ROMANIAN JOURNAL OF INFORMATION SCIENCE AND TECHNOLOGY
Volume **16**, Number 4, 2013, 336–350

OPINCAA: A Light-Weight and Flexible Programming Environment For Parallel SIMD Accelerators

Călin BÎRĂ[1], Lucian PETRICĂ[1,2], Radu HOBINCU[1]

[1]*Politehnica* University of Bucharest
[2]IMT Bucharest

E-mail: `{calin.bira,lucian.petrica,radu.hobincu}@arh.pub.ro`

Abstract. The programming environment is often a key enabler of productivity and performance in any software development project. In a hardware-software co-design scenario, the programming environment must easily adapt to changes in the parameters of the underlying hardware computation platform, and must be able to accurately measure the performance of the hardware in order to guide software development. In this paper we present OPINCAA, a framework for programming parallel accelerators which implement the ConnexArray architecture. OPINCAA provides a C++ syntax for accelerator software development, and the infrastructure for dispatching instructions to the accelerator. OPINCAA is also designed to easily interface with architectural and circuit-level simulators, and provides tools for performance analysis and automatic tuning of accelerator code. We evaluate OPINCAA in conjunction with a FPGA implementation of the ConnexArray, and demonstrate how it can be used to develop, debug and optimize vector applications.

1. Introduction

Single Instruction Multiple Data (SIMD) computation has traditionally been implemented either through multimedia extensions for general-purpose instructions sets, as is the case of Intel SSE/AVX [3], or using many-core accelerators connected through system buses to the CPU and main memory. The multimedia extension approach has the advantage of tight integration with the general-purpose processor, as SIMD instructions can be interleaved with scalar instructions, and a single compiler is required to program both the CPU and the SIMD engine. However, the width of SIMD lanes is limited, and therefore so is the amount of parallelism which can be extracted from the application using multimedia extensions.

Conversely, the host-accelerator paradigm is an increasingly popular and widely used model for heterogeneous computing using many-core accelerators. In this paradigm, control functions execute on a general-purpose CPU, the host, while intensive computational tasks are dispatched to a special-purpose computing device, the accelerator. This execution model has been used by General-Purpose GPUs [7, 4, 6], and more recently by Intel's Xeon Phi [8].

Because the accelerator is only loosely coupled to the CPU, manufacturers are free to use custom architectures and implement massive parallelism. The harnessing of such processing power brings up new challenges, such as the large latency of communication between the CPU and the accelerator. In this context, several programming environments for host-accelerator computation have emerged: Nvidia CUDA [6], ATI Stream [1], OpenCL [10] and others. These programming environments require the programmer to write separate code for the host and accelerator, which is handled by different compilers. While these technologies provide powerful facilities for programming, debugging and profiling accelerator code, they rely on certain accelerator features, such as the ability to store code locally on the accelerator, and to execute branches and control functions in the accelerator.

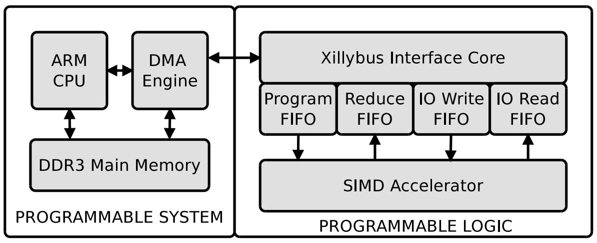


Fig. 1. Connex-ARM Architecture.

The Connex-ARM architecture, as presented in [2], is a compromise between the instruction set extension paradigm and the host-accelerator paradigm. Figure 1 presents the system architecture of the Connex-ARM, implemented on a Zynq SoC [9], whereby a ConnexArray resides in the FPGA fabric of the Zynq and is connected through the SoC bus to a dual-core ARM Cortex-A9 general-purpose processor. The ConnexArray SIMD lane is 256 bytes wide, while the standard NEON [5] SIMD unit on the ARM processors is 16 bytes wide. In this configuration, the ConnexArray has very little local program storage and control features, and is designed instead to function like an alternative, larger multimedia extension for the ARM processors.

Despite being relatively tightly coupled, with relatively low access latency, the ConnexArray is external to the ARM processor pipeline, which precludes the combining of ARM and ConnexArray instructions into the same binary, as is the case for NEON and all other multimedia extensions. Therefore, the Connex-ARM requires a new type of programming environment, which meets the following requirements:

- Host code and accelerator code can be mixed and interleaved, in a similar fashion to multimedia extensions. This requirement enables the programmer to use the host processor for control and the accelerator for data-intensive computation, without the need to include branching circuitry in the ConnexArray.
- Accelerator code can be separated from host code at run-time and dispatched through the system bus to the accelerator. For maximum flexibility, a just-in-time processing of accelerator code is desirable, whereby the accelerator program is built when first required, stored for further use, and rebuilt if any of its parameters change as a result of control code executed on the host processor.
- The programming environment must be light-weight, especially with regard to memory requirements, in order to function properly on embedded systems with small caches and main memory.

OPINCAA (OPcode INjector for the ConnexArray Architecture) is a programming environment which subscribes to all requirements listed, and enables the efficient use of the Connex-ARM system. The rest of this paper is organized as follows. Section 2 presents the OPINCAA instruction stream generator and dispatch mechanism, which represent the core functionality of OPINCAA and allow the ConnexArray to be used in conjunction with the ARM processors. Section 3 describes the set of architectural simulators included with OPINCAA for functional verification support. Section 4 discusses modules within the OPINCAA framework which help the programmer profile and optimize accelerator code, while Section 5 lists conclusions and avenues for future work.

2. OPINCAA Core

The most important task of OPINCAA is to allow the programmer to utilize the ConnexArray accelerator. For this purpose, the core functionality of OPINCAA is related to the concept of *kernel*, which is a section of code designed to run on the accelerator instead of the host. OPINCAA manages kernel creation, assembly and dispatch, as well as all data transfers required for the completion of kernel execution. The system architecture is presented in Fig. 2.

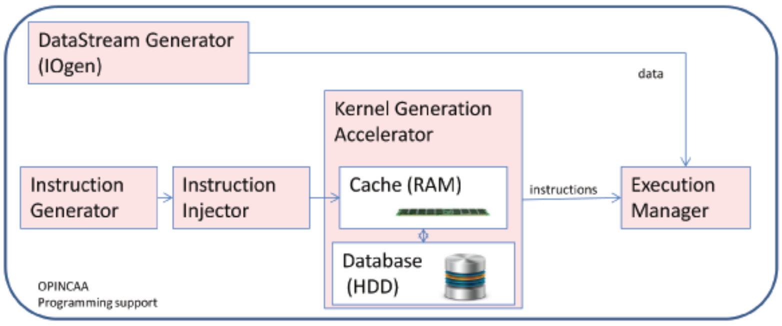


Fig. 2. OPINCAA Core Programming support.

A code generation module handles instruction generation and injection, as well as kernel caching. Data transfers are controlled, and transferred data is packetized by the IOgen module. Overall control of the execution and data transfer processes is handled by the Execution Manager.

2.1. Code Generation and Injection

The kernel generation process outputs machine code for the Connex-ARM. The generator consists of two functional units that work together: the instruction generator, and the instruction injector. The instruction generator is a collection of C++ classes that implement vector types and operators for vectors. The vector data type is implemented by the cnxVector class, that defines the characteristics of a Connex machine vector.

The instruction generator models the available vector memory within the Connex-ARM: the register file and the local storage. The instruction generator instantiates and utilizes 32 objects from the cnxVector class, denoted R0 to R31, which represent the hardware vector registers of the parallel Connex-ARM machine. In order to allow indexed accessed of the Connex-ARM vectors (e.g., from inside loops), a vector of cnxVector objects is exposed as R[]={ R0, . . . , R31}. Kernels operate exclusively on these 32 vector objects.

Additionally, the instruction generator instantiates 1024 vectors in the LS[] array, which model the Connex-ARM local storage. Operations cannot take place between elements of LS directly, and instead these must be copied from the LS vector to the R vector beforehand. Another important distinction between the LS array and the R array is that, whereas R can only be indexed by a scalar, LS can also be indexed by a vector, namely an element of R.

For ease of use, arithmetic and logical C++ operators were overloaded to function with Connex vector data types. The OPINCAA syntax is therefore similar to C++, and the “+”,”-”,”*”,”&”,”|”,”^”,”~”,”>>”,”<<” operators compute their respective functions element-by-element when applied to Connex-ARM vectors. In addition to overloaded operators, OPINCAA exposes several keywords and block types which serve to implement functionality not present in traditional C-style syntax. These are the following:

- EXECUTE_IN_ALL and EXECUTE_WHERE_[LT/EQ], which defines a block of instructions which execute conditionally on selected parts of a vector. The condition ALL executes the instructions irrespective of flag values, LT executes instructions only in PEs with the LT flag set, and EQ executes instructions in PEs with the EQ flag set.

- NOP introduces a one-cycle delay

- INDEX is a command to load the index of each PE into the register file

- REDUCE_ADD(Rx) is the reduction by addition of vector Rx, whereby all elements of Rx are added together

- REPEAT(N)/END_REPEAT defines a block of instructions which will be executed N times, such as in a for loop. The REPEAT loop is not unrolled at assembly-time, and the loop is executed in the Connex-ARM accelerator

- CELL_SH[L/R] is the vector rotation, to the left or right

Algorithm 1 gives an example of a kernel written in OPINCAA syntax.

Algorithm 1 Example Kernel

```
void GenerateKernelDemo(int DEMO_KNR) {
        BEGIN_KERNEL(DEMO_KNR);
        /* execute on all machines */
        EXECUTE_IN_ALL(
                NOP;
                LS[100] = R4;
                R3 = R1 * R2;
                LS[R1] = R7;
        )
        /* execute only on some machines */
        EXECUTE_WHERE_LT(
                R1 = INDEX;
        )
        /* execute on all machines */
        EXECUTE_IN_ALL(
                R3 = LS[R6];
                R1 = ~R3;
                R5 = R3 >> 5;
                R1 = R1 ^ R1;
        )
        END_KERNEL(DEMO_KNR);
}
```

Connex-ARM instructions consist of an op-code, a destination field and two source fields, which denote where the result will be stored, and where the operands are in the vector register file, respectively. Instructions are generated at run-time, when either an operation on a Connex-ARM vector is encountered, or an assignment to a vector is made. Each overloaded operator, when executed, creates an instruction object, in which the left and right operands are initialized to the indexes of the vector objects involved in the computation. The assignment operator "=" initializes the destination field and the op-code. The instruction is subsequently ready to be injected into the kernel code.

Algorithm 2 Complex Kernel

```
void SumAbsDiff(int x) {
        R31 = R30 - R[x];
        R31 = R31 * R31;
        REDUCE_ADD(R31);
}

void InitEuclidianDistanceKernel() {
        BEGIN_KERNEL(EuclidianDistanceKNR)
        EXECUTE_IN_ALL (
                for(int x=0; x<30; x++) {
                        R[x] = LS[x];
                }
                for(int y=0 ; y<NUM; y++) {
                        R30 = LS[y + 30];
                        for(int x=0; x<30; x++) {
                                SumAbsDiff(x);
                        }
                }
        )
        END_KERNEL(EuclidianDistanceKNR)
}
```

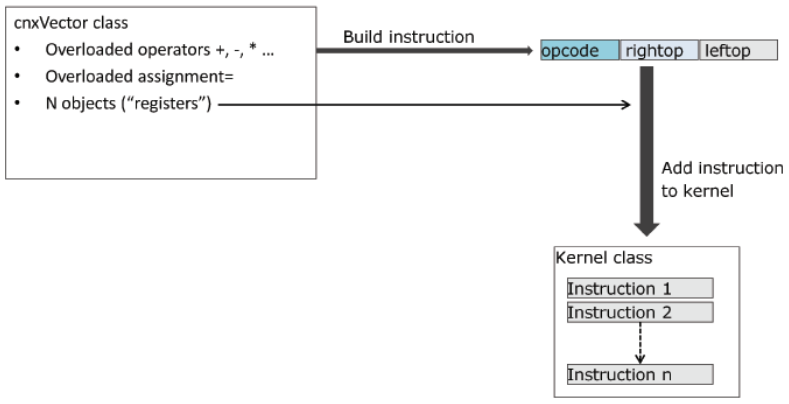


Fig. 3. Instruction Generation.

This method of run-time instruction generation behavior (*i.e.*, the host parses the code in the kernel at run-time, for the purpose of assembling it) through operator overloading allows for three important features:

- Kernel code can contain arbitrary scalar code which will execute on the host during kernel assembly,

- Loops over vector instructions can be unrolled at kernel assembly time through the use of simple host-side loops (for / while etc.),
- Complex (fused) instructions can be defined beyond the physical Connex-ARM instruction set

To exemplify the utilization of these features, let us assume we want to calculate the Euclidean distance of 30 vectors against NUM points. Assuming the 30 vectors are in LS[0] to LS[29] and the NUM vectors are in LS[30] to LS[29 + NUM], we can compute the distances using the kernel in Algorithm 2. The for loops execute on the host, and effectively result in unrolled loops in the final kernel executed on the accelerator. All other computation which does not involve vectors will also execute on the host, such as x and y index tests and incrementation, and computation of the LS indexes.

Common sets of instructions were implemented in OPINCAA as fused instructions. Fused instructions inject two or more Connex-ARM instructions into the kernel, with the immediate benefit of reduced apparent code size and greater programmer productivity. Examples of fused instructions are:

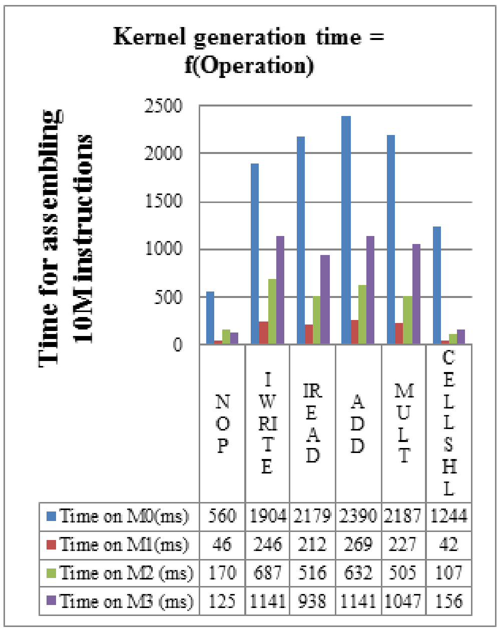
- Multiplication of all vector elements with a scalar constant,
- Addition of a scalar constant to all vector elements

Beyond the productivity gain, fused instructions give the opportunity to actually implement the combined instructions in hardware, without requiring a rewrite of of the code or OPINCAA operators, resulting in a performance increase.

The instruction injector is controlled through three syntactic structures, which are (i) the BEGIN_KERNEL macro, (ii) the END_KERNEL macro and (iii) the overloaded equality ("=") operator between two vector expressions. The BEGIN_ KERNEL and END_KERNEL are two C preprocessor macros that surround a block of code and signal to the injector that the expressions contained within the enclosed block are to be injected in a particular kernel, which can be assigned a name. The equality operator instructs the injector to add the corresponding instruction to the kernel, not before the instruction generator has finished assembling the machine instruction. The instruction injector does not know the size of the kernel in advance, which makes memory allocation an issue. The injector can either make use of dynamically-sized C++ arrays to construct the kernel, or utilize a two-pass process whereby the first pass estimates memory requirements and the second pass does the actual assembly and injection. Both solutions have been utilized and did not present significant performance differences. Figure 4 presents the speed of the assembly process for kernels containing a single type of instruction, on four different computers:

- Machine 0 (M0) : Xillinux (Ubuntu 32-bit) CPU: ARM A9 dual core @ 667MHz.
- Machine 1 (M1): Windows 7 64-bit, CPU: Intel i7-2670QM @ 2.8 GHz
- Machine 2 (M2): Windows 7 64-bit, CPU: AMD Turion 64 X2 @ 1.6 GHz
- Machine 3 (M3): Windows XP (32-bit), CPU: Intel Atom Z520 @ 1,33 GHz

As expected, the time for assembling NOP instructions is the shortest, because there are no operands. Instructions with 3 operands take longest to assemble and inject.



	NOP	IWRITE	IREAD	ADD	MULT	CELLSHL
🟦 Time on M0(ms)	560	1904	2179	2390	2187	1244
🟥 Time on M1(ms)	46	246	212	269	227	42
🟩 Time on M2 (ms)	170	687	516	632	505	107
🟪 Time on M3 (ms)	125	1141	938	1141	1047	156

Fig. 4. Assembly Speed.

2.2. Kernel Execution

The Execution Manager handles both the kernel execution and the data transfer. Kernel execution is initiated via the macro EXECUTE_KERNEL(int KernelNumber) which abstracts a function call that transfers the kernel instructions to the parallel machine. The actual transfer of instructions from the host to the accelerator utilizes the DMA engine and a specialized driver which exposes a set of special files, called “pipes”. Pipe files act as software First-In First-Out (FIFO) buffers. When writing or reading from pipes, DMA transfers are automatically set up by the operating system to move data between the pipe file and the accelerator hardware FIFOs. Therefore, the pipe files act as a virtual image of the hardware FIFOs of the Connex-ARM. The use of pipe files enables easy integration between OPINCAA, the Connex-ARM accelerator and the instruction set simulator.

The transfer of input/output data to and from the accelerator is implemented via instructions or read/write functions which utilize the IO system of the parallel machine. In the Connex-ARM architecture, data transfer is possible in the following ways:

- Host-to-accelerator via the VLOAD instruction
- Host-to-accelerator via ExecutionManager’s IOwrite capability

- Accelerator-to-host via the REDUCE instruction
- Accelerator-to-host ExecutionManager's IOread capability

When the same data is needed in all the cells of the Accelerator, the preferred method for data transfer is the VLOAD instruction, which replicates a single 16-bit value into a Connex-ARM vector. When different data is to be fed to the accelerator's cells, the IO capability will be used. The data transferred is assembled together by the Datastream Generator, as indicated by Fig. 2. Its task is to packetize the to-be-transfered data according to a specific IO protocol used by the parallel machines.

2.3. Kernel Storage

OPINCAA defaults to Ahead-of-Time (AOT) assembly, whereby the kernels are assembled as they are defined in the host, and before being required for execution. A Just-in-Time (JIT) mode is available, whereby the kernel is assembled when a call to execute the kernel occurs. Efforts were put into eliminating the assembling time during runtime by using a database of previously assembled kernels. This is the task of the Kernel Generation Accelerator in Fig. 2. OPINCAA can build kernels offline and store them in the database. At startup, OPINCAA can (on-demand) load some of the offline-generated kernels so that when a request is made for executing one of the kernels, it just runs them from cache.

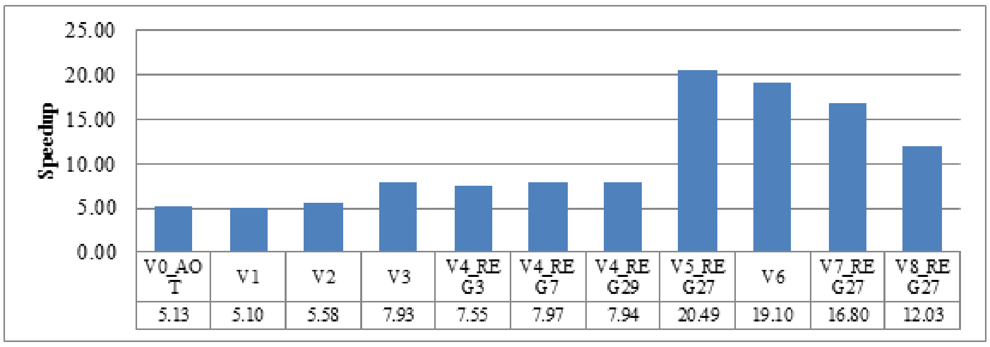
On powerful systems that can compute faster than they can access data in the database, as is the case with M1, there was no speedup, or even slow-down. However, on weaker like M0 and M3, speedup was observed to be as much as 6x when two kernels, each having 480K instructions, were retrieved from the database instead of being assembled at runtime. The solution for an even higher speedup is to have faster access to the database (*e.g.*, a RAM drive).

3. Instruction Set Simulator

In order to enable development and debugging of software in the absence of a Connex-ARM accelerator, OPINCAA is bundled with an instruction set simulator that is a separate executable from OPINCAA, and which creates the set of pipe files normally exposed by the Connex-ARM accelerator. Therefore, there is no modification required to OPINCAA in order to run on the simulator instead of real hardware.

Beyond the obvious necessity for instruction set compatibility with the Connex-ARM accelerator, the main requirement for the simulator is to provide good simulation speed. Figure 5 presents simulator performance for various kernels implementing the sum of absolute differences algorithm, when running on a computer utilizing a Core i7 2670QM processor. The performance of the kernels running on real Connex-ARM hardware in the Zynq FPGA serves as benchmark. The kernel variants utilize varying numbers of registers, transfer sizes, fused or simple instructions, and rolled or unrolled loops. The simulation times on the selected processor are at least five times smaller (hence, speedup over 5x) than execution time on real Connex-ARM hardware on the

Zynq device, which is to be expected given the large frequency differential (GHz for the i7 processor, 100MHz for the Connex-ARM) and benchmarks show an increase of speed by 3.5x through the use of SSE/AVX instructions to simulate Connex-ARM vector instructions.



V0_AOT	V1	V2	V3	V4_REG3	V4_REG7	V4_REG29	V5_REG27	V6	V7_REG27	V8_REG27
5.13	5.10	5.58	7.93	7.55	7.97	7.94	20.49	19.10	16.80	12.03

Fig. 5. Simulator Speed.

The simulation speed is more than sufficient for rapid code development and debugging, but decreases linearly with Connex-ARM vector size. A GPU-enabled version of the software simulator was also developed. The GPU version is about 10 times slower that the CPU-only version for 128-element Connex-ARM vectors. The issue is the very small number of threads running on the GPU, between 128 and 256. GPUs only provide optimal speedup if enough threads are be issued to cover up memory latencies (*e.g.*, several thousands). Small kernels were also a problem since the effort of transferring their code is not matched by their execution time. The simulator transfers the kernel to the nVIDIA accelerator, then calls the kernel using one thread per simulated EUs. Results are transferred back after the kernel completes its task. The GPU simulator only becomes faster than the CPU simulator when Connex-ARM vectors are increased to thousands of elements.

4. Optimization Support

OPINCAA provides programmer support for software optimization through three components: the Performance Logger (OPL), the Performance Advisor (OPA), and the AutoTuner (OPAT). Figure 6 is an illustration of the design cycle using the various OPINCAA optimization components. Using the profiler, the programmer can get insight into critical sections and improve the implementation. The advisor analyzes the performance and makes suggestions regarding the design of the application, while the auto-tuner bypasses the programmer altogether and performs an automated design space exploration to find optimization opportunities. These components will be described in detail in the following sections.

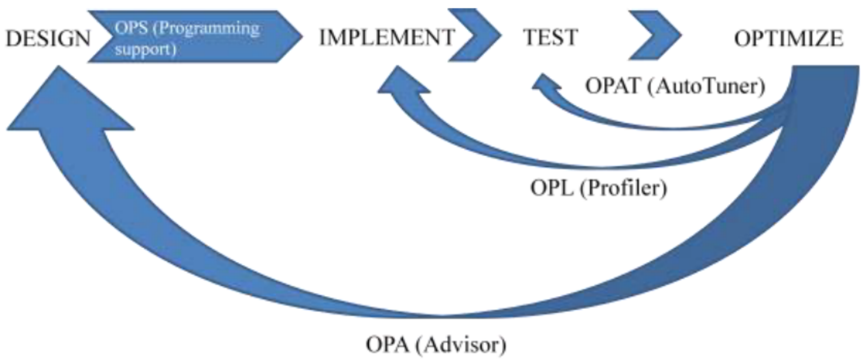


Fig. 6. OPINCAA Optimization Cycle.

4.1. Profiler

The profiler (OPL) measures Execution Manager activity and logs the duration of various sections of the application, in order to present a graphical report to the programmer. The profiler is an optional component, activated by a compilation directive, so that it can be removed to avoid a performance penalty. The user application is decomposed into the following operations:

- KCRE (Kernel Creation: memory allocation, instruction counting, kernel compilation if compilation is performed AOT – Ahead Of Time)
- KCPL (Kernel Compilation: instruction assembler; the task performed by OPINCAA's Instruction Generator and Instruction Injector)
- HOST operations (pre/post processing of data)
- IOWR (data transfers from the Host to the Accelerator)
- IORD (data transfers from the Accelerator to the Host)
- KRED (Kernel Reductions: transfer reduced data from the accelerator to the host)
- KEXE (Kernel Executions: transfer and run program over the accelerator)
- User-defined operation, available for the user to measure arbitrary operations or functions in the code. By default, it is mapped to trace the entire computation process, hence the name (COMP)

All these operations are logged with START / STOP events into the OPL's RAM trace. A report is printed on console or to a log file on demand. The file can then be inspected using the PLogViewer C# graphical application. The graphical break-down of execution into components makes it easy to determine how long kernel execution is taking or what sections can be executed in a parallel fashion. Figure 7 illustrates a 40 millisecond trace of the execution of a kernel. The kernel follows a write-compute-read pattern, and the relative durations of the sections can clearly be seen.

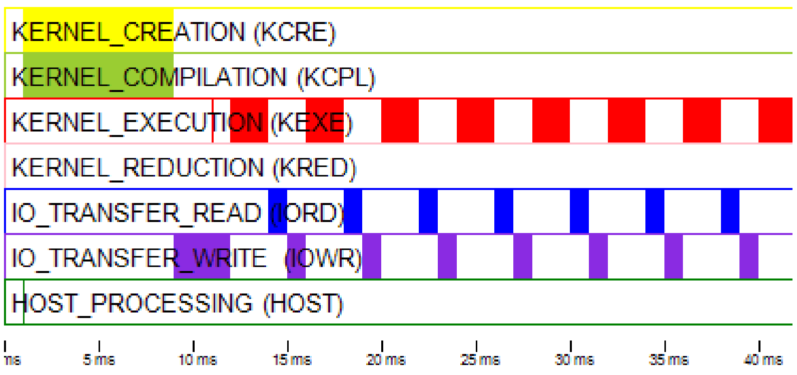


Fig. 7. OPINCAA Optimization Cycle.

4.2. Advisor

The OPINCAA Performance Advisor is designed to provide the programmer with hints as to where performance can be improved, by comparing the user application performance agains the maximum performance of the underlying Connex-ARM accelerator. The advisor uses OPL to profile the user application, and several carefully designed benchmarks that can expose the underlying hardware limitations. The benchmarks run either on demand, or right before application characterization. Results can be stored for future reference. The benchmarks relate to the following metrics:

- Kernel-Launch Delay: launches 2-instruction kernels in a loop and returns time in number of Connex-ARM accelerator instructions necessary for one simple kernel launch.
- IoReadSpeed: launches transfers (in a loop) from Accelerator to Host. This benchmark tries to find the best size (highest perceived bandwidth), depending on vector size.
- IoWriteSpeed: launches transfers (in a loop) from Host to Accelerator. This benchmark tries to find the best size (highest perceived bandwidth), depending on vector size.

OPA runs after an application has completed successfully and reports what metrics of the application are significantly below hardware capabilities. OPA gives advice on the following matters:

- Static register occupancy on the accelerator
- Dynamic IO transfer size

- Dynamic kernel size
- Static localstore occupancy
- Dynamic Unused Host time
- Dynamic performance ratio (Effective / Peak Performance)

As an example of a possible optimization advice, let us consider an application which requires the transfer of 384 vectors from the host to the accelerator, processes them and transfers them back to the host, on the Zynq device. The IoReadSpeed and IoWriteSpeed benchmarks yield the transfer speed characteristics in Fig. 8. Write speed depends on transfer size because of the particular size of DMA buffers, end-point FIFOs, and interrupt handling. Given the DMA performance information, the advisor may make the suggestion that the 384 vector write be split into a 128 vector write and a 256 vector write, for an average transfer speed of 206 MB/s, which is 11% better than the 185 MB/s achievable when transferring all the vectors in a single block.

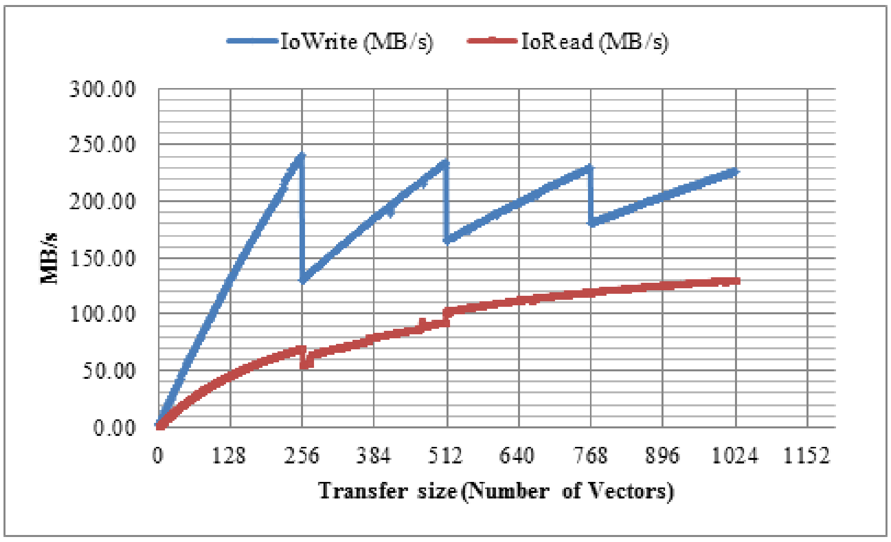


Fig. 8. IO Transfer Speed.

4.3. Auto-tuner

The OPINCAA Auto-tuner is a tool which enables the programmer to define a set of ranges for the parameters of the application, which the auto-tuner will explore in a brute-force manner. The auto-tuner will try all combinations of parameters within their respective ranges, and report the performance, in terms of runtime, for all combinations. Figure 9 presents the stages of OPAT operation.

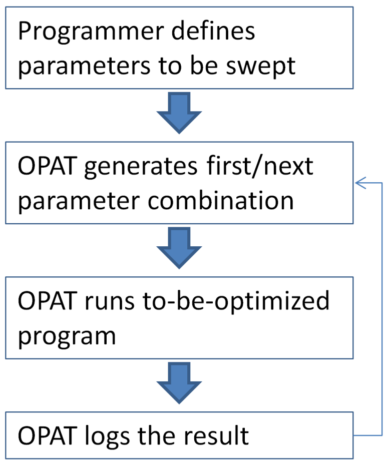

Programmer defines parameters to be swept
OPAT generates first/next parameter combination
OPAT runs to-be-optimized program
OPAT logs the result

Fig. 9. OPAT Operation.

5. Conclusions and Future Work

OPINCAA is a framework for the programming of Connex-ARM accelerators, from a C++ context, using ordinary C++ compilers on any platform. OPINCAA enables development of host and accelerator programs within the same source file, and exposes all the functionality of the Connex-ARM system to the programmer through vector data types and vector operators. Beyond the provided programming capability, OPINCAA provides a Connex-ARM simulator, which can be used instead of the hardware accelerator, on any platform, in order to develop and debug Connex-ARM applications.

Additionally, OPINCAA provides three components that help the programmer with performance optimizations: a profiler, an advisor which analyzes the code and the underlying Connex-ARM enabled platform and makes suggestions with regard to imcreasing performance, and an auto-tuner which enables the programmer to define parameters of his application, which the auto-tuner will explore in a brute-force manner to find the best-performing combinations of parameters.

While OPINCAA is fully functional, there are several avenues left to explore in future work. Section 4 described a GPU version of the instruction set simulator which is designed to perform well for large Connex-ARM vectors. Research is currently under way with regard to a dual-mode CPU/GPU simulator which is able to switch the method of simulation based on the parameters of the simulated kernels: vector sizes, kernel size, and others.

References

[1] AMD, Stream SDK (formerly ATI Stream), `http://developer.amd.com/gpu/AMDAPPSDK/Pages/default.aspx`, 2005.

[2] Bira C., Gugu L., Hobincu R., Petrica L., Codreanu V., Cotofana S., *An Energy Effective SIMD Accelerator for Visual Pattern Matching*, 4th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies, 2013.

[3] Firasta N., Buxton M., Jinbo P., Nasri K., Kuo S., *Intel avx: New frontiers in performance improvements and energy efficiency*, Intel white paper, 2008.

[4] Hong Sunpyo, Kim Hyesoon, *An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness*, ACM SIGARCH Computer Architecture News, volume **37**, pp. 152–163, ACM, 2009.

[5] Jang Minwoo, Kim Kukhyun, Kim Kanghee, *The performance analysis of ARM NEON technology for mobile platforms*, Proceedings of the 2011 ACM Symposium on Research in Applied Computation, pp. 104–106, ACM, 2011.

[6] Kirk D., NVIDIA CUDA software and GPU parallel computing architecture, ISMM, volume **7**, pp. 103–104, 2007.

[7] Lindholm E., Nickolls J., Oberman S., Montrym J., *NVIDIA Tesla: A unified graphics and computing architecture*, Micro, IEEE, **28**(2):39–55, 2008.

[8] Reinders J., *An overview of programming for Intel Xeon processors and Intel Xeon Phi coprocessors*, 2012.

[9] Santarini M., *Zynq-7000 EPP sets stage for new era of innovations*, Xcell Journal, **75**:8–13, 2011.

[10] Stone J.E., Gohara D., Shi G., *OpenCL: A parallel programming standard for heterogeneous computing systems*, Computing in science & engineering, **12**(3):66, 2010.