Automatic Offloading of C++ for the Cell BE Processor: a Case Study Using *Offload*

Alastair F. Donaldson
Oxford University
Computing Laboratory
Oxford, UK
Email: alastair.donaldson@comlab.ox.ac.uk

Uwe Dolinsky, Andrew Richards, George Russell
Codeplay Software Ltd.
45 York Place
Edinburgh, UK
Email: {uwe,andrew,george}@codeplay.com

***Abstract*—*Offload* C++ is an extended version of the C++ language, together with a compiler and runtime system, for automatically offloading general-purpose C++ code to run on the Synergistic Processor Elements (SPEs) of the Cell Broadband Engine (BE) processor. We introduce Offload C++ by presenting a case study using the approach to offload parts of an image processing application. The case study introduces the language extensions; illustrates the core technology on which the technique is based: automatic call-graph duplication, and automatic generation of data-movement code; shows how parallelism can be achieved by offloading work to multiple SPEs simultaneously, while the Power Processor Element (PPE) core simultaneously performs additional work; and demonstrates our solutions to dealing with complex language features such as function pointers and multiple compilation units.**

***Keywords*-Multicore programming; Cell BE; compilers; call graph analysis**

I. Introduction

In this paper we illustrate *Offload C++*, which is an extended version of the C++ language, together with a compiler and runtime system, for automatically offloading general-purpose C++ code to run on the Synergistic Processor Elements (SPEs) of the Cell Broadband Engine (BE) processor [1]. The Cell BE processor consists of a host core, the Power Processor Element (PPE), which is a regular CPU connected to main memory, together with eight SPEs – programmable vector processors each with 256K local RAM (referred to as *local store*), which can be accessed without contention, and the facility to move data between local store and main memory via direct memory access (DMA).

Essentially, Offload C++ allows the Cell BE to be programmed using a familiar threading paradigm: portions of code to be executed in an asynchronous SPE thread are enclosed in an *offload block*. This code is compiled for the SPEs; furthermore, automatic call graph duplication is used to compile all functions called (directly or indirectly) from an offload block for the SPEs. A distinction is made, at the language level, between pointers to data in host (PPE) memory, and data in local (SPE) store. This pointer information is propagated through the automatic call-graph duplication process, so that data-movement code is automatically generated. This means that the user does not need to write low-level DMA transfers by hand.

Technical details of Offload C++ are presented in [2]. The contribution of this paper is a practical case study showing how Offload C++ can be used to offload parts of an image processing application to run, in parallel, across all cores of the Cell BE processor. Through the case study we introduce the Offload C++ language extensions, explain how automatic call graph duplication works, and show how sophisticated language features such as function pointers and multiple compilation units are catered for. We also show how the system facilitates hand-tuning, allowing the developer to write specialised versions of key functions, specifically for the SPEs. This allows a step-by-step approach to program offloading, where functions are initially offloaded solely via call-graph duplication, then optimized by the programmer as desired.

We follow the case study with an experimental evaluation showing speedups obtained applying Offload C++ to a number of image processing benchmarks, and present a discussion of related work.

II. An image sharpening filter

Figure 1 shows C++ source code for an image processing filter which has the effect of *sharpening* a $W \times H$ image, computing output pixel at position (x, y) by applying the kernel of Figure 2 to the neighbourhood of input pixels centred at (x, y).

The source code is intended to run on the Cell PPE, thus colour pixels are represented as 4-wide `float` vectors (where the 4th element of the vector is unused) via the `float4` datatype which, together with overloaded operators including addition and multiplication, is supported by the PPE. The computed output pixel value is then *clamped*, so that each vector element lies in the range $[0, 1]$.

	-1	
-1	5	-1
	-1	

Figure 2. Sharpening kernel.

III. Offloading the filter to the Cell SPEs

A. Offloading to a single SPE

1) Offload blocks and offload threads: Our aim is to get the sharpening filter to run in parallel on the Cell SPEs. To begin with, we demonstrate how to offload the entire filter

```
float clamp(const float4 v) {
  v[0] = (v[0] < 0 ? 0 : (v[0] > 1 ? 1 : v[0]));
  ... /* same for v[1], v[2], v[3] */
}

float4 sharpen_pixel(const float4* pixels,
                      int i, int j) {
  const float4 minus1(-1.0f, -1.0f, -1.0f, -1.0f);
  const float4 five(5.0f, 5.0f, 5.0f, 5.0f);
  float4 dest =
    pixels[(i-1)*W+j] * minus1 +
    pixels[i*W+(j-1)] * minus1 +
    pixels[i*W+j]     * five +
    pixels[i*W+(j+1)] * minus1 +
    pixels[(i+1)*W+j] * minus1;
  return clamp(dest);
}

void sharpen(float4* input, float4* output) {
  for(int y=1; y<H-1; y++)
    for(int x=1; x<W-1; x++)
      output[y*W+x] = sharpen_pixel(input,y,x);
}
```

Figure 1. Sharpening filter.

```
void sharpen(const float4* input, float4* output) {
  offload_handle_t handle = __offload(input,output)
  { for(int y=1; y<H-1; y++)
      for(int x=1; x<W-1; x++)
        output[y*W+x] = sharpen_pixel(input,y,x);
  };
  offload_join(handle);
}
```

Figure 3. Sharpening filter offloaded to one SPE.

to run on a *single* SPE: this is achieved by enclosing the body of sharpen in an *offload block*, as shown in Figure 3.

The offload block is the prime language construct introduced by Offload C++, and consists of the **__offload** keyword, followed by a list of parameters (discussed below), followed by a lexical scope {...} which may contain arbitrary C++ code (requiring a few user annotations as discussed below, and limited by SPE local store size). Upon entering an offload block, the Offload C++ runtime launches an SPE thread, which executes the code inside the block *asynchronously*. The runtime immediately returns a handle to the PPE thread, which can be used to wait for the SPE thread to complete via a call to library function offload_join. We refer to SPE threads launched via offload blocks as *offload threads*.

2) Parameters to offload threads: Code inside the offload block of Figure 3 refers to input and output, which are parameters to sharpen, therefore they are located on the PPE stack. Because offload threads run asynchronously it is possible, in general (and often usual), for the calling function to return before the offload thread completes, or for the calling function to modify variables on whose initial values the offload thread depends. To avoid such complications, any stack variables which the offload thread requires from the enclosing scope must be passed as parameters to the block. This causes a copy of these variables to be passed to the offload thread on creation, avoiding problems with the original variables changing or going out of scope. Global variables need not be passed as parameters, as their lifetime is not bound to the stack frame of the PPE thread. If an offload thread really does require access to a variable v on the PPE stack (rather than to a copy of v) this can be achieved by passing a pointer to v as a parameter to the block.

3) Call-graph duplication and outer pointers: The offload block in Figure 3 calls sharpen_pixel, which in turn calls clamp. The Offload C++ compiler analyses the call graph rooted in an offload block, and compiles SPE versions of all methods. Thus code written for the PPE core can be re-used by offload blocks – it is not necessary to write separate versions of functions for the SPEs. Offload C++ uses an additional pointer qualifier, **__outer**, which indicates that a pointer inside an offload block refers to data declared outside the block, *i.e.* in PPE memory. Pointers declared outside offload blocks have the **__outer** qualifier by default, thus input and output are **__outer** pointers in Figure 3. Within an offload context, reading/writing via **__outer** pointer causes data to be transferred from/to main memory to/from SPE local store, whereas reading/writing via a standard pointer results in a normal local store access. Outer pointers enable the Offload C++ compiler to automatically generate data movement code: for example, writing to output[y*W+x] causes data to be transferred from SPE local store into this main memory array.

In Figure 3, when the compiler duplicates sharpen_pixel it does so considering parameter pixels (see Figure 1) as an outer pointer. This means that in the duplicate of sharpen_pixel compiled for the SPEs, reads via pixels correspond to operations to move data from main memory into SPE local store. If sharpen_pixel were called from elsewhere in the offload block with a standard pointer provided as the pixels parameter then the compiler would compile *another* duplicate of the method, this time where accesses via pixels correspond to local store accesses.

Figure 3 illustrates that, with very few changes, Offload C++ enables a portion of code to be offloaded to a Cell SPE. We did not need to use the **__outer** qualifier explicitly, change the core code for the sharpening filter at all, or write any data-movement code – all this is handled automatically by the system.

B. *Offloading to multiple SPEs* and *the PPE*

Offloading to a single SPE can be a goal in itself: this frees the PPE core so that other PPE threads can do useful work. However, for a parallelisable application such as our image filter (it is clear that image pixels can be processed independently), we can offload to multiple SPEs in parallel. In addition, to utilize the whole processor, we can get the

```
void sharpen_slice(const float4* input, float4*
          output, const int start, const int end) {
  for(int y=start; y<end; y++)
    for(int x=1; x<W-1; x++)
      output[y*W+x] = sharpen_pixel(input,y,x);
}

void sharpen(const float4* input, float4* output) {
  offload_handle_t handles[N];
  for(int i=0; i<N; i++) {
    handles[i] = __offload(input, output, i) {
      const int start = (H-PPE_PART)/N*i + 1;
      const int end = (H-PPE_PART)/N*(i+1) + 1;
      sharpen_slice(input, output, start, end);
    };
  }

  const int ppe_start = ((H-PPE_PART)/N)*N + 1;
  const int ppe_end = H - 1;
  sharpen_slice(input, output, ppe_start, ppe_end);

  for(int i=0; i<N; i++) {
    offload_join(handles[i]);
  }
}
```

Figure 4. Sharpening filter offloaded to multiple SPEs *and* PPE.

PPE to compute part of the output image. Code to achieve this is shown in Figure 4, where `N` is the number of SPEs to be used and `PPE_PART` specifies how many rows of the image should be handled by the PPE. A loop launches `N` offload threads, storing the resulting handles in an array. While these threads execute asynchronously, the PPE computes output for (in practice slightly more than) `PPE_PART` image rows. The PPE then uses the array of handles to wait for each offload thread to complete. Loop index `i` is passed as a parameter to the offload block so that each offload thread is given a distinct value of `i` which it uses to determine which rows to process.

The attractive feature here is that the functionality for the sharpening filter did not have to be manually duplicated to get the filter to run on both types of processor: automatic call graph duplication means that the PPE and SPEs can share the same source code.

Note that Offload C++ does *not* perform automatic parallelisation. In Figure 4, the image processing algorithm has been explicitly parallelised. Offload C++ automates the movement of data between main memory and SPE local store; the programmer must enforce race-free access to global data between threads, using locks if necessary.

IV. Offloading with function pointers

Suppose that our image processing application offers a number of effects, which are all applied on a pixel-by-pixel basis. Rather than having a separate method for each effect analogous to `sharpen` in Figure 4, it makes sense to parameterise the `sharpen` function of Figure 4 with an argument specifying which effect is to be applied (renaming the parameterised function to something more general). One approach would be to pass an integer specifying which effect is to be applied, and use a **`switch`** statement to call the appropriate function on each pixel. A more scalable approach is to pass a function pointer parameter through which a call can be made.

For Offload C++, the challenge here is to perform automatic call-graph duplication in the presence of function pointers. For a call via a function pointer from an offload scope to succeed, it is necessary for the function referred to by the pointer to have been compiled for the SPEs. However, the compiler cannot know, statically, which functions may be called via the pointer. It may sometimes be possible to derive function pointer targets via a "points to" analysis, but such analyses do not scale well for large applications; Offload C++ does not currently provide this facility.

The solution adopted by Offload C++ is to equip an offload block with a *domain*, specifying the names of functions which it is acceptable to call via function pointers from the offload block. The compiler duplicates all specified functions for the SPEs, and creates a table mapping PPE function pointer addresses to SPE duplicates. At runtime, this table is used to resolve function pointer calls, leading to a runtime exception if a call is made, via a pointer, to a function not specified in the domain. Requiring the user to provide the domain manually is not unreasonable: in practice, function pointers are typically used to call one of a particular class of functions, *e.g.* image filters in our example, or collision response functions in a video game, of which the programmer is well aware.

Figure 5 shows how our image processing example can be adapted to apply one of two filters, the sharpening filter and an additional greyscaling filter (for which we do not show source code), using a function pointer. The figure declares a function pointer type, `effect_t`, for pixel effects, and a parameter of this type is passed through the call-graph, and applied to individual pixels in `apply_to_slice`. To make this work, the offload block in `apply_effect` is equipped with a function domain `[...]` consisting of the pair of methods `sharpen_pixel` and `greyscale_pixel`. Because versions of these methods are required which accept an outer pointer as their first parameter, the function names are cast to `effect_outer_t`, which is the same as `effect_t` except the first parameter has type **`const __outer float4*`**. This domain information allows the Offload C++ compiler to duplicate suitable versions of these methods for the SPEs.

Function domains also support C++ virtual methods: for a virtual method in a base class, the domain can be used to specify a subset of derived classes implementing the method. Duplicates of these versions will be compiled for the SPEs, and may thus be invoked via a virtual call.

V. Support for multiple compilation units

To perform fully automatic call-graph duplication, the compiler requires access to *all* methods called (indirectly)

```
typedef float4 (*effect_t) (const float4*, int, int); /* Function pointer type for pixel effects */
typedef float4 (*effect_outer_t) (const __outer float4*, int, int); /* __outer version */

float clamp(...) { /* as before */ }

float4 sharpen_pixel(...) { /* as before */ }

float4 greyscale_pixel(const float4* pixels, int i, int j) { ... /* Performs greyscaling effect */ }

void apply_to_slice(effect_t effect, const float4* input, float4* output, const int start, const int end) {
  for(int y=start; y<end; y++)
    for(int x=1; x<W-1; x++)
      output[y*W+x] = effect(input,y,x); /* Filter applied depends on function pointed to by 'effect' */
}

void apply_effect(effect_t effect, const float4* input, float4* output) { ...
    /* __offload block now equipped with a domain, and 'effect' is passed to 'apply_to_slice' */
    handles[i] = __offload [ (effect_outer_t)sharpen_pixel, (effect_outer_t)greyscale_pixel ] /* Domain */
                           (effect, input, output, i) /* Parameters */ { ...
      apply_to_slice(effect, input, output, start, end);
    }; ...
  apply_to_slice(effect, input, output, ppe_start, ppe_end);
  ...
}
```

Figure 5. Function pointer used to apply either sharpening or greyscaling filter, offloaded to multiple SPEs and PPE.

from an offload scope. While many such methods may reside in the same compilation unit as the offload block, in a large application there will be multiple compilation units, and it is likely that a call graph will cross compilation unit boundaries. In our image processing example, it would be reasonable for `sharpen_pixel` and `greyscale_pixel` to be implemented in `sharpen_pixel.cpp` and `greyscale_pixel.cpp` respectively, with only their prototypes visible in corresponding header files. Suppose that `apply_effect` (*cf.* Figure 5) is in a separate source file, `main.cpp`. The compiler knows it needs to duplicate `sharpen_pixel` and `greyscale_pixel` as they are specified in the function domain for an offload block, but source code for these functions is not present in `main.cpp`.

Support for multiple compilation units is provided in Offload C++ by the `__dupicate` attribute, which may be applied to the prototype of a function. This attribute specifies that the function (and its call graph) should be duplicated for the SPEs, with a given configuration of outer pointers. This is illustrated in Figure 6: the function prototype for `sharpen_pixel` in `sharpen_pixel.h` is annotated with an attribute specifying that it should be duplicated as if it had been declared `float4 sharpen_pixel(const __outer float4*,int,int)`. A similar annotation is added to `greyscale_pixel.h`. Note that it is only necessary to annotate the boundaries of compilation units: once `sharpen_pixel` has been annotated as shown in Figure 6, any functions it calls which reside in `sharpen_pixel.cpp` will be automatically duplicated. For example, `sharpen_pixel` calls `clamp` in Figure 1. If these functions are both located in `sharpen_pixel.cpp` then the user will *not* need to additionally mark `clamp` for duplication.

VI. Tuning via SPE-specific optimizations

So far, we have shown that Offload C++ facilitates portability of source code between the PPE and SPE cores of the Cell BE processor. While this is a big win, for some applications it may come at the price of optimal performance. Once a portion of source code has been offloaded to an SPE in a portable manner, the programmer may wish to write specialised versions of certain key functions, geared towards the SPEs. Specialisation may involve re-organising control-flow structures to cater for lack of branch prediction, or may be involve making direct use of SPE intrinsics, such as vector operations or DMA calls.

In Figure 1, the `clamp` function operates on a vector in an element-by-element fashion. Suppose the programmer wants to write their own version of `clamp`, making use of two SPE intrinsics: `spu_cmpgt(`v_1, v_2`)`, which compares vectors v_1 and v_2 element-wise, returning a boolean vector b with $b[i] = \mathit{true}$ if and only if $v_1[i] > v_2[i]$, and `spu_sel(`b, f, t`)`, which returns a vector v such that $v[i] = f[i]$ if and only if $b[i]$ is false, and $v[i] = t[i]$ otherwise (where b is a boolean vector and f, t are numeric vectors). This can be achieved by declaring a separate version of `clamp`, prefixed with the **`__offload`** keyword, as shown in Figure 7. Functions prefixed with **`__offload`** (known as *offload functions*) are only compiled for the SPEs and can thus include SPE intrinsics which would be illegal in general-purpose code.

Offload functions can be used to optimize data-movement. By default, Offload C++ uses a software cache to manage reads/writes from/to host memory. While this is significantly more efficient than issuing many small DMA operations,

```
/* In "sharpen_pixel.h": */
float4 sharpen_pixel(const float4* pixels, int i, int j)
        __attribute__((__duplicate(float4 (const __outer float4*, int, int))));

/* In "greyscale_pixel.h": */
float4 greyscale_pixel(const float4* pixels, int i, int j)
        __attribute__((__duplicate(float4 (const __outer float4*, int, int))));
```

Figure 6. Multiple compilation units require duplication attributes.

```
__offload float4 clamp(const float4 v) {
  const float4 zero(0.0f, 0.0f, 0.0f, 0.0f);
  const float4 one (1.0f, 1.0f, 1.0f, 1.0f);
  float4 res = spu_sel(v, one, spu_cmpgt(v, one));
  return spu_sel(res, zero, spu_cpmgt(zero, result);
}
```

Figure 7. Accelerating using SPE intrinsics.

further efficiency can be gained for examples with regular data access patterns by using double-buffering to prefetch data. For performance-critical code, Offload C++ includes a set of header files which define templated stream classes, which provide a clean interface for reading/writing contiguous data, with an efficient underlying implementation using DMA intrinsics. Alternatively, the programmer can go further and opt to write low-level DMA transfers by hand, at the expense of PPE-SPE portability.

VII. Experimental evaluation

We present experimental results applying Offload C++ to a set of five image processing filters operating on a 512×512 pixel image, performing: embossing, sharpening, Laplacian edge detection, greyscaling, and noise reduction. Experiments are performed on a Sony PlayStation 3 console, for which 6 SPEs are available to the programmer. The original code executing on a single hardware thread of the Cell PPE is used as a baseline against which to compare versions where computation is offloaded to either 1 or 6 SPEs. Results are shown in Figure 8. For each offloaded benchmark, the figure indicates whether we have performed no additional manual optimization (naïve), optimizations to buffer output (buffered output), input and output (buffered I/O), or extensive manual optimizations (fully optimized).

The emboss filter computes each output pixel by applying a 3×3 kernel to the input image. Offloading the filter to a single SPE results in a performance decrease compared with serial PPE code, due to the latency introduced by per-pixel DMA transfers of inputs to the SPE *vs.* direct access to memory. This drop in performance is compensated for by adding SPEs: offloading to six SPEs results in a $3.27\times$ performance improvement.

Straightforward offloading of the sharpening filter to one SPE results in a six-fold performance decrease (not shown in Figure 8). Optimizing the offloaded code to use SPE vector operations, and using double buffering to transfer output pixels to main memory efficiently brings single SPE performance to within 75% of serial performance. Running this optimized version with all SPEs active results in a $2.96\times$ performance improvement over serial code. For this benchmark, input values are still read via the software cache, resulting in blocking DMA reads on cache misses.

For the Laplacian filter, which computes output pixels using a 5×5 kernel, we find significant improvements can be gained by avoiding use of the software cache via explicit pre-fetching of input data. By using explicit DMA intrinsics to maintain a copy of five rows of input pixels in SPE local store, and buffering output for transfer to main memory a row at a time, offloading to a single SPE out-performs the PPE version by $3.13\times$, with a $6.51\times$ performance improvement using 6 SPEs. The price for this is increased source code complexity, and loss of PPE-SPE portability. However, we were able to apply these manual optimizations incrementally, starting with a simple, non-optimized offload and gradually working towards a finely tuned version.

The greyscale filter computes a greyscale value from the RGB components of an individual pixel. While offloading naïvely to a single SPE results in a five-fold performance decrease, applying double buffering optimizations to both input and output, to interleave computation and data transfers, increases performance over the PPE code by $3.06\times$. The performance gain comes at a cost of code complexity – the optimized filter is $\sim$100 lines of code *vs.* $\sim$15 lines for the simple version. Performance does not scale well for this example: the speedup increases to just $3.44\times$ with 6 SPEs, this is due to the lightweight nature of the greyscaling computation.

Finally, the noise reduction filter computes an output pixel using a 16×16 kernel. Although this results in a lot of data movement per pixel, this example is compute intensive; thus offloading naïvely to a single SPE results in a performance decrease of just 15% compared with PPE code, with a performance increase of $2.76\times$ when six SPEs are used. Further performance could be obtained by applying manual optimizations as in the other benchmarks.

VIII. Related work

Of the recent wealth of multicore programming models, closest to Offload is HMPP [3]. To offload a C or Fortran function to run on a GPU using HMPP, the function is marked, using pragmas, as a *codelet*. The main difference

Filter	Emboss	Sharpen	Laplacian	Greyscale	Noise
	naïve	buffered output	buffered I/O	fully optimized	naïve
Speedup: 1 SPE	0.6×	0.76×	3.13×	3.06×	0.85×
Speedup: 6 SPEs	3.27×	2.96×	6.51×	3.44×	2.76×

Figure 8. Speedups relative to PPE versions for offloaded image processing filters, with one and six SPEs.

between HMPP and Offload is that HMPP codelets must obey a set of restrictions, *e.g.* a codelet must be a pure, non-recursive function which cannot access main memory and can only invoke other codelets, whereas Offload permits essentially arbitrary C++ code to be offloaded to an accelerator, and offloaded code can access host memory on demand. The restrictions of HMPP facilitate implementation on GPUs, and generation of highly optimized data movement code, but limit the technique to applications for which re-writing using codelets is feasible. In contrast, Offload can be applied to more general code bases, but this flexibility makes a GPU implementation of Offload challenging, and aggressive optimization of data movement code difficult.

The idea of optimizing data movement based on regularly structured data is the basis for the Sequoia [4] and CellSs [5] programming models, as well as stream programming languages such as StreamIt [6], Brook [7], CUDA [8] and OpenCL [9]. These models encourage a style of programming where operations are described as kernels – special functions operating on streams of regularly structured data. As with HMPP, exploiting regularity and restricting language features allows effective data movement optimizations; the drawback is that these languages are only suitable for accelerating special-purpose kernels that are feasible to re-write in a bespoke language.

Call-graph duplication is related to function cloning [10], used by modern compilers for optimizations such as inter-procedural constant propagation [11]. Automatic call-graph duplication applies function cloning in a novel setting, to handle multiple memory spaces in heterogeneous multicore systems.

Technical details of Offload C++ are presented in [2] in a more general context than for the Cell BE processor. This paper compliments [2] by providing a practical case study using Offload C++, concentrating on applications for Cell, for which an Offload C++ implementation is available.

IX. Conclusions and further work

We have demonstrated the process by which of Offload C++ can be used to parallelise an image processing benchmark across all available cores of the Cell BE processor, with relatively few source code modifications. Our experimental evaluation shows the speedups achieved by applying Offload C++ to a number of image processing benchmarks.

While the current implementation of Offload C++ is targetted at the Cell BE processor, the concept is more general, being applicable to multicore systems consisting of a host with accelerators. Thus an important avenue for future work will be to develop Offload C++ for other architectures, *e.g.* GPUs. A promising approach may be to compile Offload C++ applications into OpenCL [9] to allow deployment on any devices which implement this standard, and potentially allowing the use of multiple accelerators.

The "Community" edition of Offload C++ is available on-line, free of charge, from `http://www.codeplay.com`.

Acknowledgement

Alastair F. Donaldson is supported by EPSRC project EP/G051100.

References

[1] H. P. Hofstee, "Power efficient processor architecture and the Cell processor," in *HPCA*. IEEE, 2005, pp. 258–262.

[2] P. Cooper, U. Dolinsky, A. F. Donaldson, A. Richards, C. Riley, and G. Russell, "Offload automating code migration to heterogeneous multicore systems," in *HiPEAC*, ser. LNCS, vol. 5952, 2010, pp. 337–352.

[3] S. Bihan, G.-E. Moulard, R. Dolbeau, H. Calandra, and R. Abdelkhalek, "Directive-based heterogeneous programming – a GPU-accelerated RTM use case," in *CCCT*, 2009.

[4] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: programming the memory hierarchy," in *Supercomputing*. ACM, 2006, p. 83.

[5] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta, "CellSs: a programming model for the Cell BE architecture," in *Supercomputing*. ACM, 2006, p. 86.

[6] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "Streamit: A language for streaming applications," in *CC*, ser. LNCS, vol. 2304. Springer, 2002, pp. 179–196.

[7] I. Buck, "Brook specification v0.2," http://merrimac.stanford.edu/brook/.

[8] NVIDIA, "CUDA zone," http://www.nvidia.com/cuda/.

[9] Khronos Group, "The OpenCL specification," http://www.khronos.org/opencl.

[10] K. D. Cooper, M. W. Hall, and K. Kennedy, "A methodology for procedure cloning," *Comput. Lang.*, vol. 19, no. 2, pp. 105–117, 1993.

[11] R. Metzger and S. Stroud, "Interprocedural constant propagation: An empirical study," *LOPLAS*, vol. 2, no. 1-4, pp. 213–232, 1993.