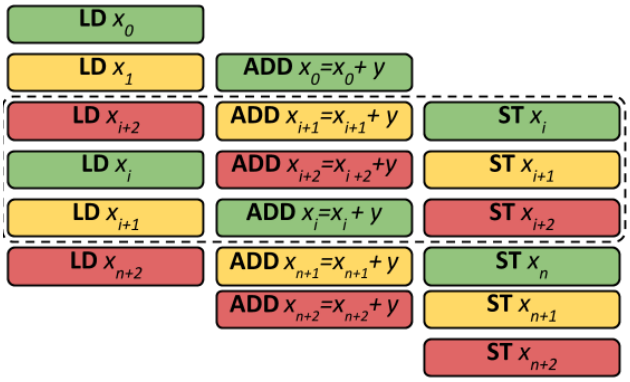

Fig. 1: Illustration of software pipelining principle for kernel in Listing 7. Instruction streams correspond to columns on this illustration. Instructions are executed in left-to-right, then top-to-bottom order.

and Fortran code; by contrast, and Satish et al. [13] suggest that code optimized with assembly or C intrinsics is just $10-40\%$ faster than C code.

In this section we describe a limited study of performance of PeachPy-generated code versus code generated by C++ compilers from equivalent C++ intrinsics. For this study we used kernels for computing vector logarithm and vector exponential functions[3] from Yeppp! library (www.yeppp.info) on an Intel Core i7-4770K (Haswell microarchitecture). A kernel takes a batch of 40 double precision elements, and computes `log` or `exp` on each element using only floating-point multiplication, addition, integer and logical operations. These kernels were originally implemented in PeachPy. For this study, we wrote an equivalent C++ version by converting each assembly instruction generated by PeachPy into an equivalent C++ intrinsic call. In the C++ source code, the intrinsics are called in exactly the same order as the corresponding assembly instructions in PeachPy code.

Several properties make this code nearly ideal for a C++ compiler:

- The code is already vectorized using intrinsic functions, so it does not depend on the quality of the compiler's auto-vectorizer.
- There is only one final branch in a loop iteration, so a loop iteration forms a basic block. In the PeachPy output, each loop iteration for `log` contains 581 instructions and the same iteration for `exp` contains 400 instructions, which gives the C++ compiler a lot of freedom to schedule instructions.
- The initial order of instructions is close to optimal, as it exactly matches the manually optimized PeachPy code. The only part left to the compiler is register allocation. If the compiler could replicate the register allocation used in the PeachPy code, it would get exactly the same performance. But the compiler could also improve upon the PeachPy implementation by finding a better instruction schedule.

Figure 2 demonstrates the performance results. None of the three tested compilers could match the performance of manually optimized assembly, although for vector logarithm gcc's output is very close. This result suggests that developing HPC codes in assembly might be necessary to get optimal performance on modern processors. In such cases, developers can leverage PeachPy to make assembly programming easier.

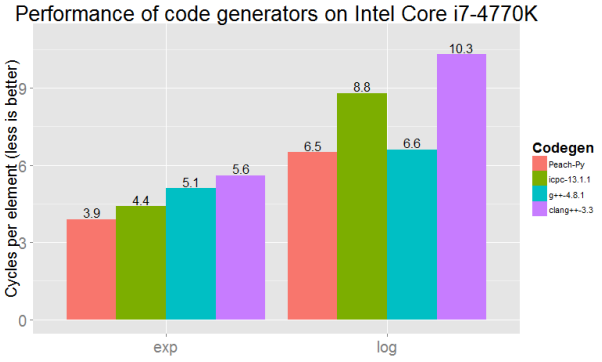

Fig. 2: Performance evaluation of code generated by PeachPy and three C++ compilers from equivalent intrinsics.

V. Conclusion

In this paper, we introduced a Python framework for developing assembly compute kernels called PeachPy. PeachPy simplifies writing HPC kernels compared to traditional assembly, and introduces a significant degree of automation to the process. PeachPy allows developers to leverage the flexibility of Python to generate a large number of similar kernels from a single source file.

Acknowledgements

We thank Aron Ahmadia for his useful and insightful comments on this research and careful proofreading of the final draft of this paper. We thank Richard Vuduc for detailed suggestions of improvements for this paper.

[3]Vector logarithm and vector exponential functions compute `log` and `exp` on each elements of an input vector and produce a vector of outputs

This work was supported in part by grants to Prof. Richard Vuduc's research lab, The HPC Garage (www.hpcgarage.org), from the National Science Foundation (NSF) under NSF CAREER award number 0953100; and a grant from the Defense Advanced Research Projects Agency (DARPA) Computer Science Study Group program.

References

[1] F. G. Van Zee, T. Smith, F. D. Igual, M. Smelyanskiy, X. Zhang, M. Kistler, V. Austel, J. Gunnels, T. M. Low, B. Marker, L. Killough, and R. A. van de Geijn, "Implementing level-3 BLAS with BLIS: Early experience," The University of Texas at Austin, Department of Computer Science, FLAME Working Note #69. Technical Report TR-13-03, Apr. 2013.

[2] K. Yotov, T. Roeder, K. Pingali, J. Gunnels, and F. Gustavson, "An experimental comparison of cache-oblivious and cache-conscious programs," in Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures, 2007, pp. 93–104.

[3] A. Rigo and S. Pedroni, "PyPy's approach to virtual machine construction," in Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, 2006, pp. 944–953.

[4] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. Seljebotn, and K. Smith, "Cython: The best of both worlds," Computing in Science Engineering, vol. 13, no. 2, pp. 31–39, March-April 2011.

[5] llvmpy: Python bindings for LLVM. [Online]. Available: http://www.llvmpy.org

[6] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation," Parallel Computing, vol. 38, no. 3, pp. 157–174, 2012.

[7] A. Friedley, C. Mueller, and A. Lumsdaine, "High-performance code generation using CorePy," in Proc. of the 8th Python in Science Conference, Pasadena, CA USA, 2009, pp. 23–28.

[8] T. Malas, A. J. Ahmadia, J. Brown, J. A. Gunnels, and D. E. Keyes, "Optimizing the performance of streaming numerical kernels on the IBM Blue Gene/P PowerPC 450 processor," International Journal of High Performance Computing Applications, vol. 27, no. 2, pp. 193–209, 2013.

[9] J. Dongarra and P. Luszczek, "Anatomy of a globally recursive embedded LINPACK benchmark," in High Performance Extreme Computing (HPEC), 2012 IEEE Conference on, 2012, pp. 1–6.

[10] F. Boesch. (2008) pyasm, a python assembler. [Online]. Available: http://bitbucket.org/pyalot/pyasm

[11] G. Olson. (2006) PyASM users guide v. 0.3. /doc/usersGuide.txt. [Online]. Available: http://github.com/grant-olson/pyasm

[12] D. J. Bernstein and P. Schwabe, "NEON crypto," in Cryptographic Hardware and Embedded Systems–CHES 2012. Springer, 2012, pp. 320–339.

[13] 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 International Symposium on Computer Architecture, 2012, pp. 440–451.