A Methodology for Efficient Use of OpenCL, ESL and FPGAs in Multi-core Architectures

Alexandros Bartzas and George Economakos

National Technical University of Athens
School of Electrical and Computer Engineering
Microprocessors and Digital Systems Laboratory
Heroon Polytechniou 9, GR-15780 Zografou, Athens, Greece
geconom@microlab.ntua.gr

Abstract. OpenCL has been proposed as an open standard for application development in heterogeneous multi-core architectures, utilizing different CPU, DSP and GPU types and configurations. Recently, the technological advances in FPGA devices has turned the parallel processing community towards them. However, FPGA programming requires expertise in a different field as well as the appropriate tools and methodologies. A feasible solution introduced recently is the adoption of ESL and high-level synthesis methodologies, supporting FPGA programming from C/C++. Based on high-level synthesis, this paper presents a methodology to use OpenCL as an FPGA programming environment. Specifically, the opportunities as well as the obstacles imposed to the application developer by the FPGA computing platform and the adoption of C/C++ as input language are presented, and a systematic way to explore both data level and thread level parallelism is given. The resulting methodology can be used for the deployment of parallel applications over a wide range of diverse CPU, DSP, GPU and FPGA multi-core configurations.

1 Introduction

Parallel processing can be considered starting at early 60s, with the D825 from Burroughs Corporation and Dijkstra's paper [3], while notions of parallelism can be found even earlier in the Analytical Engine of Charles Babbage. Since then, a lot of research and development has flourished the field, offering a variety of architectures, programming models, languages and standards. A latest development, mainly since the introduction by the computer gaming industry of ultra-high performance *Graphics Processing Units* (GPUs), is the integration of different processing elements (CPUs, DSPs and GPUs) under a common programming model like CUDA [9] and OpenCL [6].

OpenCL (Open Computing Language) is an open standard for the development of parallel applications on a variety of heterogeneous multi-core architectures, based on the C99 version of the C language. The execution model of OpenCL consists of a *host machine* connected and controlling a *compute device.* The compute device performs calculations with a number of parallel computational intensive *kernels.* Each compute device consists of *compute units* and each compute unit consists of *processing elements*, where single kernels are executed for specific sets of input data. Communication between processing elements is performed through the memory hierarchy. Each single kernel invocation is also called *work-item.* Work-items are organized into *work-groups.* Work-items have private memory for fast computations while work-groups have a block of local memory, common to all work-items. All work-groups have also access to global memory, which is used for host initiated initialization procedures.

OpenCL, since its introduction, has been reported to support different CPUs, DSPs and GPUs, in a variety of heterogeneous configurations. Recently, the technological advances in *Field Programmable Gate Array* (FPGA) devices, with hundreds of GFLOPs, maximum power efficiency and low cost, has turned the parallel processing community towards them, with a number of publications [11,12,5,8,10] proposing OpenCL as a programming language for FPGAs also. FPGAs, parallel processing and GPUs co-existed for many years, however they were considered isolated and disjoint fields, offering optimizations at different levels in system design. While indeed parallel processing is aiming at a higher optimization level than FPGAs, the main reason for this isolation has been the different programming models and languages used in each case. This is starting to change, as FPGA complexity is requesting higher level programming models for productive exploitation of their capabilities (millions of logic gates, thousands of advanced DSP blocks, hundreds of GFLOPS). Such models start from C level languages and mainly involve *High-Level Synthesis* (HLS) and *Electronic System Level* (ESL) design platforms [2], for the automatic translation of algorithmic into architectural design descriptions.

This paper presents a methodology for the adoption of OpenCL as an FPGA programming environment, based on the ESL platform CatapultC [4]. CatapultC accepts C/C++/SystemC behavioral untimed system descriptions that should follow specific coding guidelines, and through a number of directives (or GUI commands) applies HLS transformations to produce optimized bit-accurate *Register Transfer Level* (RTL) architectural descriptions. The methodology of this paper is a systematic application of each HLS transformation, by a meta-engine placing and tuning CatapultC directives into OpenCL code. The main concern in this process is that even though CatapultC can produce hardware from C, efficient hardware needs effort and architectural synthesis expertise. This expertise is coded in the meta-engine, which iterates through different possible and feasible HLS directive applications to generate optimal hardware implementations of OpenCL kernels. With this approach, the opportunities as well as the obstacles imposed to the application developer by the FPGA computing platform and the adoption of C/C++ as input language are investigated, and a systematic way to explore instruction-level, data-level and thread-level parallelism is given.

2 Related Research

A number of recent publications are considering using parallel programming models (OpenCL and CUDA) as a programming language for FPGAs also.

In [12] the authors present a detailed example, the MyBayes bioinformatics application, of using the CUDA stream-based programming model as an intermediate step for hardware design. All design steps are done manually and no specific methodology is reported. In [8] and [10] two methodologies are given for mapping OpenCL kernels to reconfigurable hardware. The methodologies involve compiler optimizations that map kernel code into fixed hardware templates, which are then written in hardware description languages. While both methodologies are complete and cover many different issues (computations, memory hierarchies and interfacing), the resulting hardware cores are template-based and do not cover in detail lower level design issues. In [5], the authors present another similar methodology, targeting *Application-Specific Processors* (ASPs). They use a custom design environment and map OpenCL kernels into either common or custom ASP instructions. Another approach, closer to this paper is reported in [11], where CUDA code is passed though another HLS tool. Directives and pragmas are used to control the tool but no systematic and iterative application is reported, as in the proposed methodology. HLS is rather considered as a single pass procedure. From the industrial point of view, FPGA vendors have been actively involved in the use of OpenCL for FPGA programming [1], offering specific frameworks that take advantage of the parallelism expressed in OpenCL code and generate template-based FPGA implementations.

3 Translation Methodology

The main idea of this paper is the proposal of a semi-automated methodology to translate OpenCL code into a form suitable for CatapultC, with which hardware is synthesized using HLS. Since OpenCL is based on C99, which is also recognized by CatapultC, this translation does not bring major changes to the input code. The whole process is performed by a custom source-to-source translator, that either infers (if possible) or accepts by the user (this justifies the term semi-automated) details to OpenCL code like pointer sizes, loop boundaries, input parameters and expected return values. The basic steps are the following.

- Each kernel is isolated and HLS synthesizes a hardware components for it.
- Pointers used as formal parameters in functions are converted to arrays with specific dimensions, for correct memory allocation.
- Return values are inserted as formal pointer parameters in the kernel function. This coding technique generates output registers for them.
- Barrier OpenCL instructions are converted into CatapultC I/O transactions with ready/acknowledge interfaces.
- Array sizes are enlarged to reach powers of 2, when feasible. This simplifies synthesis of memory access related hardware.
- Data types are changed into bit accurate and simulation efficient types supported by CatapultC.
- Conditional statements are supplemented so that all mutually exclusive paths are clearly defined. This helps CatapultC schedule them correctly.

- OpenCL specific directives are temporary removed. They are taken into account later, during system integration.
- CatapultC pragmas and directives are inserted. These pragmas and directives control all HLS transformations.

After translation, an iterative procedure is initiated, which works as a meta-engine modifying CatapultC pragmas and directives. At each iteration, which is performed with a predefined scenario (i.e. a loop's initiation interval is decreased by one in each meta-engine iteration), a new solution is produced. The meta-engine finishes when no new solutions can be produced (further modification of pragmas and directives produces invalid solutions) and the best solution with respect to performance and resource usage is selected for FPGA implementation. In the following subsections details about frequently used pragmas and directives are given.

3.1 Loops

Loops are the main source of optimization in algorithmic synthesis because most computations are performed within loops. Three are the main loop transformations in CatapultC, loop pipelining, loop unrolling and loop merging.

Loop pipelining is controlled by the initiation interval directive. This directive takes a numeric value argument and denotes that each loop iteration will wait that number of control steps before it starts. After that time, the first loop iteration partially overlaps the next and CatapultC generates a pipelined implementation which gets faster as the initiation interval directive gets smaller (minimum value is 1 control step). Our methodology starts with a value equal to the loop iteration latency (larger values would insert idle control steps) and decreases it as long as feasible solutions (valid with respect to the implementation technology) are found. Loop pipelining is a throughput optimization and gives better results if applied in the outer loops in nested loop configurations.

Loop unrolling is controlled by the unrolling directive. Each loop iteration, either unrolled or not, is considered by CatapultC to take at least 1 control step to finish. With unrolling, we investigate the opportunity to put more operations within this limit and lower the repetitions and thus, get faster hardware. Our methodology starts with an unrolling value of 2 and increases it until feasible solutions are found. Very long values tend to serialize the whole loop, which may prevent pipelining, so they are avoided. Loop unrolling is a latency optimization and gives better results if applied in the inner loops of nested loop configurations.

Loop merging can combine loops with identical bounds. Normally, CatapultC schedules consecutive loops one after the other, with no overlapping. If the loops however have identical bounds and data dependencies permit it, both loops can be executed in parallel, by merging their corresponding iterations.

3.2 Memories and Synchronization

CatapultC can map data objects either in register files or in memories. Small data objects can be mapped in register files, with very fast access times but more complicated control logic. Register files are implemented in FPGAs with *Look-Up Table* (LUT) elements. Large data objects can be mapped in memories with slower access times but less complicated control logic, like dedicated *Block RAM* (BRAM) in FPGAs, or off-chip. To control these options CatapultC uses a threshold directive. Data objects smaller than threshold are mapped into registers while objects larger than threshold are mapped into memories. Moreover, each data object can be forced to be mapped in either type of resource. In our methodology, data objects are mapped both in register files and memories and the best solution is chosen.

Memories in CatapultC have two properties that affect performance, the number of ports (single or dual port memories) and the number of interleaved blocks (1 or more) used. Both properties, controlled by appropriate directives, increase the number of memory accesses in a single control step. On the other hand, complicated memory configurations require complicated control logic. In our methodology both single and dual port memories are selected and interleaving is applied iteratively, until the best solution is found.

OpenCL uses barrier commands for synchronization. In our methodology, each barrier command is converted in a ready/acknowledge signal, which is controlled by the host. A kernel reaching a barrier command sends a ready signal to the host and waits acknowledge. Whenever the host receives ready from all kernels, it responds with the corresponding acknowledge signals.

3.3 System Integration

After all kernels have been synthesized through the proposed meta-engine into hardware blocks, the whole system can be realized (host and kernels). Two modes of execution are supported, as shown in figures 1 and 2. In figure 1, the FPGA device plays the role of a compute device and the host is an external workstation, connected through a high speed interface like the PCI-Express. Multiple FPGAs can be seated in separate boards (one board for each compute device), equipped with a PCI connector. Inside the FPGA, a PCI-Express core is responsible for the communication between the host and the compute device. Alternatively, in figure 2, the whole system (host and compute device) can be realized into the same FPGA, using an embedded processor as host. This solution offers faster connection between host and kernels (i.e a local bus), but since embedded processor ports of OpenCL are not widely available, it can be considered as a future extension. As an exception, in some simple cases, a small controller (manually designed from host OpenCL code) can play the role of the embedded processor.

Summarizing, the proposed methodology can explore different levels of parallelism. During hardware synthesis with CatapultC, instruction-level parallelism is explored by arranging individual instructions for best parallel execution. Data-level parallelism, either as *Single Instruction Multiple Data* (SIMD) or *Single Program Multiple Data* (SPMD) form, is explored by using the same kernel for all work-items and using either shared or local memory. Finally, thread-level parallelism in a *Multiple Instructions Multiple Data* (MIMD) form is explored by synthesizing different kernels and deploying them into different work-groups

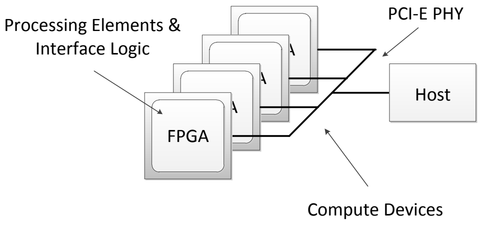

Fig. 1. FPGAs as compute devices

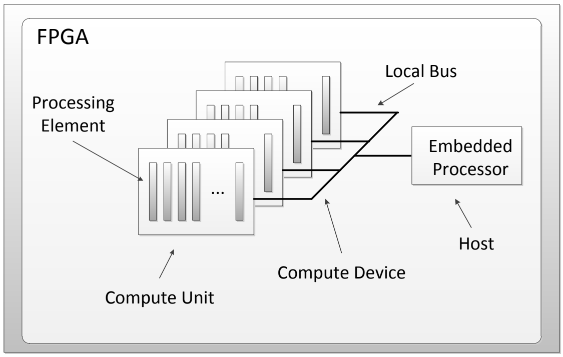

Fig. 2. FPGA as host and compute device

and/or work-items. This wide optimization space leaves many opportunities for system designers to achieve remarkable performance.

4 Experimental Results

The above presented methodology has been tested with a number of OpenCL kernels found in the NVIDIA OpenCL SDK version 4.1. Three kernels have been selected, parallel matrix multiplication, parallel discrete cosine transform (DCT) and parallel inverse discrete cosine transform (IDCT). Parallel matrix multiplication calculates 1 out of an 128x128 result while parallel DCT and parallel IDCT calculate 1 stage out of an 8x8 2 dimensional transformation. Part of the original DCT OpenCL code and the translated CatapultC code are given in the following two listings (omitting `C_x` DCT coefficient and constant declarations), where the great number of similarities as well as the coding guidelines of the previous section can be seen. Specifically, in the CatapultC listing (where all tool specific commands and directives are shown in boldface lines), the bit accurate `ac_int` and `ac_fixed` data types are used, the size of the `D` parameter of the `DCT8` function is inserted and the `hls_design` (meaning that each call corresponds to the same dedicated hardware component and not to an inlined routine), `hls_pipeline_init_interval` and `hls_unroll` directives are inserted with pragma declarations. In the outer `DCT8x8` hardware module, the hls_pipeline_init_interval is equal to 27, which is the minimum value found to generate a feasible solution.

Listing 1.1. OpenCL code

```
inline void DCT8(float *D){
  float X07P=D[0]+D[7]; float X16P=D[1]+D[6]; float X25P=D[2]+D[5];
  float X34P=D[3]+D[4]; float X07M=D[0]-D[7]; float X61M=D[6]-D[1];
  float X25M=D[2]-D[5]; float X43M=D[4]-D[3];
  float X07P34PP = X07P + X34P; float X07P34PM = X07P - X34P;
  float X16P25PP = X16P + X25P; float X16P25PM = X16P - X25P;
  D[0] = C_norm*(X07P34PP+X16P25PP); D[2] = C_norm*(C_b*X07P34PM+C_e*X16P25PM);
  D[4] = C_norm*(X07P34PP-X16P25PP); D[6] = C_norm*(C_e*X07P34PM-C_b*X16P25PM);
  D[1] = C_norm*(C_a * X07M - C_c * X61M + C_d * X25M - C_f * X43M);
  D[3] = C_norm*(C_c * X07M + C_f * X61M - C_a * X25M + C_d * X43M);
  D[5] = C_norm*(C_d * X07M + C_a * X61M + C_f * X25M - C_c * X43M);
  D[7] = C_norm*(C_f * X07M + C_d * X61M + C_c * X25M + C_a * X43M); }
// 8x8 DCT kernels
__kernel __attribute__((reqd_work_group_size(BLOCK_X,BLOCK_Y/BLOCK_SIZE,1)))
void DCT8x8(__global float *d_Dst,__global float *d_Src,
            uint stride,uint imageH,uint imageW) {
  __local float l_Transpose[BLOCK_Y][BLOCK_X + 1];
  const uint localX=get_local_id(0); const uint localY=BLOCK_SIZE*get_local_id(1);
  const uint modLocalX = localX & (BLOCK_SIZE - 1);
  const uint globalX=get_group_id(0)*BLOCK_X+localX;
  const uint globalY=get_group_id(1)*BLOCK_Y+localY;
  if((globalX-modLocalX+BLOCK_SIZE-1>=imageW)||(globalY+BLOCK_SIZE-1>=imageH))
    return; //Process only full blocks
  __local float *l_V = &l_Transpose[localY +         0][localX +         0];
  __local float *l_H = &l_Transpose[localY + modLocalX][localX - modLocalX];
  d_Src += globalY * stride + globalX; d_Dst += globalY * stride + globalX;
  float D[8];
  for(uint i = 0; i < BLOCK_SIZE; i++) l_V[i * (BLOCK_X + 1)] = d_Src[i * stride];
  for(uint i = 0; i < BLOCK_SIZE; i++) D[i] = l_H[i];
  DCT8(D);
  for(uint i = 0; i < BLOCK_SIZE; i++) l_H[i] = D[i];
  for(uint i = 0; i < BLOCK_SIZE; i++) D[i] = l_V[i * (BLOCK_X + 1)];
  DCT8(D);
  for(uint i = 0; i < BLOCK_SIZE; i++) d_Dst[i * stride] = D[i]; }
```

Listing 1.2. CatapultC code

```
#define INDEX ac_int <32,false>
#define DATA ac_fixed <32,16,true,AC_TRN,AC_WRAP>
#pragma hls_design
#pragma hls_pipeline_init_interval 1
void DCT8(DATA D[BLOCK_SIZE]) {
  DATA X07P=D[0]+D[7]; DATA X16P=D[1]+D[6]; DATA X25P=D[2]+D[5];
  DATA X34P=D[3]+D[4]; DATA X07M=D[0]-D[7]; DATA X61M=D[6]-D[1];
  DATA X25M=D[2]-D[5]; DATA X43M=D[4]-D[3];
  DATA X07P34PP = X07P + X34P; DATA X07P34PM = X07P - X34P;
  DATA X16P25PP = X16P + X25P; DATA X16P25PM = X16P - X25P;
  D[0] = C_norm*(X07P34PP+X16P25PP); D[2] = C_norm*(C_b*X07P34PM+C_e*X16P25PM);
  D[4] = C_norm*(X07P34PP-X16P25PP); D[6] = C_norm*(C_e*X07P34PM-C_b*X16P25PM);
  D[1] = C_norm*(C_a * X07M - C_c * X61M + C_d * X25M - C_f * X43M);
  D[3] = C_norm*(C_c * X07M + C_f * X61M - C_a * X25M + C_d * X43M);
  D[5] = C_norm*(C_d * X07M + C_a * X61M + C_f * X25M - C_c * X43M);
  D[7] = C_norm*(C_f * X07M + C_d * X61M + C_c * X25M + C_a * X43M); }
// 8x8 DCT kernels
#pragma hls_design top
#pragma hls_pipeline_init_interval 27
void ParallelDCT(INDEX stride, INDEX imageH, INDEX imageW, INDEX globalX,
     INDEX globalY, INDEX localX, INDEX localY, DATA *d_Src, DATA *d_Dst) {
  static DATA l_Transpose[BLOCK_Y][BLOCK_X + 1];
  localY = BLOCK_SIZE * localY; INDEX modLocalX = localX & (BLOCK_SIZE - 1);
  globalX = globalX * BLOCK_X + localX; globalY = globalY * BLOCK_Y + localY;
  if((globalX-modLocalX+BLOCK_SIZE-1>=imageW)||(globalY+BLOCK_SIZE-1>=imageH))
```

```
    return; //Process only full blocks
  else {
    DATA *l_V = &l_Transpose[localY + 0][localX + 0];
    DATA *l_H = &l_Transpose[localY + modLocalX][localX - modLocalX];
    d_Src=d_Src+globalY*stride+globalX; d_Dst=d_Dst+globalY*stride+globalX;
    static DATA D[BLOCK_SIZE];
    #pragma hls_unroll yes
    for(INDEX i = 0; i < BLOCK_SIZE; i++) l_V[i*(BLOCK_X + 1)] = d_Src[i*stride];
    #pragma hls_unroll yes
    for(INDEX i = 0; i < BLOCK_SIZE; i++) D[i] = l_H[i];
    DCT8(D);
    #pragma hls_unroll yes
    for(INDEX i = 0; i < BLOCK_SIZE; i++) l_H[i] = D[i];
    #pragma hls_unroll yes
    for(INDEX i = 0; i < BLOCK_SIZE; i++) D[i] = l_V[i * (BLOCK_X + 1)];
    DCT8(D);
    #pragma hls_unroll yes
    for(INDEX i = 0; i < BLOCK_SIZE; i++) d_Dst[i * stride] = D[i]; } }
```

Regarding kernel implementation, tables 1, 2 and 3 show the maximum performance achieved for each kernel (as throughput period in ns, the time required before a new input set can be processed by the resulting pipeline architecture) and the required hardware in terms of FPGA resources (Look-Up Table (LUT) function generators, D-type Flip-Flops (DFF), Block RAM (BRAM) and special purpose DSP blocks), both absolute numbers as well as percentages of the maximum available. For all implementations, the largest FPGA of the Xilinx Virtex-6 family was used, the 6VLX760 (with 758784 LUTs, 948480 DFFs, 720x36KB BRAM and 864 DSPs) at 200MHz.

Table 1. Parallel matrix multiplication

Sol.	Perf. (ns)	LUTs	DFFs	BRAMs	DSPs
S1	1295	85 (0.02%)	102 (0.01%)	0 (0.00%)	4 (0.46%)
S2	640	84 (0.02%)	102 (0.01%)	0 (0.00%)	4 (0.46%)
S3	320	113 (0.02%)	118 (0.01%)	0 (0.00%)	8 (0.93%)
S4	160	213 (0.04%)	191 (0.02%)	0 (0.00%)	16 (1.85%)
S5	80	335 (0.07%)	292 (0.03%)	0 (0.00%)	32 (3.70%)

In table 1, the first solution S1 corresponds to no optimizations selected. Solution S2 corresponds to initiation interval set to 1, while solutions S3, S4 and S5 keep this value and add an unrolling factor of 2, 4 and 8 respectively. In tables 2 and 3, solutions S1, S2 and S3 work directly with global memory and utilize fast BRAMs (nonzero in BRAM column), which is a common block for all kernels. This offers advantages at the circuit level (smaller memory controllers, less DFFs) but performance is low because of the large number of global memory accesses (barrier commands blocks every kernel before writing its result). Furthermore, in the same tables, solution S1 corresponds to no optimizations selected, solution S2 corresponds to initiation interval set to 4 (the minimum achieved), solution S3 corresponds to minimum initiation interval and full loop unrolling, solutions S4 and S5 are like S2 and S3 with double width local memories (64 bit I/O ports with 32 bit operands) and solution S6 is like S5 with subfunctions implemented as hardware components and not as inlined code.

In all tables, solution S1 or S2 is the worst implementation. All other solutions are sorted so that each one is better than the previous with respect to performance. Looking at resources, in many solutions less than 1% of the available hardware is used, so there is room to implement large number of kernels. The only resources that limit the number of kernels are the DSP blocks, which increase up to a significant percentage as more parallelization is attempted.

Table 2. Parallel discrete cosine transform

Sol.	Perf. (ns)	LUTs	DFFs	BRAMs	DSPs
S1	455	4158 (0.88%)	1702 (0.18%)	1 (0.14%)	37 (4.28%)
S2	640	4194 (0.88%)	2084 (0.22%)	1 (0.14%)	48 (5.56%)
S3	110	3563 (0.75%)	2354 (0.25%)	1 (0.14%)	23 (2.66%)
S4	30	3649 (0.77%)	2261 (0.24%)	0 (0.00%)	46 (5.32%)
S5	15	5273 (1.11%)	4339 (0.46%)	0 (0.00%)	62 (7.18%)
S6	10	5453 (1.15%)	6292 (0.66%)	0 (0.00%)	64 (7.41%)

Table 3. Parallel inverse discrete cosine transform

Sol.	Perf. (ns)	LUTs	DFFs	BRAMs	DSPs
S1	450	3002 (0.63%)	1688 (0.18%)	1 (0.14%)	38 (4.40%)
S2	800	4703 (0.99%)	2001 (0.21%)	1 (0.14%)	52 (6.02%)
S3	70	3331 (0.70%)	1859 (0.20%)	1 (0.14%)	34 (3.94%)
S4	35	2489 (0.52%)	1519 (0.16%)	0 (0.00%)	54 (6.25%)
S5	15	5329 (1.12%)	4259 (0.45%)	0 (0.00%)	56 (6.48%)
S6	10	5498 (1.16%)	5491 (0.58%)	0 (0.00%)	56 (6.48%)

From the above tables it is shown that a systematic directive application to each kernel code through the proposed meta-engine can produce good quality results in an automated way. In order to get an indication of the overall system performance improvement, we implemented a system based on the architecture of figure 2, with a simple controller to move data from global to local memories, using the slowest solution (S1 in table 2) and the fastest solution (S6 in table 2) of the DCT algorithm. Also, since these solutions differ in resource usage, for S1 16 parallel kernels where mapped onto the FPGA while for S6 8. Image sizes of 256x256, 512x512, 1024x1024 and 2048x2048 were selected and performance (as reported in CatapultC) at a 600MHz clock speed was compared with the

Table 4. Performance comparison between FPGA and GPU

Platform	Execution time (ns)			
	256x256	512x512	1024x1024	2048x2048
Virtex-6(S1)	662102	1216167	2324299	4540563
Virtex-6(S6)	399822	772103	1510840	2988349
Radeon	755398	1225752	2958031	10160484

OpenCL solution found in [7], based on the Radeon HD 6970 GPU at 850MHz. Execution times in ns are shown in table 4.

While the results of table 4 are not 100% objective (not many implementation details are given in [7]), they show a system level speedup ranging from 1.8x (256x256) to 3.4x (2048x2048). Also, S6 (fewer but optimized kernels) is faster than S1 (more but not optimized kernels), which is a justification of our approach. Better results are expected with the Virtex-7 FPGA family (offers twice the resources of Virtex-6), as soon as CatapultC libraries become available.

5 Conclusions

This paper presents a methodology for the adoption of OpenCL as an FPGA programming environment, based on the systematic application of HLS transformations by a meta-engine. The main concern in this process is that even though HLS tools can produce hardware from C, efficient hardware needs effort and some architectural synthesis expertise. This expertise, as shown with experimental results, is captured in the meta-engine, which iterates through different possible and feasible directive applications, and generates optimal hardware implementations. As future extensions, the use of both CUDA and OpenCL under the same environment is considered as well as the use of heuristics in the meta-engine iterations, to speed up the process and produce better results.

References

1. Altera Corporation, `http://www.altera.com/opencl/`
2. Coussy, P., Morawiec, A.: High-level Synthesis: From Algorithm to Digital Circuit. Springer (2008)
3. Dijkstra, E.W.: Solution of a Problem in Concurrent Programming Control. Communications of the ACM 8(9), 569 (1965)
4. Fingeroff, M.: High-level Synthesis Blue Book. Xlibris Corporation (2010)
5. Jaaskelainen, P.O., de La Lama, C.S., Huerta, P., Takala, J.H.: OpenCL-based Design Methodology for Application-Specific Processors. In: 10th International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, pp. 223–230. IEEE (2010)
6. Khronos Group, `http://www.khronos.org/opencl/`
7. Kim, C.G., Choi, Y.S.: A High Performance Parallel DCT with OpenCL on Heterogeneous Computing Environment. Multimedia Tools and Applications (2012)
8. Mingjie, L., Lebedev, I., Wawrzynek, J.: OpenRCL: Low-Power High-Performance Computing with Reconfigurable Devices. In: 20th International Conference on Field Programmable Logic and Applications, pp. 458–463. IEEE (2010)
9. NVIDIA Corporation, `http://www.nvidia.com/object/cuda_home_new.html`

10. Owaida, M., Bellas, N., Antonopoulos, C.D., Daloukas, K., Antoniadis, C.: Massively Parallel Programming Models Used as Hardware Description Languages: The OpenCL Case. In: International Conference on Computer-Aided Design, pp. 326–333. IEEE/ACM (2011)
11. Papakonstantinou, A., Gururaj, K., Stratton, J.A., Chen, D., Cong, J., Hwu, W.M.W.: FCUDA: Enabling Efficient Compilation of CUDA Kernels onto FPGAs. In: 7th Symposium on Application Specific Processors, pp. 35–42. IEEE (2009)
12. Pratas, F., Sousa, L.: Applying the Stream-Based Computing Model to Design Hardware Accelerators: A Case Study. In: 9th International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, pp. 237–246. IEEE (2009)