From SMPs to FPGAs: Multi-Target Data-Parallel Programming

Barry Bond
Microsoft Research, Redmond
barrybo@microsoft.com

Andrew Davidson
University of California, Davis
aaldavidson@ucdavis.edu

Lubomir Litchev
Microsoft Research, Redmond
lubol@microsoft.com

Satnam Singh
Microsoft Research, Cambridge
satnams@microsoft.com

Abstract
Is it possible to devise a data-parallel programming model that is sufficiently abstract to permit automatic compilation from a single description into very different execution targets like multicore processors, GPUs and FPGAs? And if such a compilation process is possible what is the cost of such an approach compared to the alternative of designing specific specialized implementations for each target? This paper examines a high level data-parallel programming model which is designed to target multiple architectures. Using this model we describe the implementation of the core component of a software defined radio system and its automatic compilation into SSE3 vector instructions running on a SMP system; an implementation for GPUs that targets both NVidia and ATI graphics cards; and an implementation that targets Xilinx FPGAs. For the GPU case we compare the performance and design effort of our Accelerator based system with several hand coded CUDA implementations. Our goal here is to qualitatively evaluate the amount of programming effort versus achieved performance in order to assess the viability of improving programmer productivity by raising the level of abstraction of the data-parallel programming model.

1. Introduction
Data-parallel programming promises to provide an accessible model for mainstream programmers to exploit the benefits of parallel processing on multicore processors and GPUs [6]. Furthermore, more exotic computational devices like FPGA circuits provide a high degree of fine grain parallelism yet lack a viable programming model for regular software engineers. Much work on data-parallel programming for such platforms involves a great deal of architecture specific details. For example, efficient CUDA programs specify carefully the movement of data between shared (local) memory and global memory and efficient OpenMP programs make careful use of pragmas to manage shared information between multiple parallel loops. FPGAs have to be used by designing circuits using a hardware description language. Is it possible to devise a high level parallel programming model that abstracts away many architecture specific details and allows a single description to be compiled to multiple computational targets with reasonable performance? We present a system called Accelerator which we believe makes progress in this direction and help to raise the programmer’s productivity for writing software for parallel systems varying from SMP systems through GPUs to hardware circuits executing on FPGA chips.

The Accelerator system presented in this paper is similar to a previously described system [7] although the system described in this paper runs as native code (implemented in C++) and supports three targets: GPUs; SSE3 vector instructions on multicore processors; and Xilinx FPGAs. A significant difference between our DSL approach and other techniques is that JIT-ing is used to provide a simpler model of use than CUDA [1] and a library based approach allows us to be largely language neutral.

Unlike CUDA [1] or OpenCL the Accelerator system heavily abstracts the GPU computing layer. The user writes data-parallel programs for Accelerator by focusing on the inherent aspects of their data-parallel algorithm. Each target then has the job of efficiently mapping these target-independent descriptions onto a specific computing fabric. This approach has several advantages including ease of use (compared to CUDA-style descriptions) and the ability to map the same description to very different target architectures (e.g. GPUs, processors and FPGAs). A disadvantage of this approach is that we may not always get the most efficient implementation compared to a hand designed system where the user has exploited knowledge of a particular target architecture e.g. details about data-layout and data-movement.

The ability to write a computation once in a form that can be automatically compiled to three very different targets provides an important stepping stone towards a programming model for heterogeneous systems that mix regular multicore processors, GPUs adapted for general purpose computing and FPGA-based 2D computing surfaces.

2. Accelerator Programming Model
The Accelerator programming model provides multi-dimensional arrays and associated data-parallel whole array operations. Array elements may be floating point values (single or double precision), 32-bit integers or booleans. As a concrete example we show an F# Accelerator program that performs the point-wise addition of two arrays using a GPU:

```fsharp
open System
open Accelerator.ParallelArrays
let main(args) =
    let x = new FloatParallelArray (Array.map float32 [1; 2; 3; 4; 5])
    let y = new FloatParallelArray (Array.map float32 [6; 7; 8; 9; 10])
    let z = x + y
    use dx9Target = new DX9Target()
    use dx9Target = new DX9Target()
```

2010/10/4
When executed this program uses the GPU to compute the result array containing the elements 7; 9; 11; 13; 15.

Accelerator is a library which can be used from any programming language that supports foreign function calls to C. We have used Accelerator from C++, C#, F# and Haskell. Although the current implementation works under various versions of the Windows operating system it is in principle possible to port the multicores and FPGA targets to other operating systems. Our current dependency on DirectX 9 prevents an easy port of the GPU target although one could implement a new GPU target based on OpenCL.

Accelerator programs are represented as expressions over Accelerator data-parallel arrays. Accelerator provides constructors to convert a native array in C into a ‘wrapped’ data-parallel array which typically represents a data-parallel array in some other address space e.g. on the memory of a graphics card.

A selection of element-wise operations over 2D arrays is shown in Table 1. There are corresponding operations available over 1D arrays. A key distinction between Accelerator programs and other techniques for data-parallel programming (e.g. CUDA) is that Accelerator encourages the use of whole array operations. This is because arbitrary array indexing on many architectures is an expensive operation. For the element-wise operations the ‘loop’ is implicitly part of the operation being performed and this allows the Accelerator system to make decisions about how to automatically parallelize such operations.

Reduction operations operate over 2D arrays in a specified dimension and return a 1D array. We use 0 to denote reduction along rows and 1 to denote reduction along columns. Some reduction operations are shown in Table 2. For example, Sum(1) will sum up the elements in each column of a 2D array to return the corresponding 1D array of sums.

A very important class of operations provided by Accelerator are memory transforms which some of which are shown in Table 3. These operations give Accelerator the information it needs to understand memory access patterns which can in turn be used to generate efficient code and circuits. The Section operation computes a subset of a data-parallel array that consists of regularly spaced slices of the original array. The Shift operation returns an array which has its elements shifted (left, right, up, down) by a specified amount. Rotate cyclically permutes rows or columns. Replicate increases the dimensions in a data-parallel array by tiling with the original array. Expand increases the dimensions of a data-parallel array by wrapping elements from the original array. Pad also increases the dimension of an array by using a pad value to insert before and after each dimension. Transpose perform a matrix transposition on an array. An example of another operation is Stretch which increases the dimensions of a data-parallel array by replicating existing elements a specified number of times.

Array dimensions can be increased and decreased by rank changing operation as illustrated in Table 4. Accelerator also provides several other kinds ofoperations e.g. inner and outer product. Stencil-style computations [8] are examples of problems that map well to the Accelerator model.

The Accelerator system constructs an expression graph (DAG) in the heap which represents the desired data-parallel computation. The expression graph contains nodes for operations and memory transforms (e.g. see Figure 1). For on-line targets this expression graph is quickly JIT-ed into GPU code via DirectX 9 or into SIMD SSE3 code using our own customized JIT-er. The data-parallel computation can also be instantiated for an off-line target like an FPGA circuit.

Evaluation of an Accelerator expression follows these stages:

---

**Table 1. Examples of element-wise operations**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sum(0)</td>
<td>( R_i = \sum_j A_{i,j} )</td>
</tr>
<tr>
<td>Sum(1)</td>
<td>( R_i = \sum_j A_{i,j} )</td>
</tr>
<tr>
<td>Maximum value</td>
<td>( R_i = \max_j A_{i,j} )</td>
</tr>
</tbody>
</table>

**Table 2. Examples of reduction operations**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Section (( b, c, s, i, j ))</td>
<td>( R_{i,j} = A_{i,b} + b \times i, b_j + s_j \times j )</td>
</tr>
<tr>
<td>Shift (( m, n ))</td>
<td>( R_{i,j} = A_{i-m,j-n} )</td>
</tr>
<tr>
<td>Rotate (( m, n ))</td>
<td>( R_{i,j} = A_{(i-m) \mod M, (j-n) \mod N} )</td>
</tr>
<tr>
<td>Replicate (( m, n ))</td>
<td>( R_{i,j} = A_{i \mod m, j \mod n} )</td>
</tr>
<tr>
<td>Expand (( b, a, i, j ))</td>
<td>( R_{i,j} = A_{(i-b, mod M, (j-b) \mod N} )</td>
</tr>
<tr>
<td>Pad (( m, a, m, a, c ))</td>
<td>( R_{i,j} = \begin{cases} A_{i, \max(b, m) - n} &amp; \text{if in bounds} \ c &amp; \text{otherwise} \end{cases} )</td>
</tr>
<tr>
<td>Transpose(0,0)</td>
<td>( R_{i,j} = A_{j,i} )</td>
</tr>
</tbody>
</table>

**Table 3. Examples of transform operations for size \( M \times N \) arrays**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Drop Dimension (0)</td>
<td>( R_i = A_{0,j} )</td>
</tr>
<tr>
<td>Drop Dimension (1)</td>
<td>( R_i = A_{i,0} )</td>
</tr>
<tr>
<td>Add Dimension (1)</td>
<td>( R_{i,j,k} = A_{i,k} )</td>
</tr>
</tbody>
</table>

**Table 4. Rank changing operations**

---

![Figure 1. Expression Graph](image-url)
1. The program specifies an Accelerator computation by making library calls to build up an expression graph in the heap that denotes the required computation.

2. An evaluation function is called and given the expression graph as its argument. The first step involves target-independent optimizations (e.g. loop fusion) over an intermediate representation (IR) which we refer to as the IR graph.

3. The second phase involves target specific optimizations and code generation. For example, the Shift operation for a SSE3 multicore target might cause certain values to be shifted into and out of registers to implement a windowing operation. For an FPGA target the shift operation will result in the addition of extra shift registers and special address generation circuitry to help remember and reuse values read from memory.

4. The generated code is executed on the target (e.g. GPU or multicore processor) or for an off-line target the appropriate source code is generated (e.g. VHDL for the FPGA target).

5. For an on-line target the generated code is executed on the target and the result array is then transmitted back to the host C program and appears as regular C array.

Accelerator performs various optimizations on expression graphs including:

(a) common sub-expression optimizations: Accelerator detects commonly, often used sub-expressions in the graph and isolates them as a separate "sub-routine" (a tree of the forest of DAGs in the sequence of DAGs that need to be executed);

(b) constant folding;

(c) fusion of operations: this is the optimization that gives Accelerator biggest performance gain.

For example in the DAG Accelerator detects operations that can be fused within a single pixel shader or nest loop (a loop that iterates over all the data in the parallel array.) Using fusion Accelerator can fuse aggressively very non-trivial sets of operations. It is often the case that a programmer will not be able to see such optimization potential because of the complexity of the resulting data access/boundary patterns. If the data access pattern constitutes non-fusible operations Accelerator splits the DAG into set of sub-DAGs that are fusible. These sub-DAGs are executed sequentially although each sub-DAG internally is executed in parallel. The programming model itself (the building of DAG) guarantees the elimination of dead code. Accelerator doesn’t use SSA representation.

3. The DirectX9 GPU Target

The current GPU target for Accelerator works by dynamically JIT-ing pixel-shader code for the DirectX 9 (DX9) system. This system provides a processor independent technique for performing computations on almost any GPU. Data-parallel computations are written against an abstract pixel shader model and are elaborated dynamically onto the available of pixel shaders processors on a given graphics card. Programs written against the PixelShader 3 model have access to 32 temporary registers, 10 texture coordinate registers, 224 floating-point constant registers and 16 sampler registers with a minimum depth of 512 instructions per pixel shader. The design of the DX9 Accelerator target has been published elsewhere so we focus on the description of the SSE3 multicore target and the FPGA target which represent new work which makes Accelerator capable of programming heterogeneous multicore systems.

4. The SSE3 Multicore Target

The SSE3 multicore target dynamically generates vector instructions across multiple cores to implement a data-parallel Accelerator computation. The target uses SSE3 vector instructions which implement SIMD operations using special SSE registers which are vector registers available on the IA processors. These instructions allow the execution of instructions on packed data (calculations and comparisons) in parallel. An SSE register can pack and operate on up to 4 floats/integers and 2 doubles.

Currently the SSE3 multicore target of Accelerator works only on 64-bit processors. It dynamically generates the nest loops with the operations in memory and calls the generated code using x64 calling convention. The Registry Allocator uses all of the x64 OS available registers: 16 XMM and 16 GPR (General Purpose Registers). Figure 2 shows the architecture of the Accelerator SSE3 multicore target. The Common Framework of Accelerator creates a set of fusible operation DAGs that are passed to the SSE3 multicore target. For each DAG in the set of DAGs the multicore target does the following:

1. Split the input data between the cores available. The data is not necessarily split evenly between the cores.

2. Allocate Registers for each core. The registers are allocated in parallel for each core. Based on the memory layout of the split data, some core might require different numbers of registers to run the loop-nests. The register allocator is based on a Sethi-Ullman Register Allocator. There are result locations for each operation and data node (XMM registers or spilled memory locations). There are also base address and offsets for each data node and for the result (GPR 64 bit). The spilling of registers is executed in a separate pass. When a result location of an operation/data node is stack memory the algorithm finds the most remotely used register from this point in the children sub-DAGs and that is the register that is spilled into the stack memory location associated with the operation/data node. The register is not de-spilled until the operation that uses it is to be executed. This approach allows for reusing a spilled register if one is needed before we have to de-spill it. The Register Allocator works in conjunction with the Stack Manager that allocates spill memory locations for when needed.

3. Code Generation for each core. The code generated for each core for executing the DAG might be different based on the memory layout of the data used by the DAG. The code generator generates the loop nest for each core as well as emits the instructions for loading the input data in the XMM result register of the data nodes as well as executing the SSE instruction that implement the operations in the DAG. In cases where there are no SSE instructions that corresponds to specific DAG operation, the emitted code simply calls into the corresponding C language runtime function i.e. for SSE3 sin, cos, log2, pow, pow2 instructions. For better precision some operations also need to be performed on a extended representation of the data value i.e. in order to achieve the same precision in com-

Figure 2. Accelerator SSE3 Multicore Target architecture

3
plex operations as the non SSE execution multiplications and divisions on floats are executed by converting the floats to double, executing the operation on the double representation of the data and then converting back to float values.

4. **Execute the generated code for each core.** The executor simply creates one thread per core and runs the generated code for each core on the corresponding thread in parallel. Our experiments show that the Windows scheduler does a very good job of not migrating threads between cores.

### 4.1 Splitting Data between Cores

Accelerator follows a functional programming model. When a Data Node (a parallel array that wraps the user data) is created in the DAG at runtime, Accelerator creates its own copy of the data. The Accelerator’s copy is not affected by the further changes of the data by the program that created the Data Node. The data copy is 16 byte aligned to provide for using the faster, aligned XMM MOV instructions.

The result array, where the output of the operation is stored is allocated by the user. Accelerator can deal with non 16 byte aligned result allocation, but for best performance a 16 byte aligned allocated result arrays are advisable. As long as the data access pattern and boundary handling are the same, most operations are fusible into single nest loop for the multicore target. There are a few operations that are not fusible and are executed as separate sub-DAGs in the Accelerator program DAG. Among them are all the Reductions operations, and Inner/Outer product. Splitting of the data between cores for these operations is somewhat different than the rest of the sub-DAGs.

For all but the Reductions and Outer/Inner Products the data load is split between cores based on the result array size. In this case the numbers of elements in the result array are split between the cores in a way that each core is responsible for calculating more or less equal parts of the result. The beginning offset of the core’s part is always 16 byte aligned and the number of bytes from the result each core is responsible to fill is also a multiple of 16. The number of elements the last core calculates is most likely different than the rest of the cores and more often than not is not a multiple of 16. Figure 3 shows how splitting is done on a 2D array.

In case of Reduction and Inner and Outer Products the data load between cores is split based on the size of input parameters. In such cases each core is responsible for calculating a number of rows/columns from the input. Figure 4 shows possible splits of the data for 2D array and Figure 5 for a 1D array. In the case of 1D array each core calculates parts of the row and then the final result is calculated by combining the result of each of the core as appropriate.

### 5. The FPGA Target

The FPGA target compiles Accelerator data-parallel descriptions into FPGA circuits which can be executed on FPGA chips made by Xilinx. The compilation flow involves a sequence of target specific optimizations to map the data-parallel computation into efficient digital circuit data-paths which is statically scheduled. The Accelerator system encourages the use of whole array operations which allows us to make efficient address generator circuits.

The decision to use just whole array operations allows us to make several optimizations for laying data out in memory and the design of address generators e.g. for avoiding the unnecessary re-reading of data values. For example, the shift operation tells us how we can re-use data read from memory by adding appropriate delays, as illustrated in Figure 6.

Here we see how to optimize the implementation of expressions that include variables that shift a single data source by different values by adding delays for negative shifts and anti-delays for positive shifts. We can add further delays at the input to eliminate the anti-delays. The FPGA target performs this and many other optimizations which allow us to produce a very efficient 1D convolver from an straightforward description which has the architecture show in Figure 7.

The code one writes for the FPGA target is very similar to the code for the GPGPU DX9 target and the SSE3 x64 multicore target. However, the FPGA target is off-line i.e. computing the result of a data-parallel computation in the generation of VHDL source code files plus .XCO files for Xilinx’s Core Generator system for the instantiation of floating point cores. Each of the input arrays are compiled into BlockRAMs that are initialized with the corresponding
binary IEEE floating point representation of the input values. The generated circuit is statically scheduled highly pipelined data-path that reads simultaneously from multiple BlockRAMs, performs the required computation and stores the results in one or more output BlockRAMs. Alternatively, we can use the parameters mechanism of Accelerator to populate the input BlockRAMs from real-time values generated by sensors or other external devices.

The main reason the FPGA target is off-line is due to the fact that vendor place and route tools take a very long time to execute making a JIT-ing model infeasible. Furthermore details of how data is transmitted from the host computer to the FPGA system (or card) and back are not dealt with by the generic FPGA target and currently need to be dealt with manually by the user. As standard APIs are developed for host to co-processor communication we expect to be able to further abstract FPGA co-processors to the same level at which we have abstracted communication with the GPU subsystem and SSE3 vector code.

The reason we can generate efficient code for FPGA circuits is that we can exploit information in the expression graph about how data sources are accessed and what memory transforms are being applied e.g. in order, in reverse, transposed, with a stride etc. Furthermore, operations describing data shifts allows us to retain previously read values for reuse later which avoids unnecessary reads. A combination of such optimizations allows us to build efficient address generation circuits for BlockRAMs or off-chip memory which streams data quickly into our data-paths.

One design choice we had to face for the FPGA target involved how to schedule the movement of data through pipeline registers in our design. It is not always possible to directly feed the output of one computation (through wires and a single register) into the following computation because we need to balance pipeline delays to ensure each operator receives its inputs at the correct time and we also have to account for multi-cycle operations e.g. most floating point operations take around 10 cycles to produce their first result. Another reason we need to schedule data-movement is that we need to be able to implement memory transforms like strides and shifts efficiently without introducing control logic which would slow down the system and increase area and power. One approach would be to introduce hand-shaking signals to indicate when each operator is ready for new inputs and when its outputs are valid. However, our synthesis approach uses a library of operators that have fixed delays known at compile time. We can thus compute a static schedule which is simply implemented by adding the appropriate level of register delays by means of shift-registers. Furthermore, we ensure our shift registers are efficiently implemented on the FPGA through mappings into shift register LUTs or into BlockRAMs depending on the size and depth of the required shift register.

6. Case Study: Software Defined Radio

We use a non-synthetic software digital radio filter application for our target benchmarks, and portability tests. We also create various CUDA implementations of this application, each version containing an additional optimization that would require a higher understanding of the CUDA programming model and hardware implications. We use each of these variations to illustrate the extra effort involved in creating further optimizations, especially for a beginner in data-parallel computing.

Our Software Defined Radio (SDR) application has a good deal of innate parallelism available, but it is not embarrassingly parallel. Each filtration stage can be considered a convolution stage on a pre-defined input filter. The sampling data is downconverted for each carrier frequency we wish to process, this is done by means of equation (1), or separated into two parts as shown in equations (2) and (3). This downconverted signal goes through a filtration stage, similar to convolution. The resulting filtered signal is decimated by a discrete factor, and the resulting samples are filtered again. This process is illustrated in Figure 8, and all steps are performed in a data-parallel fashion.

\[
a_i = b_i * e^{-2\pi f i \Delta x/\Delta t}
\]

\[
a_i^{real} = b_i^{real} * \cos 2\pi i \Delta x/\Delta t - b_i^{imag} * \sin 2\pi i \Delta x/\Delta t
\]

\[
a_i^{imag} = b_i^{imag} * \cos 2\pi i \Delta x/\Delta t + b_i^{real} * \sin 2\pi i \Delta x/\Delta t
\]

If we buffer the sample data, and send them to the target architecture as packets. Each element in the buffer can be treated independently in both the downsample and decimation stages. There are some read data-dependencies in the filtration stage, which require boundary padding between buffer stages.

In our experiments we used two NVIDIA CUDA capable cards, as well as an ATI Radeon HD5870 which has 1600 stream processing units, 80 texture units a 2GB of memory. The NVIDIA cards used were a low-end Quadro FX 580 and a recently released GeForce GTX 470, which have 32 cores, peak performance of 108 GFLOPS/s, 25.6 GB/s bandwidth to global memory, and 240 cores, peak performance of 1088.64 GFLOPS/s, 133.9 GB/s bandwidth to global memory respectively. The new GTX 470 also contain many architectural changes that support a shared cache at every block, much faster atomic operations, and ECC hardware. The two CUDA capable cards represent two opposite ends of modern GPUs performance spectrum. Thus a performance comparison of our Accelerator implementation is highlighted in Section 9 and our various CUDA implementations highlighted in Section 8 demonstrate the scalability of both programming models. As an additional comparison, we show results of our SDR filter on an ATI Radeon HD5870 using DirectX9 as a backend target.

7. Accelerator Implementation

We illustrate the simplicity of our Accelerator implementation by showing actual sections of code which perform our filtration and decimation stage. As the Accelerator programming model uses...
a higher abstraction language such that the user is oblivious to hand-tuned parameters (e.g. CUDA thread and block parameters). The user therefore is only aware of vector, or matrix operations, allowing Accelerator to issue calls and optimize for each target in an automated fashion.

7.1 SDR Sample in Accelerator

The SDR implementation is in C++ using the Accelerator library. Below are a few simple lines of code that perform the filtration stage with N constants of our SDR in Accelerator:

```cpp
FPA Fres, Fres_im;
FPA input2D = Stretch(input, stretchFactor, 2);
FPA input2D_im = Stretch(input_im, stretchFactor, 2);
FPA apply = FPAt+2*PI;
Fres = input2D*Cos(apply) - input2D_im*Sin(apply);
Fres_im = input2D_im*Cos(apply) + input2D*Sin(apply);
fil = Fres+a[0];
fil_im = Fres_im+a[0];
for (i = 1; i < N; i++)
    { intptr_t shl = {0, -1};
      fil += ShiftDefault(Fres, 0.0f, sh, 2) * a[i];
      fil_im += ShiftDefault(Fres_im, 0.0f, sh, 2) * a[i];
    }
```

As shown in Figure 8, we do this filtration stage for both the real and imaginary components of the down-sample stage.

In contrast for each of our CUDA implementations, we dedicate an entire hand-written kernel to perform the filtration process. We must also organize threads and blocks, and for the optimized variations, we manage the intra-thread communication and shared resources.

The decimation stage from Figure 8 can be easily accomplished through one high-level Section call. This followed by another filtration stage means the core data-parallel code can be done in under ten instructions. This highlights one of the major advantages of a high-level abstract data-parallel language. The system manages the data-parallel resources automatically, allowing the user to quickly create working accelerated code. As highlighted in Sections 2 and 3, through the use of Directed Acyclical Graphs (DAGs), and Just In Time (JIT) compilation optimizations and scheduling on the target device are done automatically.

8. CUDA Implementations

In 2007 NVIDIA released its CUDA Programming Model, which exposed much of the available NVIDIA GPU hardware for programming. Previously shader languages were used for GPU Computing (also known as General Purpose GPU Computing). This was a step towards more flexibility which has resulted in a wide variety of algorithms which take advantage of these exposed features.

Unfortunately, tools and libraries to support this growing area have lagged behind. Though some exist, such as CUDPP, CUFFT, and CUBLAS, developers for the most part must develop their application from scratch [1]. Those new and unexperienced to fine-grained parallelism often struggle to exploit any speedups. The CUDA programming model also lacks portability, a program developed with CUDA can only be run on CUDA capable cards (currently only made by NVIDIA).

OpenCL is a promising new programming model that offers a similar close-to-the-machine, low-level abstract language, while also supporting portability among a variety of platforms and architectures. However, since the language’s abstraction layer is similar to that of CUDA, performance between platforms would suffer greatly unless the core-code is re-written or a higher-level abstract language is written over OpenCL to take advantage of the differences in architecture. As an example, low-level code written to target an NVIDIA GPU would not perform well when translated to multi-core code. For many algorithms (and even smaller functions) there would even be major performance differences between AMD GPUs and NVIDIA GPUs. Therefore, for portable code, we believe a higher-level abstract language would be preferred. Otherwise, code must be re-written with different optimization focuses for each target architecture.

We create several implementations of the SDR filter described in Section 6 using CUDA. There are four implementations, a baseline implementation that would be most obvious to a beginner data-parallel programmer, and three incrementally optimized implementations. In the following subsections we will go into more detail regarding our baseline CUDA implementation, and each variation (set of optimizations) we take. The variations we explore are:

1. Baseline. Most Straightforward Implementation: All data resident in GPU Global Memory
2. Variation 1. Use resident constant memory available in CUDA to store filter values. Saves time from global memory reads.
3. Variation 2. Use shared memory within a block such that there is a good amount of data re-use.
4. Variation 3. Allocate extra registers per thread that store intermediate values, rather than saving back to shared or global memory.

8.1 Baseline

Our baseline implementation takes the most direct route to solving the problem. This method stores the input and filter constants in global memory. Each thread is responsible for producing a subset of the necessary output values. Although it is the simplest way of solving the problem in CUDA, it is still much more involved than the Accelerator version. The programmer must manage thread accesses, ensuring reads occur in a coalesced manner, ensure intermediate values within the filter are not overwritten due to race conditions etc. which is no small feat for a non-expert GPU programmer.

Though our implementation ensures important things such as coalesced reads, and a balanced work-load, the reason it is considered the baseline performance is it does not take advantage of memory re-use through shared memory and constant memory.

![Figure 9. Classification of several data-parallel programming models.](image-url)
8.2 Variation 1: Shared Memory
The CUDA programming execution model groups threads into working blocks. Threads within a block can communicate with each other using a limited amount of shared memory, which is much faster than accessing global memory. Our first variation exploits this shared memory as a user-managed cache, a common optimization among experienced CUDA programmers. In our variation, each thread within a block first stores a set of input samples from shared memory into shared memory. Then, ensuring the reads occur in a bank conflict free manner, each thread performs the filtration stage accessing multiple samples from shared memory.

Therefore, if there are \( N \) filter stages occur we save \( N-1 \) global memory reads, replacing them with shared memory reads. This requires an extra synchronization point within our kernel, in order to avoid rewriting shared memory results once a thread is ready to process the next output to be filtered.

8.3 Variation 2: Constant Memory
Our next variation uses the per block constant memory available in the NVIDIA programming model. Constant memory is cached, so multiple reads of the same constants per block guarantee memory re-use and should therefore speed-up the kernel execution. In our case, each output, a sum of products between a set of inputs and filter values, can re-use filter values in constant memory. It also reduces register and shared memory use.

8.4 Variation 3: Intermediate Registers
Our final optimization involves the use of registers to store intermediate filter results. Saving intermediate data back into shared memory or global memory is very costly when faster read-write registers are available per physical thread. Though the performance improvement can be significant, these types of optimizations are often overlooked by even experienced CUDA programmers.

For example, the reduction primitive within the CUDA SDK saved intermediate values back into shared memory space, and then performed a block-wise log(t) reduction, where \( r \) is the number of threads per block. Using registers instead to store these intermediate values reduces the write time, at the cost of 'register pressure' to the multicore processor. However, for the filtration kernel, there are plenty of unused registers per multicore, allowing this optimization to have no 'register pressure' cost.

9. Performance Comparisons
We compared the performance of our four CUDA variations in our case study against our Accelerator implementation on our two CUDA capable cards, a Quadro 580 and a GTX 470. This direct comparison against the four CUDA variations serves as a general performance measuring stick. We also compared our implementation against a number of device architectures to measure the performance of our Accelerator code. As this is a streaming application, we experimented with a variety of buffer sizes and compared the throughput of the samples processed per millisecond for each implementation. This also means our SDR filter computation can be offloaded to multiple targets at the same time.

9.1 CUDA vs Accelerator Performance
Figure 10 gives a comprehensive performance comparison of our Accelerator version and our four tuned CUDA implementations. If for each implementation we compared the performance of its optimal buffer size we see that the Accelerator implementation is a little more than twice as slow as the most optimized CUDA implementation, but more than six times faster than the naïve implementation and first optimization. However, for the newer line of cards, once the CUDA thread parameters have been hand-tuned, all CUDA implementations outperform the Accelerator implementation. Also of interest, the CUDA baseline kernel, and first two variations have nearly identical performance.

This is due to NVIDIA’s new architecture which supports global memory caching such that the use of shared memory and constant memory as a user-managed cache is not as critical for maintaining performance. However, the use of intermediate registers in Variation 3 still gives a significant speedup over the other three CUDA kernels. In general, the CUDA kernels outperform Accelerator by 1.5 for the first three kernels, and by about 4.5x for the fastest kernel on the latest card. Though for the newest cards, Accelerator performance does not outperform the CUDA implementations, we argue that due to its little effort for high reward speedups over conventional CPU or multi-core code, Accelerator is a good model for inexperienced data-parallel programmers.

10. Accelerator Multi-Target Performance Comparison
Comparing the Accelerator implementation on multiple targets shows that GPU architectures are well suited for SDR filtration as illustrated in Figure 11. Another significant observation is the relatively small difference in performance between different GPU generations. As an example, Table 5 shows that the Quadro FX 580 and GTX 470 have an order of magnitude difference in peak performance, and five times difference of global memory bandwidth. The fact that Accelerator’s performance does not change very much between these two cards, illustrates a need for Accelerator to improve its data-parallel target implementation if it wishes to remain a viable solution for beginner data-parallel programmers who wish to exploit up-to-date hardware.

The reason for Accelerator’s poorer performance on the GTX 470 in comparison with CUDA is Accelerator’s current use of the DirectX 9 texture target for computation. NVIDIA’s architecture has focused on speeding up access to global memory, especially if this is done in a coalesced fashion. As our SDR CUDA kernel implementations are all optimized for coalesced reads and the new architecture inherently supports caching, there are no advantages to using DirectX 9’s texture memory. We believe that if Accelerator’s GPU back-end targets were changed from DirectX 9 to DirectX 11 (using DirectCompute), or a further matured OpenCL, the Accelerator performance for GPUs would scale much better with current architecture trends. As this is quite possible to do under Accelerator’s dynamic library model, future work is needed to test multiple GPU back-end targets such as OpenCL and DirectX 11 whilst maintaining a high level abstract language like Accelerator. Even with this current disadvantage, the speedups on Accelerator in comparison with single-core or multi-core systems is quite significant and easy to achieve. Along with this coding effort advantage, the portability of the system allows the programmer to target multiple platforms at the same time. In the SDR radio example, it is possible to have a multi-core system process a set of signals concurrently with the GPU processing another set of signals with no change to the code-base.
For the multicore SSE3 experimental results shown in Figure 11 we used two different machines. The four core machine has one Intel Xeon X5550 processor which contains four cores running at 2.66GHz with 12GB of memory under a 64-bit version of Windows 7 Enterprise. The 24 core machine has four Intel Xeon E7450 processors running at 2.4GHz with 32GB of memory under a 64-bit version of Windows Server 2008 Enterprise R2. The result shows steady improvements in performance as the sample buffer size is increases for most of the targets.

We also produced an FPGA implementation for the same benchmark computation. The FPGA target generates a VHDL file which implements the core data-path and it also creates project side files to automatically generates floating point cores for addition, multiplication and the CORDIC computation of sine and cosine as well as cores that convert between floating and fixed point number representations (the CORDIC core uses fixed point). We targeted a Xilinx high performance Virtex-6 XC6VLX240T FPGA which is available on the Xilinx ML-605 development board which is one of the systems available in our laboratory. We synthesized the real and imaginary parts of the system separately and report the results for just the real part (the results are identical for the imaginary part). Using the Xilinx ISE 12.1 tools we generated a circuit which operates at 317MHz i.e. it will produce 317 megasamples per second. The real part of the design uses 10,204 look-up tables (LUTs) which represents just 6% of the available area on the chip. The imaginary part would use a further 10,204 LUTs or its computation can be interleaved with the real part at the cost of halving the throughput. The design uses 108 DSP48E1 components (out of 768) and these are used to make high speed floating point operations to implement floating point addition, subtraction, multiplication and also to implement high speed fixed point operations for implementing sine and cosine operations. The Xilinx XPower tool estimated the power consumption at 2.7W at ambient temperature of 25°C which is a small fraction of the power consumed by the corresponding multilcore or GPU system. For the XCVLX240T FPGA we can achieve an aggregate performance of 1,100 megasamples per second using 7 instances of the core circuit (interleaved, limited by the number of DSP48E1 blocks) compared to a peak performance 552 megasamples per second using the fastest CUDA implementation on the NVIDIA 470GTX card. We also implemented the same circuit on a low cost low power Spartan-6 series X6SLX75T FPGA (layout shown in Figure 12 with used cells marked in light blue) which produces a design which can process 143 megasamples per second with an estimated power consumption of 0.636W.

These results show that a single Accelerator description of the filtering and decimation core operations of the software radio filter produce effective implementations for three very different targets. Each target is valuable in a different context e.g. depending on price, performance and power consumption.

11. Related Work

The CUDA system [1] provides an off-line approach for compiling data-parallel descriptions of kernels written in a special language which has to be compiled used a special compiler for NVIDIA. In contrast, our data-parallel descriptions can be written in any language that has interop with C and compiled with any C or C++ compiler. Furthermore, our model is on-line for the GPGPU and multilcore targets whereas CUDA is always off-line.

The closest related work to Accelerator is the RapidMind [8, 9] system which has been acquired by Intel. Intel’s Ct system [2] has also been used for option pricing [3]. There has also been work at the bytecode level for off-loading computations to a GPU [5].

12. Future Work

The GPU performance of our system is limited by DirectX 9 which does not model the memory hierarchy available on modern GPUs. Developing a GPU target based on DX11 DirectCompute or OpenCL would allow us to significantly improve performance whilst remaining vendor independent. The FPGA target can be extended to allow the user to specify resource constraints by specifying priority for area conservation or performance or power consumption. We also plan to produce an end-to-end SDR system extended to 2D signals (i.e. video).

13. Conclusions

This paper shows how it is possible to use an embedded domain specific language to ease the process of writing parallel software for multiple execution platforms by permitting a high level compilation from a single data-parallel description. Furthermore, we used the core of a software defined radio example to measure both the difference in development effort and the difference in performance compared to a hand coded version in CUDA. Our conclusion is it is possible to achieve significant and meaningful performance improvements from a high level model like Accelerator whilst retaining a good level of programmer productivity. However, for absolute performance it will always be better to produce specialized versions e.g. in CUDA for GPUs, using SSE3 intrinsics for SSE3 vector instructions and hand coded circuits in VHDL or Verilog for FGPA.

For certain kinds of data-parallel descriptions it is possible to devise a abstract language of parallel operations and embed this language into a concrete language like C++ or F# and then use a JITting model to dynamically generate code for GPU and multicore targets. The same descriptions can also be compiled to FPGA circuits although we have to use an off-line model because vendor place and route tools are too slow. The ability to express a data-parallel computation once and then have it automatically compiled to three different targets is a very useful capability for the exploitation of many-core heterogeneous systems.

References

Figure 10. CUDA vs. Accelerator benchmarks
Figure 11. Benchmark comparisons for Accelerator GPU and multicore SSE3 targets

Figure 12. Layout of the real part of the SDR core circuit on a Spartan-6 X6SLX75T FPGA