Boosting SIMD Benefits through a Run-time and Energy Efficient DLP Detection

Michael Guilherme Jordan, Tiago Knorst, Julio Vicenzi and Mateus Beck Rutzig
Electronics and Computing Department - Federal University of Santa Maria – Santa Maria – Brazil
{michael.jordan, tiago.knorst, julio.vicenzi}@ecomp.ufsm.br; mateus@inf.ufsm.br

***Abstract* — Data Level Parallelism has been improving performance-energy tradeoff of current processors by coupling SIMD engines, such as Intel AVX and ARM NEON. Special libraries and compilers are used to support DLP execution on such engines. However, timing overhead on hand coding is inevitable since most software developers are not skilled to extract DLP using unfamiliar libraries. In addition, DLP detection through compiler, besides breaking software compatibility, is limited to static code analysis, which compromises performance gains. In this work, we propose a runtime DLP detection named as Dynamic SIMD Assembler, which transparently identifies vectorizable code regions to execute in the ARM NEON engine. Due to its dynamic fashion, DSA keeps software compatibility and avoids timing overhead on software developing process. Results have shown that DSA outperforms ARM NEON auto-vectorization compiler by 32% since it covers wider vectorized regions, such as Dynamic Range, Sentinel and Conditional Loops. In addition, DSA outperforms hand-vectorized code using ARM library by 26% reducing 45% of energy consumption with no penalties over software development time.**

Keywords— DLP, SIMD, Vectorization, ARM NEON

I. Introduction

Speech and vision recognition have been taking important role in the current era of cognitive computing to analyze human behaviors [1]. In particular, such application domains are benefit from Single Instruction Multiple Data (SIMD) machines since their algorithms are plentiful of Data-Parallel Statements.

Currently, SIMD engines are present in market processors, which can support the execution of such application domains. ARM NEON [2], Intel SSE/AVX [3] and IBM Altivec [4] are vector engines coupled to general purpose processors with the purpose of benefiting energy-performance tradeoff on data-parallel applications. The execution of such engines is supported by vector instructions that can be generated by: automatic vectorization through compiler and hand-coding using low-level functions available on specific programming libraries.

Both compiler techniques and libraries directives focus on exploiting Data Level Parallelism (DLP) opportunities on loops statements, since same operations are repeated over data independent structures. However, the vectorization of most loop statements, such as dynamic range, sentinel and conditional loops relies on runtime information, which restrict compiler and hand-coding DLP coverage. In addition, besides breaking software compatibility, timing overhead on hand-coding is inevitable since most software developers are not skilled to extract DLP using unfamiliar libraries.

In this work, we propose a runtime DLP detection, named as Dynamic SIMD Assembler (DSA), which transparently identifies vectorizable code regions, builds SIMD instructions and triggers the SIMD engine. Due to its dynamic fashion, DSA keeps binary compatibility and avoids timing overhead on software developing process. In addition, unlike compiler and hand-coding techniques, DSA is capable of vectorizing all aforementioned loop statements since it is aware of runtime information, which boost the DLP coverage and, consequently, performance-energy tradeoff over techniques based on static analysis. Summarizing, this work contributes to:

- show that is mandatory a runtime vectorization exploitation to boost the applications DLP coverage;
- propose an energy efficient runtime vectorization technique capable of boost applications performance by increasing vectorization coverage of techniques based on static analysis with no penalties over software development time.

This work is organized as follows; Section II presents the Related Work. Section III presents the Dynamic SIMD Assembler System. Methodology and Results are shown in Section IV. Finally, we present conclusions and future work in Section V.

II. Related Work

In the academic field, many researches have been exploiting Data Level Parallelism (DLP) to achieve performance improvements and energy savings. Sui Yulei [5] extends the LLVM compiler [6] to automatically vectorize Loop-Oriented Pointer. This technique is able to increase the number of vectorizable basic blocks achieving performance improvements from 2.95% to 7.23%.

Similar to [5], Zhou Hao [7] presents the Loop-Mix compiler that vectorizes loops focusing on reorganizing data when mixed SIMD parallelism (inter-loops and intra-loops) is considered. This technique outperforms the Loop-ILV by 36%. Sara S. Baghsorkhi [8] proposes FlexVec, a partial vectorization technique to dynamically adjust vector length for loops affected by cross-iteration dependencies. FlexVec extends the AVX-512 ISA showing a Geomean performance improvement over the original AVX ISA from 9% to 11%. Dorit Nuzman [9] proposes a compiler based techquite aiming to vectorize outer loop. It shows performance improvements of 3.13 and 2.77 when coupled to a Cell BE SPU and a PowerPC970, respectively.

Tian Xinmin [10] presents a set of C/C++ directives extensions for SIMD programming capable of automatic translating both functions and loops. Significant speedups (from 3.07x to 4.69x) are achieved when these optimizations are applied. Bramas Berenger [11] proposes Inastemp, a lightweight open-source C++ library that provides SIMD Vectorization to several ISA, such as SSE, AVX, AVX512 and ALTIVEC/VMX. The authors claim that Inastemp shows the same performance on exploiting DLP than libraries developed for specific ISA.

ARM NEON [2] is a SIMD engine coupled to the ARMv7 architecture that is triggered through specific ARM SIMD instructions. ARM supports two approaches to produce ARM NEON code: compiler auto-vectorization and ARM NEON software library.

Despite the advantages of automatic auto-vectorization through the compiler shown in [2][5][7][8][9], their static code exploitation limits the performance gains since it is not capable of identifying

vectorizable regions that depends on data that is only available at runtime, such as: conditional, dynamic range loops and sentinel loops. In addition, hand coding using software libraries proposed in [2][10][11] inevitably causes timing overhead since most software developers are not skilled to extract DLP using unfamiliar low-level functions. Thus, in this work, we propose Dynamic SIMD Assembler (DSA) that automatically vectorizes code regions to execute in a SIMD engine. DSA is a hardware module coupled to an ARM Processor responsible for detecting vectorizable code regions, generating ARM SIMD instruction and triggering NEON engine at runtime. Due to its dynamic nature, DSA keeps binary compatibility and avoids timing overhead on software developing process. Moreover, performance improvements with energy savings are feasible to achieve in a wide range of application domains since the proposed approach covers larger vectorizable code regions than static analysis techniques, such as count loops, conditional statements, dynamic range and sentinel loops.

III. Dynamic SIMD Assembler

A. System Overview

The Dynamic SIMD Assembler (DSA) is tightly coupled to the ARMv7-A processor [12]. Figure 1 shows the system overview. As it can be seen, the DSA is composed of a SIMD instruction logic detection and two cache memories (*DSA Cache and Verification Cache*). The DSA Cache is responsible for storing information about the built SIMD instructions over the vectorizable loops. The Verification Cache (V-Cache) stores the addresses of data memory accesses performed into the vectorizable loops (more details about caches in Section IV).

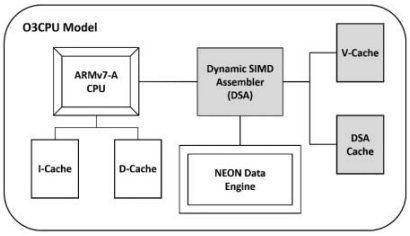

Figure 1. System Overview

Figure 2 shows an overview of how the DSA works. In the first scenario (*Scenario 1 – DSA Loop Analysis*), the DSA and ARMv7-A processor operate in parallel. While the ARM processor executes the incoming instructions, the DSA is in a probing mode, searching for a vectorizable loop to build SIMD instructions. In such execution mode, the NEON Engine remains deactivated. If the DSA detects a vectorizable loop, the second scenario is triggered (*Scenario 2 – DSA Loop Execution*). In this scenario, the DSA deactivates the ARMv7-A processor and activates the NEON Data Engine to execute the built SIMD instruction (Vectorized Instructions). It is important to notice that the DSA works in parallel with the ARMv7-A CPU execution, which means that the processor's critical path is not affected by the DSA.

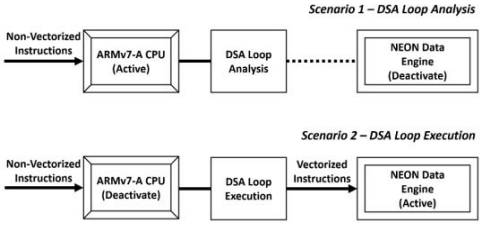

Figure 2. Execution Flow

B. Dynamic SIMD Assembler DLP Coverage

As explained before, the Dynamic SIMD Assembler is capable of exploiting vectorizable regions at runtime, which extends DLP Exploitation of hand coding using ARM library and auto-vectorization compiler. Figure 3 shows loop examples of (A) count loop; (B) dynamic range loop; (C) conditional loop; (D) loop with a function call. As it can be seen, the pseudocode (A) presents a simple vectorizable loop in which both the compiler and DSA would be capable of vectorize. The pseudocode (B) has a dynamic range loop, where the loop size is determined by an input or even a data calculated at runtime. The pseudocode (C) has a loop containing conditional statements, which the execution is also determined at runtime. The same evaluation can be made for the pseudocode (D), which has a loop containing a function call that depends on a variable calculated at runtime. In this way, the pseudocodes (B), (C) and (D) cannot be vectorized by the compiler auto-vectorization techniques since they depend on data manipulation at runtime. However, as the DSA (Dynamic SIMD Assembler) analyzes the application code at runtime, it is capable of vectorizing all aforementioned situation.

Summarizing, the DSA covers full vectorization of: Count Loops, Function Loops, Outer and Inner Loops, Dynamic Range Loops and Sentinel Loops. In addition, as it can be seen in next section, the DSA also supports partial vectorization of loops with cross-iteration dependencies.

```
1: procedure vectorizable_Loop()
2:     for i in range 0 to 400 do
3:         v[i] = a[i] + b[i];
4:     end for;
5: end procedure
```
A

```
1: procedure dyn_range_Loop()
2:     for i in range 0 to input/x do
3:         v[i] = a[i] + b[i];
4:     end for;
5: end procedure
```
B

```
1: procedure conditional_Loop()
2:     for i in range 0 to 400 do
3:         if input/x == 1 do
4:             v[i] = a[i] + b[i];
5:         else
6:             v[i] = a[i] - b[i];
7:         end if;
8:     end for;
9:end procedure
```
C

```
1: procedure function_Loop()
2:     for i in range 0 to 400 do
3:         v[i] = function_call(input/x);
4:     end for;
5: end procedure
```
D

Figure 3. Examples of Loops

C. Dynamic SIMD Assembler Overview

The Dynamic SIMD Assembler (DSA) detection process is based on a State Machine (SM) composed of six stages: Loop Detection, Data Collection, Dependency Analysis, Store ID/Execution, Mapping and Speculative Execution. Each one of these stages is activated in different loop iterations.

As it can be seen in Figure 4, the Loop Detection stage is triggered by the end of the first loop iteration. The Loop Detection stage is responsible for:

- detecting the presence of a loop;
- checking the existence of innermost-loop and outer-loops;
- accessing the DSA cache, checking if the current loop is already vectorizable.

The Data Collection stage is triggered in the second loop iteration. This stage is responsible for:

- evaluating the loop range (number of iterations), vectorizable instructions and their operands;
- identifying the existence of function calls and conditional code inside the loop;
- storing the addresses of data memory accesses in the Verification Cache.

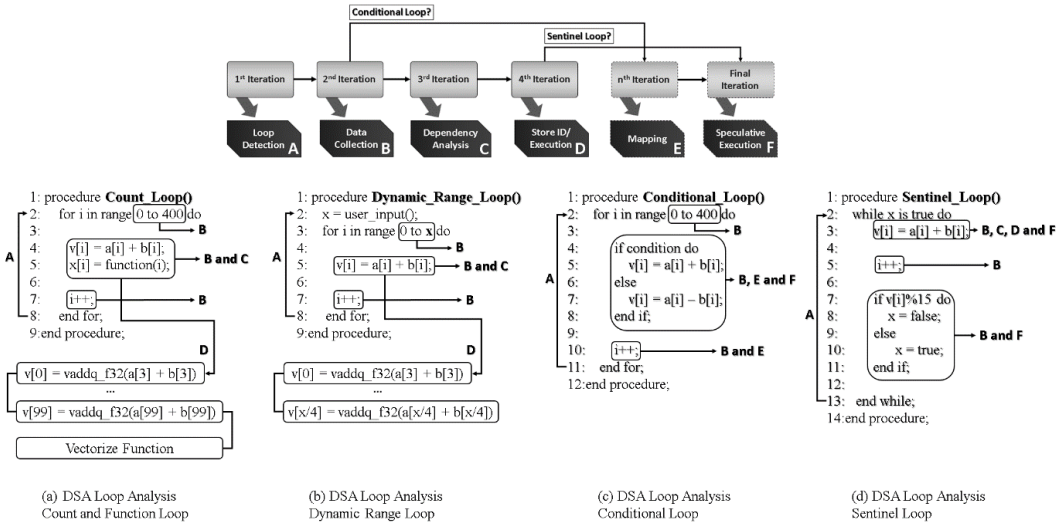

Figure 4. DSA Analysis and Execution Process

The Dependency Analysis stage is triggered in the third loop iteration. This stage is responsible for:

- analyzing the cross-iteration dependency (dependencies between two or more iterations in the same loop statement).

The Store ID/Execution stage is triggered in the fourth loop iteration. This stage is responsible for:

- concluding the vectorization of Count loops, Function loops, Outer/Inner loops, Dynamic Ranged Loops, Sentinel loops and Partial loops;
- generating and saving the loop identification (ID) in the DSA Cache;
- building SIMD instruction and activating the execution of the ARM NEON engine.

The Mapping stage is only activated for Conditional loops. This stage is responsible for:

- mapping the executed conditional code statements;
- detecting cross-iteration dependencies between conditional statements.

The Speculative Execution stage is only activated for Conditional and Sentinel loops. This stage is responsible for:

- selecting data generated during the vectorization in the end of loop execution (Sentinel and Conditional Loop);
- tracking Sentinel loop range;
- storing mapped conditions of Conditional Loop for further executions.

Figure 4 shows a DSA execution example by considering: Count and Function Loop (a), Dynamic Range Loop (b), Conditional Loop (c) and Sentinel Loop (d).

Following the *Count_Loop() (a)* procedure example, the Loop Detection stage (A) detects the loop by the end of the execution of the first iteration by analyzing instruction address gaps and branches. In the second iteration, the Data Collection stage (B) identifies the loop range (*400*) and the value of the increment/decrement (*i++*). In addition, such stage: stores the addresses of the data memory accesses (*Mem[a[i]], Mem [b[i]] and Mem[v[i]]*) in the Verification Cache; and identifies function calls inside the loop (*x[i] = function[i]*) by verifying branches and the memory address gap between instructions fetched from memory. The detection of function calls is mandatory to analyze cross-iteration dependencies since the increment/decrement register can be modified for an operation inside the function call. In the third iteration, the Dependency Analysis Stage (B) analyses data dependencies between iterations (more detailed in subsection D). For the current example, the DSA identifies that no cross-iteration dependency exists and triggers the Store ID/Execution Stage. Such stage stores the Loop ID in the DSA cache to avoid repeating loop analysis and builds SIMD instructions to execute the remaining iterations in the ARM NEON engine. The DSA needs four parameters to generate SIMD instructions: data type, loop range, operation and ARM NEON execution support. For such an example the parameters are: *float*, *400*, *add*, *128-bit wide*, respectively. Thus, the DSA generates an instruction equivalent to the *vaddq_f32* instruction of the NEON architecture. Since the corresponding ARM NEON engine can operate 128 bits in parallel and the float type is 32-bit wide data, the DSA divides the loop range by the factor four, running the *vaddq_f32* one hundred times, instead of executing a non-vectorizable *add* operation four hundred times.

In the *Dynamic_Range_Loop (b)* procedure example, the loop size is calculated at runtime but before the loop execution. In this case, the loop analysis passes through the same steps as the *Count_Loop (a)* example. However, instead of having a single analysis when the loop executes for the first time, the *Dynamic_Range_Loop (b)* must be analyzed on every execution, since the loop range can change on each loop execution, the Dependency Analysis Stage (C) needs to verify if the vectorization is allowed based on current value of the loop range.

Considering the *Conditional_Loop (c)* example, the Loop Detection Stage (A) detects the loop by the end of the execution of the first iteration. The Data Collection Stage (B), besides collecting the necessary data to vectorize the loop, also detects if all instruction addresses within the loop range were accessed. In case a instruction address gap is detected, a conditional loop is confirmed. In such case, the Mapping stage (E) is activated. This stage is responsible for mapping every condition within the loop body and detecting any cross-iteration dependency. If no cross-iteration dependency is detected, all conditions within the loop can be vectorized considering the remaining loop range. During the remaining loop execution, the DSA maps every accessed condition. While the mapping is activated, the vectorized instructions (*if: v[i] = a[i] + b[i] and else: v[i]=a[i]-b[i]*) are not executed. At the end of the loop, the Speculative

Execution Stage (F) selects the appropriate results based on the mapping process.

The *Sentinel_Loop (d)* vectorization is based on Speculative Execution. The DSA assumes a speculative loop range when detecting such loop type since it is not feasible to have such information beforehand. In the Data Collection Stage, besides collecting all the necessary data to vectorize the loop, the DSA chooses a loop range that maximizes utilization of the functional units available in the ARM NEON engine. Assuming a 128-bit wide ARM NEON, the DSA chooses a speculative loop range of four, in order to use all vector units, since the operands width is 32 bits (*32 bit float*). In the third iteration, in the Dependency Analysis Stage (C), the DSA analyses and predicts any cross-iteration dependency based on the speculative loop range. If no Cross-Iteration Dependency is found, the Store ID/Execution stage (D) is activated and the instructions are vectorized based on the speculative loop range. In addition, the loop ID is saved in the DSA cache. The speculative execution can provide three situations:

- if the loop executes fewer iterations than the speculated number of iterations, the execution results of the speculated number of iterations are written back, the remaining results are discarted and the loop range is updated in DSA cache;
- if the loop executes greater iterations than the speculated number of iterations, the execution results of the speculated number of iterations are written back, the further iterations are executed by the general purpose processor and the loop range is updated in DSA cache;
- if the loop executes the speculated number of iterations, the ARM NEON results of the speculated number of iterations are written back and the speculative range is maintained in the DSA cache.

D. Cross-iteration Dependency Prediction

At the memory access point of view, a cross-iteration dependency exists when the same data memory address is accessed in different loop iterations. The DSA cross-iteration analysis starts in the 2nd loop iteration, where the addresses of data memory accesses are saved in the Verification Cache (VC). Even having the memory addresses in the VC and comparing them to the memory addresses performed on every iteration, one cannot discard cross-iteration dependencies in future iterations. Assuming such situation, we have implemented *Cross-iteration Dependency Prediction*.

The equations below describe the steps of the prediction process, where $M_{Read[2]}$ and $M_{Read[3]}$ is the memory address accessed by a $MemRead$ ($load$) instruction in the second and third loop iterations, respectively. $M_{Read[lastIteration]}$ is the memory address accessed by a load instruction in the last executed iteration (Equation 4), x is the interval between $M_{Read[2]}$ and $M_{Read[lastIteration]}$ (Equation 1), $M_{Write[2]}$ is the memory address accessed by a *MemWrite (store)* instruction in the second iteration (Equations 2 and 3), M_{Range} is the memory address range between the $M_{Read[2]}$ and $M_{Read[3]}$ (Equation 5), CID means Cross-Iteration Dependency and *NCID* means No Cross-Iteration Dependency.

$$M_{Read[3]} \leq x \leq M_{Read[lastIteration]} \quad (1)$$

$$M_{Write[2]} \in x \rightarrow CID \quad (2)$$

$$M_{Write[2]} \notin x \rightarrow NCID \quad (3)$$

$$M_{Read[lastIteration]} = M_{Read[2]} + (M_{Gap} * (lastIteration - 2)) \quad (4)$$

$$M_{Gap} = |M_{Read[3]} - M_{Read[2]}| \quad (5)$$

Considering the equations above, if the $M_{Write[2]}$ is within the memory address range of $M_{Read[3]}$ and $M_{Read[lastIteration]}$ (Equation 2), the loop would have a cross-iteration dependency since the load instruction of a future loop iteration could perform a memory access in the same memory address of the store instruction executed in the second loop iteration. The memory address of the load instruction executed in the last iteration is predicted based on the sum of the $M_{Read[2]}$ and the equation $(M_{Gap} * (lastIteration - 2))$ (Equation 4). Thus, in case of $M_{Write[2]}$ is out of the memory address interval of $M_{Read[3]}$ and $M_{Read[lastIteration]}$ (Equation 3), one can ensure that the loop has no cross-iteration dependency.

Figure 5 illustrates an example of how *Cross-iteration Dependency Prediction (CIDP)* works. In such example, the DSA detects that there is no cross-iteration dependency between 2nd and 3rd iteration. Thus, by the end of the 3rd loop iteration, the *CIDP* is activated by applying Equation 5 ($M_{Gap} = |0x104 - 0x100| = 0x004$). Using Equation 4, one can calculate the memory address of the load instruction of the last iteration $M_{Read[lastIteration]} = 0x100 + 0x020 = 0x120$. By applying Equations 1 and 2, the *CIDP* detects that $M_{Write[2]} = 0x108$ is within the interval $(M_{Read[3]} \leq x \leq M_{Read[lastIteration])} = 0x100 \leq x \leq 0x120)$, which produces a cross-iteration dependency.

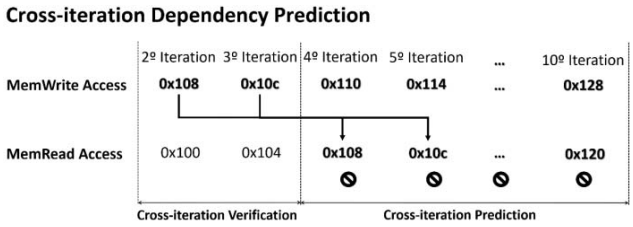

Figure 5. Example of a Cross-iteration Dependency Prediction Process

E. Dynamic SIMD Assembler Partial Vectorization

Despite having cross-iteration dependencies, loops can be partially vectorized by avoiding vectorization of iterations that produces dependencies. Figure 6 shows how the partial vectorization works. As it can be seen, the CIDP detects cross iteration dependency between the 2nd Iteration and the 11th Iteration due to the data memory address 0x124. However, there is a gap between the 2nd Iteration to the 10th that could be vectorized. Thus, for such an example, the DSA performs the vectorization detection process from 1st to 4th Iterations, which allows the vectorization from the 4th up to 10th iteration which provides data to the vectorization of the iterations from 11th up to 19th. The same process repeats until the end of the loop execution.

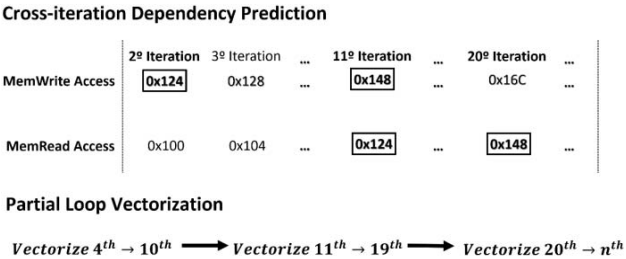

Figure 6. DSA Partial Vectorization Technique

IV. RESULTS

A. Methodology

We have coupled the DSA to an ARMv7 ISA processor using the O3CPU model of gem5 [12] simulator to evaluate the proposed approach. To gather performance results, we have compared the DSA with:

- an ARMv7 ISA processor without NEON engine (ARM Original Execution);
- an ARMv7 ISA processor coupled to a NEON architecture exploiting DLP through the support of the ARM NEON auto-vectorization compiler (ARM NEON AutoVec);
- an ARMv7 ISA processor coupled to a NEON architecture exploiting DLP through hand-coded applications using ARM NEON library (ARM NEON Hand-Coded).

Table 1 shows the configurations of all setups. It is important to notice that we coupled the same ARM NEON architecture in ARM NEON AutoVec, ARM NEON Hand-Coded and DSA, which provides the same DLP exploitation degree. We used Cadence RTL Compiler [13] to gather energy results from the VHDL description of the Dynamic SIMD Assembler and McPAT of the ARMv7 processor.

Table 1. System Setups

	ARM Original Execution	ARM NEON DSA	ARM NEON AutoVec	ARM NEON Hand-Coded
Processor	gem5 O3CPU (ARMv7)	gem5 O3CPU (ARMv7)	gem5 O3CPU (ARMv7)	gem5 O3CPU (ARMv7)
Superscalar Width	2-wide	2-wide	2-wide	2-wide
CPU Clock	1GHz	1GHz	1GHz	1GHz
L1 Cache	64 kb	64 kb	64 kb	64 kb
L2 Cache	512 kb	512 kb	512 kb	512 kb
Cache Policy	LRU	LRU	LRU	LRU
Parallelism (NEON)	Not Used	Type Dependent 128-bit Wide	Type Dependent 128-bit Wide	Type Dependent 128-bit Wide
DSA Cache	-	8 kb	-	-
Verification Cache	-	1 kb	-	-

B. Benchmarks Characterization

Aiming to create a heterogeneous workload to evaluate the proposed approach, we have selected benchmarks from different suites following their opportunities to exploit DLP: MM 64x64 [14] and RGB-Grayscale [14], which provide great opportunities; Susan E [14] and JPEG [16] that provide medium opportunities; and Bit Counts [14], Susan C [14], Susan S [14] and Gaussian Filter [15], which have low opportunities. Figure 7 presents a static profiling that considers the percentage of each loop type in the aforementioned benchmarks. Such analysis quantifies the presence of vectorizable loops statically, which means that weight over the execution time of each loop is not considered.

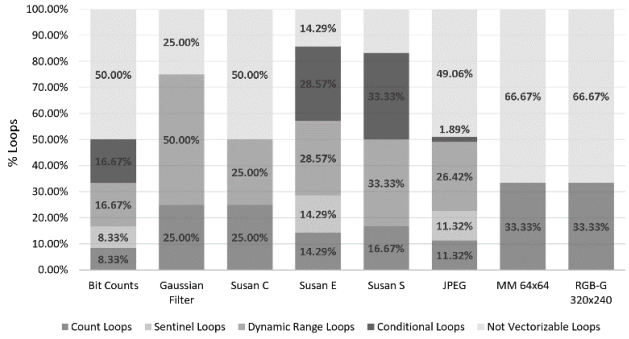

Figure 7. Percentage of Loop Types in the Selected Applications

As it can be seen, there are different degrees of vectorization opportunities in the selected benchmarks, 86% of the Susan E loops can be vectorized but just 33% of MM 64x64 and RGB-G 320x240. However, as explained before, considering 86% of Susan E vectorizable loops, only 14.3% (Count Loops) could be vectorized at compile and programming time. On the other hand, all 33% of the loops of MM 64x64 and RGB-G 320x240 are vectorized at compile and programming time. However, on average, only 21% of the application loops can be vectorized at compile and programming time. A runtime analysis potentially increases such coverage to 57%, since it can vectorize all considered loops types. The benchmarks characterization indicates the need for a runtime analysis to boost application performance on exploiting DLP.

As shown in Section III, the DSA produces a time overhead to detect vectorizable code regions and build NEON instructions. Table 2 shows the percentage of the execution time spent in the DSA detection process. As it can be seen, Bit Counts and Susan E spend 26.20% and 15.58% of the execution time detecting vectorizable loops. Such applications contain sentinel loops that relies on Mapping Stage execution (Figure 4) which leaves active during the whole vectorization/execution process. The remaining benchmarks spend, on average, only 1.53% of the whole execution time detecting vectorizable regions showing the acceptable overhead of the proposed approach.

Table 2. DSA Detection Latency

	Bit Counts	Gaussian Filter	Susan C	Susan E	Susan S	JPEG	MM 64x64	RGB-G 320x240
DSA Latency	26.20%	0.53%	1.91%	15.58%	3.71%	2.69%	1.74%	1.68%

C. Performance

Figure 8 shows the performance improvements of the ARM NEON DSA, the ARM NEON AutoVec and the ARM NEON Hand-Coded over the ARM Original Execution. As it can be noticed, the proposed technique provides performance improvements over the ARM Original Execution in all benchmarks. The performance gains increase as the DLP opportunities increase as well. Bit Counts (low DLP opportunities) shows performance improvements of 32% while RGB-G 320x240 (great DLP opportunities) of 70%. RGB-G and MM 64x64, besides having only 33.33% of vectorizable loops (shown in Figure 7), such loops consume most of the execution time, which explains the high acceleration in both benchmarks. Results demonstrate the efficient runtime DLP exploitation of the proposed approach by showing, on average, 45% of performance improvements over ARM Original Execution running applications with heterogeneous DLP opportunities.

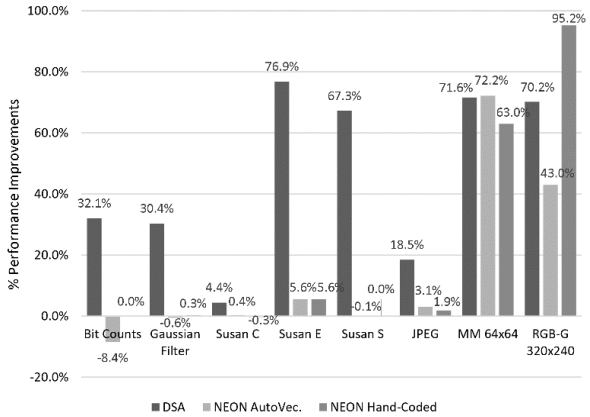

Figure 8. Performance Improvements over ARM Original Execution

Due to the larger DLP exploitation opportunities of DSA over the static analysis of the ARM NEON AutoVec, the proposed approach outperforms the compiler technique in all benchmarks but MM 64x64 by only 0.6%. Performance gains of the proposed approach over compiler technique comes from the vectorization of Sentinel Loops, Dynamic Ranged Loops and Conditional Loops vectorization which are not capable to be achieved at compile time. As it can be seen in Figure 7, considering Susan E, ARM NEON AutoVec covers only 14.3% of vectorizable loops (Count Loops) while DSA boost such covering to 86% which results on 71,5% of performance improvements over the compiler technique. In addition, the ARM NEON AutoVec provides performance penalties in Bit Counts, Gaussian Filter and Susan S since the greater latencies of NEON instructions allocated by the compiler were not diluted by DLP performance gains. Besides keeping the binary compatibility, broken by the ARM NEON AutoVec, the proposed approach provides, on average, 32% of performance improvements over the ARM NEON AutoVec technique considering benchmarks with different DLP opportunities.

Similar to the ARM NEON AutoVec, due to its dynamic DLP exploitation, the proposed approach outperforms ARM NEON Hand-Coded.

D. Energy

Figure 9 shows the energy consumption of DSA, ARM NEON AutoVec and ARM NEON Hand-Coded considering the ARM Original Execution as a baseline. As it can be seen, the DSA achieves greater energy savings than static analysis approaches in all benchmarks but RGB-G 320x240 since such an application boost performance due to code optimizations using ARM NEON library. On average, DSA achieves 45%, 31.2% and 23.5% of energy savings over ARM Original Execution, ARM NEON AutoVec and ARM NEON Hand-Coded, respectively.

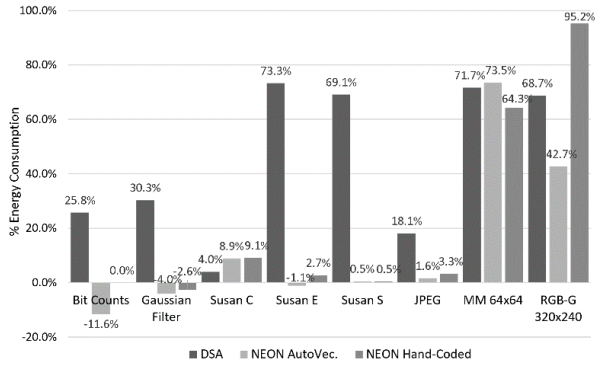

Figure 9. Energy Savings over ARM Original Execution

Table 3 shows the energy consumption percentage of the DSA hardware relative to the whole system energy (ARMv7 CPU + NEON Engine). As it can be noticed, the DSA detection process is responsible for, at most, 11% and, on average, for 2.8% of the whole system energy. Summarizing, the experiments have shown that the lightweight DSA detection achieves higher performance than static analysis approaches with lower energy consumption maintaining binary compatibility with no penalties on software development time.

Table 3. DSA Energy Consumption

	Bit Counts	Gaussian Filter	Susan C	Susan E	Susan S	JPEG	MM 64x64	RGB-G 320x240
ARM+NEON Consumption	88.52%	99.77%	99.52%	93.47%	97.54%	99.11%	99.40%	99.90%
DSA Consumption	11.48%	0.23%	0.48%	6.53%	2.46%	0.89%	0.60%	0.10%

V. Conclusion and Future Work

In this work, we propose the Dynamic SIMD Assembler (DSA) that automatically vectorizes code regions to execute in ARM NEON. In comparison with compiler and programming techniques, due to its dynamic nature, DSA boosts DLP coverage, keeps binary compatibility and avoids timing overhead on software developing process. Experimental results show that the DSA outperforms both ARM NEON Auto-Vectorization and ARM NEON Hand-Coded methods by 32% and 26%, respectively, while keeping binary compatibility and software productivity. In terms of energy, the DSA shows 45%, 31% and 23.5% of energy savings over the ARM Original Execution, ARM NEON AutoVec and ARM NEON Hand-Coded considering applications with heterogeneous DLP opportunities. For future works, we intend to merge DLP, ILP and TLP exploitation in a single MPSoC by using DSA in a heterogenous fashion.

ACKNOWLEDGEMENT

We are grateful to the institutions listed below for the financial support that they are providing us: Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) and Fundação de Amparo à pesquisa do Estado do Rio Grande do Sul (FAPERGS).

References

[1] Dharmendra S. Modha, Rajagopal Ananthanarayanan, Steven K. Esser, Anthony Ndirango, Anthony J. Sherbondy, and Raghavendra Singh. 2011. Cognitive computing. Commun. ACM 54, 8 (August 2011), 62-71.

[2] Reddy, Venu Gopal. "Neon technology introduction." *ARM Corporation* (2008).

[3] Lomont, Chris. "Introduction to Intel advanced vector extensions." *Intel White Paper* (2011): 1-21.

[4] Diefendorff, Keith, et al. "Altivec extension to PowerPC accelerates media processing." *IEEE Micro* 20.2 (2000): 85-95.

[5] Sui, Yulei, et al. "Loop-oriented array-and field-sensitive pointer analysis for automatic SIMD vectorization." *ACM SIGPLAN Notices*. Vol. 51. No. 5. ACM, 2016.

[6] Lattner, Chris, and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation." *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2004.

[7] Zhou, Hao, and Jingling Xue. "Exploiting mixed SIMD parallelism by reducing data reorganization overhead." *Proceedings of the 2016 International Symposium on Code Generation and Optimization*. ACM, 2016.

[8] Baghsorkhi, Sara S., Nalini Vasudevan, and Youfeng Wu. "FlexVec: auto-vectorization for irregular loops." *ACM SIGPLAN Notices*. Vol. 51. No. 6. ACM, 2016.

[9] Nuzman, Dorit, Ira Rosen, and Ayal Zaks. "Auto-vectorization of interleaved data for SIMD." *ACM SIGPLAN Notices* 41.6 (2006): 132-143.

[10] Tian, Xinmin, et al. "Compiling C/C++ SIMD extensions for function and loop vectorizaion on multicore-SIMD processors." *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*. IEEE, 2012.

[11] Bramas, Berenger. "Inastemp: A Novel Intrinsics-as-Template Library for Portable SIMD-Vectorization." *Scientific Programming* 2017 (2017).

[12] Binkert, Nathan, et al. "The gem5 simulator." *ACM SIGARCH Computer Architecture News* 39.2 (2011): 1-7.

[13] Cadence, R. T. L. "Compiler User's Manual."

[14] Guthaus, Matthew R., et al. "MiBench: A free, commercially representative embedded benchmark suite." *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*. IEEE, 2001.

[15] Bradski, Gary, and Adrian Kaehler. "OpenCV." *Dr. Dobb's journal of software tools* 3 (2000).

[16] Lee, Chunho, Miodrag Potkonjak, and William H. Mangione-Smith. "MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems." *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, 1997.