Automatic Implementation of Evolutionary Algorithms on GPUs using ESDL

Steve Dower
Swinburne University of Technology
http://stevedower.id.au

***Abstract*—Modern computer processing units tend towards simpler cores in greater numbers, favouring the development of data-parallel applications. Evolutionary algorithms are ideal for taking full advantage of SIMD (Single Instruction, Multiple Data) processing, which is available on both CPUs and GPUs. Creating software that runs on a GPU requires the use of specialised programming languages or styles, forcing practitioners to acquire new skills and limiting the portability of their developments. In this paper, we present an automatic translation from ESDL, a domain-specific language for composing evolutionary algorithms from arbitrary operators, to C++ AMP, a C++ extension for targeting heterogeneous hardware. Generating executable code from a simple platform-independent description allows practitioners with varying levels of programming expertise to take advantage of data-parallel execution, and enables those with strong expertise to further optimise their implementations. The automatic transformation is shown to produce code less optimal than a manual implementation but with significantly less developer effort. A secondary result is that GPU implementations require a large population, large individuals or an expensive evaluation function to achieve performance benefits over the CPU. All code developed for this paper is freely available online from http://stevedower.id.au/esdl/amp.**

I. Background

A. General-purpose GPU

It has been recognised for many years now that while Moore's law[1] currently still holds, the implication of exponentially increasing CPU clock speeds does not [1], [2]. With physical limitations (such as the speed of light) constraining the ability to process data more quickly, chip manufacturers now provide parallelised devices that process more data simultaneously. The rise of SIMD (Single Instruction, Multiple Data) instruction sets and processor packages with multiple cores have enabled CPUs to provide ever-increasing power at the expense of generality. Not every algorithm can be parallelised, and few to an extent that provides a linear performance increase with the degree of parallelisation. Algorithms that are best suited for parallelisation typically process every element of a large ($N > 10^5$) data set independently. [3]

Despite the prevalence of multi-core CPUs, another approach gaining popularity for highly parallel applications is the use of GPUs (Graphics Processing Units). While high-end desktop CPUs are capable of a degree of parallelism up to thirty-two,[2] even entry-level graphics cards provide parallelism in the hundreds or thousands. The difference between the two is the complexity of operations that may be performed on each element, which reflects the distinct purpose and intent of each type of processor. CPUs are designed for extended, non-linear sequences of complicated instructions on linear sequences of data; GPUs are intended to perform simple operations, typically arithmetic and rarely branching, to large arrays of data. While both forms of processor are theoretically capable of performing any task, in practice, selecting the right processor to use for a task is necessary to achieve optimal performance.

Early use of GPUs for computation involved cleverly defining and using texture data and shaders. More recently, manufacturers have recognised the growing use of the GPU architecture for non-graphical applications and have started providing interfaces for general-purpose computing. Popular examples include CUDA for devices made by NVIDIA, OpenCL for any manufacturer providing an implementation for their platform, and DirectCompute, based on Microsoft DirectX. Each of these is based on writing "kernels"—programs to execute on the GPU—that are loaded and invoked at runtime from a CPU-based program. Each kernel is designed to perform an operation on a very small subset of data, typically one element, and the GPU architecture allows thousands of instances of this kernel ("single instruction") to execute simultaneously on a full array ("multiple data"). Kernels are typically written in relatively low-level languages, the equivalent of developing in C for embedded devices, although support for the abstractions common to higher-level languages is improving. [4], [5]

B. C++ AMP

C++ AMP (C++ Accelerated Massive Parallelism) is a library and a language extension for C++ that supports development for heterogeneous platforms, including GPUs. The library provides a common interface for allocating and transferring memory between the CPU and GPU, iterating over multi-dimensional collections of data and distributing algorithm execution across heterogeneous hardware, while a single language extension, the `restrict` function modifier, allows GPU kernel definitions to

[1]The prediction, made by Intel co-founder Gordon Moore, that the number of transistors that can be placed on an integrated circuit doubles every two years.

[2]The Intel i7 processor has four cores with HyperThreading and SSE, allowing 32 simultaneous floating-point operations, provided each of the eight threads perform the same operation to four values at a time. Pipelining and out-of-order execution may increase this further. Obtaining this level of parallelisation is every bit as complicated as it sounds.

```
#include <amp.h>
using namespace concurrency;

void add_arrays(int N, float* A, float* B, float* C) {
    array_view<float, 1> vA(N, A), vB(N, B), vC(N, C);

    parallel_for_each(vC.extent,
        [=](index<1> i) restrict(amp) {
        vC[i] = vA[i] + vB[i];
    });
}
```

Fig. 1. Array addition in C++ AMP

```
FROM random_binary(length=10) SELECT 100 population
YIELD popluation

BEGIN generation
  FROM population SELECT 100 parents \
       USING fitness_proportional
  FROM parents SELECT offspring USING crossover, mutate

  FROM offspring SELECT population
  YIELD population
END
```

Fig. 2. Simple Genetic Algorithm in ESDL

intermingle with CPU code written in C++ rather than requiring separate files and syntax. [3], [4], [6]

C++ functions marked with `restrict(amp)` are limited to features supported by the instruction set of GPU processors. Notable limitations are the lack of virtual methods and polymorphism, variable references and direct invocations of code located on the CPU. All functions used in a kernel must be able to be compiled inline and be marked with `restrict(amp)`. While using polymorphic class hierarchies is not possible, templates and generic programming can be used to provide a similar form of compile-time type dispatch. User-defined types may be used as array elements and accessed within kernels using normal C++ syntax, including overloaded operators. The C++ AMP specification clearly specifies all the limitations [6].

C++ AMP does not automatically provide superior performance to CUDA or OpenCL; the value proposition is developer productivity. By analogy, C++ does not provide better performance than assembly language, though the productivity benefits are immense. The first step in such an abstraction is to allow the developer to design for their data, rather than the hardware. C++ AMP iterates over an array, while CUDA activates a large number of threads that select data based on their identifier. This abstraction allows a C++ developer to implement an algorithm using C++ AMP quickly, while optimisations for thread and memory layout may be added later if desired or necessary. In contrast, a CUDA or OpenCL developer must deal with these complexities in order to create a working algorithm.

Figure 1 shows a trivial example of adding two arrays using C++ AMP. The use of C++11 lambdas[3] is a convenience but not mandatory, and apart from the `restrict(amp)` modifier, there are no modifications to the C++ language; `array_view`, `index` and `parallel_for_each` are defined in the `concurrency` namespace.

At the time of writing, the only available implementation of C++ AMP is based on DirectCompute, limiting its use to computers running Microsoft Windows; other compiler developers are working on implementations for their own platforms. The DirectCompute implementation is available as part of Microsoft Visual Studio 11 Beta.[4]

[3]Lambda functions were introduced into C++ with the new standard in 2011 and are already supported by most compilers. [7]

[4]Details and downloads are available at http://go.microsoft.com/fwlink/?LinkId=190957

C. ESDL

Evolutionary System Definition Language (ESDL) [8]–[10] is a domain-specific language for describing evolutionary algorithms. It is based on a unified model of evolutionary algorithms viewed as a composition of independent *operators* and *groups* of individuals. For example, a simple Genetic Algorithm (SGA) consists of a selection operator and two variation operators, crossover and mutation, connected through three groups: population, parents and offspring. This model is not restricted to simply substituting one operator for another of similar type, but allows arbitrary connections between any types of operator. ESDL has been used to describe a wide range of existing algorithms without loss of fidelity, and often a significant improvement in clarity. [8]

While operators are treated as the processing mechanism, groups handle the storage and management of individuals. There is no way (nor need) to access individuals except by applying an operator to an entire group to produce a new group. The resulting operator network is a simple description of the algorithm structure independent of any particular operator's behaviour. In effect, operators are implemented while algorithms are simply composed.

Figure 2 shows an ESDL definition for an SGA. The `FROM-SELECT` statement is the main way in which groups are created, either from a *generator* (a parameterised, virtual group containing infinite individuals) such as `random_binary` or operators like `fitness_proportional`. The `YIELD` statement indicates the group containing potential solutions, typically the main population or equivalent, but which may be any one or more groups used by the algorithm. Full descriptions of ESDL are given in [8] and [9].

With all implementation abstracted into operators, the algorithm description itself is an accurate specification of the system that is independent of any particular software. As well as forming a clear textual description, an ESDL system can be parsed and executed by a software framework, as in `esec`.[5] New algorithms can be implemented and shared by specifying an ESDL system and the behaviour of any custom operators. A reader interested in using the algorithm can manually translate the ESDL to their language of choice, or use a supporting compiler or framework to do the transformation automatically.

This paper describes a transformation from ESDL to

[5]A Python-based EC experimentation framework that uses ESDL to define algorithms, available online at http://esec.googlecode.com/

C++ AMP that may be applied automatically or manually. The transformation provides a systematic and efficient approach to exploiting the benefits of GPU hardware for running evolutionary algorithms without requiring extensive experience in data-parallel development. `esdlc`[6] has been extended to support this transformation and the generated code has been validated and benchmarked against alternative implementations.

II. Evolutionary Algorithms and GPU

Various approaches have been used when restructuring evolutionary algorithms for implementation on a GPU. Most implementation experience to date has concentrated on optimisations associated with the sequential CPU, while data-parallel algorithms generally require significant reconceptualisation. Many efforts move particular operations, typically evaluation, onto the GPU. For applications where the evaluation phase consumes the majority of execution time, this approach can be very successful, since evaluation is typically independent for each individual. However, the overhead of copying data between the GPU and CPU may become significant, particularly since the entire algorithm must halt until evaluation is complete. [11], [12]

Where evaluation is not parallelisable, not a bottleneck, or the overhead of copying data is prohibitive, implementing the entire algorithm as a single GPU kernel or a sequence of linked kernels may be more appropriate [13], [14]. The use of island populations or neighbourhoods allows parallelism to be achieved without significantly changing an existing implementation [15], though directly transferring a sequential algorithm to a parallel architecture is not the best way to utilise the architecture [2]. In particular, most algorithms for random number generation and sorting are not data-parallel and translate very poorly to a GPU, often resulting in the use of tournament style selection rather than approaches requiring complete ordering (as in [14], [16]). Further, since a kernel must be applied to a significant number of data segments simultaneously—typically in the thousands for modern GPUs—to achieve useful parallelism, a complete data parallel approach is necessary to obtain any benefit.

The approach used here is based directly on the compositional nature of ESDL. Each operator is represented as a C++ class that produces a stream of individuals from another stream—an array of individuals from an existing array. C++ AMP allows array references to be passed between kernels without copying via the CPU, making it simple to implement each operator independently and compose them efficiently, and a deferred execution model reduces kernel invocation overheads. Standard or common implementations of pseudo-random number generators (PRNGs) and sorting algorithms are not yet available for C++ AMP; here we use a naïve implementation of a bitonic sorting network [17] (based on [18], without the overlapping work allocation described by [19]) and a PRNG based on a hybrid Linear Congruential Generator and Combined Tausworth Generator [20]. Verifying the optimality of these algorithms is beyond the scope of this paper; this work is concerned with the transformation from the high-level ESDL model to code for heterogeneous computing platforms.

[6]A compiler for ESDL, supporting multiple targets including Python and C++ AMP, available online at http://esdlc.googlecode.com/

```
template<class T>
class Operator {
    T source;
public:
    Operator(T source);
    array<float,2>* operator()(int count);
    array<float,2>* operator()();
};
```

Fig. 3. C++ interface for operator classes

III. From the Model to the Processor

The two main ESDL concepts that require representation in C++ AMP are groups and operators. Some flexibility available within ESDL is restricted, as much to encourage proper use of data-parallel execution as to simplify the transformation task. For example, in order to achieve the best performance from GPU-based operators, individuals need to consist of fixed-length arrays of either integers or real numbers. In effect, groups are best represented using an `array<float, N>` class.[7] Such a restriction is not inherent in ESDL, which is agnostic towards representation, but allowing individuals that are more flexible ultimately detracts from the performance benefits of SIMD execution. Here, we store fitness values for each individual as an extra element in an individual's array; using a completely separate array may have performance benefits, though these are not investigated here.

Operators represent a streaming behaviour and are applied sequentially to the sequence of individuals that flow into the new group. An upper limit on the size of this group may be specified, which maps elegantly to a pull model: each operator asks its predecessor for the number of individuals needed, until eventually the source group provides the actual individuals.[8] This model is necessary to support operators that discard individuals, such as crossovers that produce a single offspring for each pair of parents, and is also convenient for working with operators that potentially produce infinitely long streams. Each operator only needs to supply as many individuals as requested—when n individuals are requested from an operator, it might request $2n$ from its predecessor. Where necessary, operators can request 'as many as available,' either to allow aggregation of the entire source stream or because no size was specified for the destination group.

For our implementation, an operator is a class that provides the interface shown in Figure 3: a constructor taking the

[7]Representations need not be so limited; our implementation supports variable length individuals (with a predefined maximum) and user-defined structures. For the purposes of discussion, however, assuming a linear genotype—`array<float, 2>`—is a useful simplification.

[8]The alternative, a push model, has each operator provide as many individuals as it can to the subsequent operator. This is a useful conceptualisation for a distributed system, but cannot be implemented efficiently in a limited-memory environment.

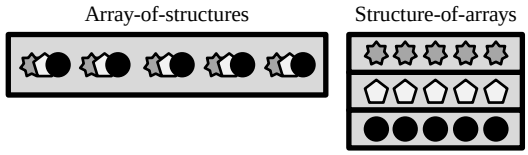

Fig. 4. Two approaches to storing parallel streams of data

TABLE I
SUMMARY OF MAPPINGS BETWEEN CONCEPTS AND CLASSES

Concept	Class
Groups	`array<float, 2>*`
Joined groups	`array<float, 2>*[]`
Operators	`Operator` (Figure 3)
Generators	`Operator` (with no source)

preceding operator (and operator-specific parameters) and two overloads of the 'call' operator: one with and one without a size specifier. The first operator in any chain is the merge operator, which concatenates a variable number of `array<float, 2>` instances; all other operators—in particular, those defined by a user—can assume that their source is a callable operator. Those operators that are known to produce infinite-length results may omit the parameterless overload in order to produce a compilation error rather than an out-of-memory condition at runtime. Because templates are used, inheritance from an actual `Operator` base class is unnecessary and implementing matching functions is sufficient. The `operator()` methods are shown as returning pointers to `array` instances, since returning by value would perform deep copies on the GPU. Our actual implementation uses reference-counted pointers (`shared_ptr`) rather than raw pointers.

The ability to join groups with the `JOIN-INTO` statement is broadly intended for two purposes: to *associate individuals* for combined evaluation and to *associate groups* for processing by specially designed operators [8]. While both are theoretically equivalent, the first is more efficiently represented using an array of structures and the latter by a structure of arrays (Figure 4). A structure of arrays is more computationally efficient to create because it does not require reprocessing groups, and is better for Differential Evolution's (DE's) difference-vector mutation operation, which uses separate streams of individuals to perform an operation resulting in a regular group. Only the structures of arrays form (as an array of `array` instances) is used by our C++ AMP implementation. Table I summarises the C++ types used to implement each ESDL concept.

ESDL is based around the idea of operator composition, which makes it important that users are able to provide new operator implementations. Interoperability concerns between C++ and ESDL include the different function invocation syntax, primarily in C++ using positional rather than named parameters, insufficient information in C++ prototypes for use from ESDL and templated object instantiations requiring more information than an ESDL compiler has available. Our solution involves three parts: first, ESDL pragma commands (lines beginning with a backtick `) are used to include header files containing operator definitions. These files are scanned for specially formed comments that provide a code generation template including parameter names and default values. Finally, as well as an implementation of the `Operator` type shown in Figure 3, a separate instantiation function (a factory function, rather than a constructor) is provided, allowing parameter types to be inferred.[9] This function is invoked by code generated based on the ESDL definition and the comment specifiers. Figure 5 shows an outline of a header file containing a mutation operator for DE, and Figure 6 shows a complete DE system in ESDL that may be compiled for either `esec` or C++ AMP.

Any language-specific transformation from ESDL must deal with five constructs: *stores*, *blocks*, *yields*, *functions* and *pragmas* [8]. Stores represent each `FROM-SELECT` and `JOIN-INTO` statement and create ("store") a new group or groups from one or more existing groups and a chain of operators. Blocks are delimited by `BEGIN` or `REPEAT` and `END` statements and represent a sequence of operations that are repeated: one named block (such as `GENERATION` in Figure 6) is executed each iteration, while `REPEAT` blocks are executed a given number of times. Yields notify an external listener that the specified group should be evaluated and used for statistical analysis. Functions allow externally provided code to perform arbitrary functionality, such as writing debugging output, immediate statistical analysis or parameter adaptation. Finally, pragmas specify compiler-specific commands.

Our automatic transformation of ESDL uses pragmas to include user-defined operators and to specify a default evaluator. The default evaluator is applied to groups that are created from generators and propagates to other groups through stores. User-defined operators and functions are specified in C++ header files and included using an `` `include `` pragma that is converted to a C++ `#include` statement. Only directly included files are parsed for signature specifications.

Function calls and operator constructions are realised as direct calls to the name as provided in the signature specification. Some overload resolution is performed by the converter based on parameter names, while type-based overloading may be performed by the C++ compiler. Parameters are mapped into the locations they appear in the specification, substituting default values where necessary. Figure 7 shows an example of defining and calling a function from ESDL, including the C++ code generated for the call.

Repeated blocks are implemented using a `for` loop while named blocks use a cascaded `if-else` structure to select the block to use for the current iteration.

Since store operations provide the compositional part of algorithms specified using ESDL, they comprise most of the generated code. A store operation consists of three steps: merge or join the source groups, construct the chain of operators and take the individuals required for the destination groups. To provide merging, a special operator is constructed that takes an arbitrary number of groups as sources. One generator may

[9]C++ infers parameter types on function calls but not type definitions, constructors or static methods.

```
template<typename T>
class mutate_de_t {
    T source;
    float scale;
    float cr;
public:
    mutate_de_t(T source, float scale, float crossover_rate)
    : source(source), scale(scale), cr(crossover_rate) { }

    array<float, 2>\* operator()() {
        array<float, 2>\* individuals = source();
        ...
    }

    array<float, 2>\* operator()(int count) {
        array<float, 2>\* individuals = source(count);
        ...
    }
};

// ESDL operator: mutate_de(scale=(float)0.8,crossover_rate=(float)0.9)
template<typename T>
mutate_de_t<T> mutate_de(T source, float scale, float crossover_rate) {
    return mutate_de_t<T>(source, scale, crossover_rate);
}
```

Fig. 5. ESDL prototype and C++ function for a DE mutation operator

```
`include "mutate_de.h"
`include "rastrigin.h"
`evaluator Rastrigin()
length = 10

FROM random_real(length, lowest=-2, highest=2) SELECT (size) population USING unique
YIELD population

BEGIN GENERATION
  FROM population SELECT (size) bases USING fitness_sus(mu=size)

  JOIN bases, population, population INTO mutators USING random_tuples
  FROM mutators SELECT trials USING mutate_DE(scale, crossover_rate=CR)

  JOIN population, trials INTO trial_pairs USING tuples
  FROM trial_pairs SELECT population USING best_of_tuple

  YIELD population
END
```

Fig. 6. An ESDL system for DE

```
// ESDL function: max(a=(float)0, b=(float)0, c=(bool)false)
float max(float a, float b, bool c);

// Invocation in ESDL; named, case-insensitive
// with implicit 'A=A' parameter.
k = Max(B=5, A, C=TRUE)

// Generated invocation in C++
auto k = max((float)A, (float)5, (bool)true);
```

Fig. 7. Example function specification and invocation

be included as the last source; its infinite size ensures that no later groups or generators will ever be used. Invoking this merge operator takes the requested number of individuals from the concatenated sources. By abstracting the sequential nature of repeated taking (such that each request is fulfilled from adjacent but non-overlapping segments of the source groups) other operators do not need to reimplement this functionality. Code generated for the first step of a store operation produces an instance of the merge operator with one or more groups or generators, as shown on line 3 of Figure 8.

The second step chains the listed operators together, applying each to the result of the previous. Each operator is a function call that takes the preceding operator instance, any other parameters and returns an instance of the operator. Depending on the purpose of the operator, instantiation may perform all required processing or defer it until individuals are requested. For example, a sorting operator can evaluate the previous operator to produce a group, sort and return it immediately, while a variation operator might choose to retain the original source operator and only process individuals as requested to avoid unnecessary computation.

For `JOIN-INTO` statements, each source group is wrapped in a separate merge operator and passed to the join operator ("joiner"). A joiner must be specialised for the number of sources it can accept, which allows operator designers to restrict the application of a joiner with a minimum of runtime

```
1 // FROM population SELECT (size) bases
2 //      USING fitness_sus(mu=size)
3 auto _src_0 = merge(population);
4 auto _stream_0 = fitness_sus(_src_0, (int)size);
5 auto bases = _stream_0((int)size);
```

Fig. 8. C++ code generated for an ESDL store operation

```
1 // JOIN bases, population, population INTO mutators
2 //      USING random_tuples
3 auto _stream_1 = random_tuples(merge(bases),
4                                merge(population),
5                                merge(population));
6 auto mutators = _stream_1();
```

Fig. 9. C++ code generated for a store operation that used a joiner

overhead.[10] Only one joiner may be applied in a `JOIN-INTO` statement, and as a result, no operator chaining is required. An example of the code generated for a `JOIN-INTO` statement is shown in Figure 9.

The final step for all store operations is to create and store the new groups. Individuals are taken from the last chained operator using one of the two `operator()` overloads shown in Figure 3. Both figures 8 and 9 (at lines 5 and 6, respectively) use these methods with the C++ `auto` keyword to create and store the resulting groups.

IV. Performance Benefits

To show the possible benefits available from mapping ESDL to compilable code, we compare both the performance and development effort for five implementations of Differential Evolution (DE) [21]. The algorithm is identical (DE/rand/1/bin) in each case, and the results are similar though not identical, due to differences in random number generators. DE individuals are vectors of 10 real values, initially between -2.0 and 2.0, and a population of 10 000 individuals was used; large enough to provide reliable measurements, despite being biased towards making the GPU implementations perform better (and far too large for a ten-dimensional problem. This bias is revisited later).

The Rastrigin benchmark function:

$$f = \sum_{i=1}^{n} \left(x_i^2 - 10\cos\left(2\pi x_i\right) + 10\right)$$

is used to determine fitness, Stochastic Universal Sampling is used for fitness-proportional selection of bases and all degenerate vector selections are allowed. A scaling factor of 0.8 and a crossover rate of 0.9 are used. (These are not intended as an ideal parameter selection for DE; we are concerned with execution performance, and DE uses a sufficiently complex set of operations to produce interesting results.)

Performance is measured in milliseconds per iteration, including random number generation and statistics collection. The time taken to display results to the screen is not included. Code complexity is determined from our implementations using Halstead's complexity measures, though we point out that we have avoided the effort and development time calculations and focused on a relative comparison of the variety and repetition of operations and symbols in a piece of code [22].[11] Only code that an end user would need to write is included; compilers and standard evolutionary operators are not assessed. These values for complexity are not mathematically useful, but are an indication for readers who are not familiar with how difficult developing for a particular platform or language may be. Algorithmic complexity is irrelevant, since each implementation is of the same algorithm.

TABLE II
Comparison of relative implementation complexity and speed with 10 000 individuals

	Target	Complexity	Speed (ms/gen)
(a)	ESDL (esec)	1	430.
(b)	ESDL (C++ AMP)	12	14.1
(c)	Python	13	233.
(d)	C++	28	6.2
(e)	C++ AMP	81	7.1

The five target implementations developed were: ESDL for `esec` (Python), ESDL for C++ AMP, hand-coded Python, hand-coded C++ (using the Standard Template Library (STL)) and hand-coded C++ AMP.[12] Relative code complexity and execution times per generation are shown in Table II. ESDL for `esec` is used as a baseline for complexity with a value of one, and while ESDL for C++ AMP uses an almost identical system definition, the implementation of the mutation operator adds significant complexity. Developing from scratch for Python requires similar effort to ESDL for C++ AMP, though even with performance tuning the execution is an order-of-magnitude slower a natively compiled language. C++ has the benefit of a mature optimising compiler and extensive use of the STL reduces the effort required (or rather, transfers the effort into learning the STL). Finally, developing for C++ AMP requires an understanding of algorithm parallelisation, new libraries and the underlying hardware platform, as well as generally more complex code. All implementations were profiled and significant bottlenecks were removed to avoid unfair comparisons, though thorough tuning was not performed.

The hardware used for testing was an NVIDIA GeForce GT 540M with 96 cores and 4GB available memory (half dedicated and half system) and an Intel i7-2720QM CPU with eight logical cores (of which only one was ever used) running at 2.20GHz, 8GB RAM. CPython 2.7.2 64-bit was used for the Python implementations, while Microsoft Visual C++ 11 Beta was used for the C++ implementations. All experiments were run on Windows 7 64-bit. Performance measurements were taken using Python's `time.clock()` function and the

[10] Variadic templates could also be used for this. However, the version of the C++ compiler used does not support variadics.

[11] Most common complexity measures are not comparable between different languages and paradigms. In particular, McCabe's Cyclomatic Complexity [23] is not relevant since the algorithm is the same in each case, while coupling and cohesion measures are only relevant for object-oriented software designs.

[12] All source code is freely available from the author's website (http://stevedower.id.au/esdl/amp/), though some other dependencies are required in order to compile and use the code.

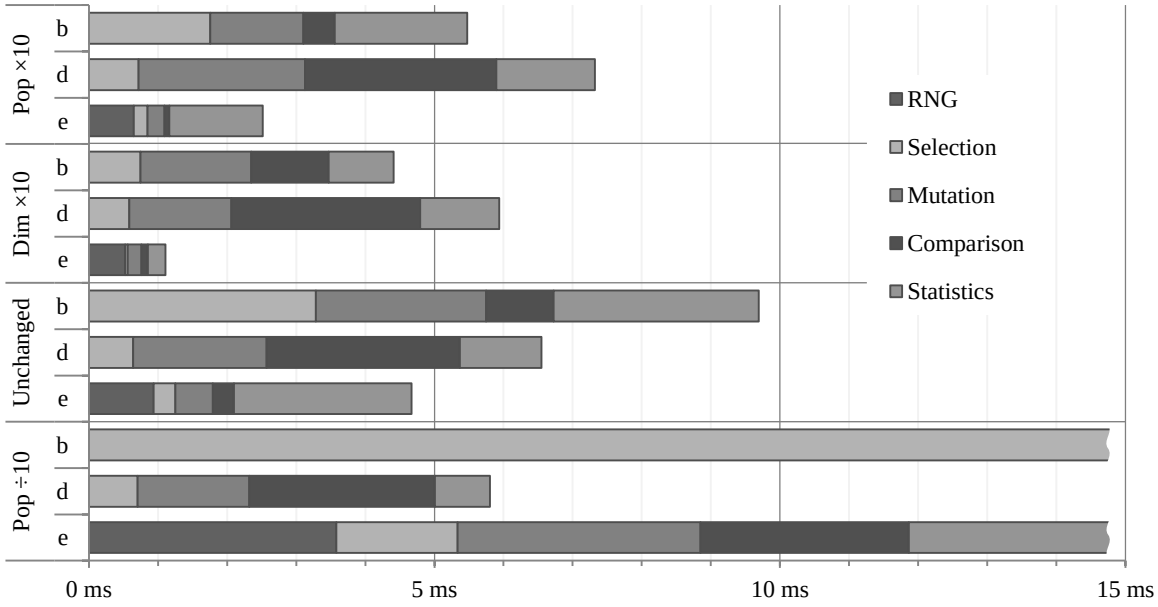

Fig. 10. Normalised comparison of (b) the transformed ESDL, (d) the C++ and (e) the C++ AMP implementations.

Concurrency Visualizer included with Visual Studio 11. The speed measurements in Table II are the mean timings over one hundred iterations. (Each algorithm was run for 105 iterations; excluding the first five allows for initialisation and lazy-loading [24].) Measurement instrumentation has a noticeable effect (approximately 10 %) on the performance, largely due to interference with C++ AMP's deferred execution model. This has not been compensated for in the reported timings.

As expected, the performance benefit achieved increased with implementation difficulty. By design, `esec` is a very flexible framework, which reduces the implementation difficulty but adds a number of overheads; function-level profiling reveals no easily-fixed bottleneck. Using any form of natively compiled language provided an order-of-magnitude performance increase. While further execution improvement is possible by implementing a monolithic algorithm (Table IIc) rather than composing discrete operators, the added effort becomes harder to justify. Further, the loss of flexibility and maintainability is a significant restriction on research applications, which are likely to involve fewer runs of a greater variety of algorithms than commercial applications where the algorithm is not being modified and timeliness is more important.

Moving from a single-threaded C++ implementation (Table IId) to the many-threaded C++ AMP implementations (Table IIb and e) showed little performance variation, despite the size of the DE population favouring the parallel implementations. To observe the sensitivity of each C++ implementation with regards to the amount of data, measurements were taken at the original settings (population of 10 000, ten dimensions), with a population one-tenth of that, a population ten times larger and with the problem dimension increased to 100. Figure 10 shows the mean time taken by each step in the three C++ implementations, normalised to match the original settings. In effect, the times for population $\times 10$ (100 000) and dimension $\times 10$ (100) are shown at one-tenth actual length and population $\div 10$ (1000) at ten times longer. An implementation that displays linear performance with the varied parameter will have similar normalised timings; shorter times indicate faster per-size execution, rather than less actual execution time.

Figure 10 shows that the performance improves most dramatically for the GPU implementations ((b) and (e); the truncated lines extend so far as to render the other values unreadable) as population size or dimension increases. The processing time for the CPU implementation (d) increases faster than population size while the others see a sub-linear increase. With a ten-dimensional problem, neither GPU implementation outperforms the CPU implementation until a population larger than ten thousand is used. Even with much smaller populations, the overwhelming number of individuals prevents DE from performing at its best, such that the performance increase due to using the GPU does not provide any actual benefit. Increasing the dimension has a more dramatic effect, suggesting that population size should not be used as the sole heuristic for data-parallel performance.

Despite the population size negating the parallelism benefits in terms of algorithm results, there are a number of useful observations from the results in Figure 10. Firstly, the evaluation timing (included as part of "Comparison") increases linearly with population size on the CPU, while being significantly sublinear on the GPU. The Rastrigin function includes a number of cosine evaluations, which are notoriously expensive on the CPU. However, this is effectively a constant factor between implementations, and linearity with population size and dimensionality validates the prior assumptions that evaluation is inherently parallelisable. Similarly, sorting the population (included in "Statistics" and also "Selection" for (b); the other implementations optimised by reusing a previous sort result) does not scale well on the CPU as population size increases.

With a well-tuned GPU sort implementation (unlike that used here) and greater optimisation in the code generator (to remove the redundant sort), better performance can be achieved in this area through updated tooling, which provides the benefit to users with no effort on their part. Sort times are unaffected by individual size except for overheads; the execution time of sort algorithms that do not require copying are constant against dimensionality.

More interesting than comparing the performance of three efficient implementations is the relative code complexity. We have already identified that, with little effort, compiling ESDL code to C++ shows an order-of-magnitude performance improvement. Currently, creating an optimised C++ implementation of an algorithm based on a textual description or existing implementation is a significant undertaking. Consider, however, the case where the publication includes an ESDL definition and a clear specification of each extra operator. An automatic conversion can produce code that requires significantly less development, debugging and validation effort, can benefit from future improvements to the tools and libraries, can act as an easily read description of the algorithm and provide execution performance comparable to hand-crafted implementations.

V. Summary

This paper has presented an approach for transforming ESDL systems into code that is executable on heterogeneous computing platforms using C++ AMP. The implementation is a proof-of-concept and does not produce the peak performance that can be achieved by data-parallel processing. However, this work has shown that the ability to take advantage of some of the available processing power does not need to be restricted to expert developers and that future improvements can be made to the transformation without requiring changes to user's existing code (specifically, their ESDL system definitions).

It is widely recognised that there is much work still required in effectively implementing evolutionary algorithms on data-parallel computing platforms such as GPUs. In particular, parallel random number generation and sort algorithms are yet to mature to a state comparable to those for CPUs. Further, GPUs do not provide any performance benefits without a sufficient amount of data, regardless of whether that is obtained by increasing the count or the dimension of solutions used, with the exception that an expensive and parallelisable evaluation function may be usefully accelerated independently from the evolutionary algorithm when data transfer overheads are not significant.

ESDL has been shown to be a useful model for describing evolutionary algorithms in a platform independent manner. The ability to compile and use an ESDL system on high-performance desktop hardware makes heterogeneous computing platform accessible to practitioners without requiring specific expertise in GPU development.

References

[1] H. Sutter, "The free lunch is over," *Dr. Dobb's Journal*, vol. 30, March 2005.

[2] ——, "Welcome to the jungle," 2011, http://herbsutter.com/welcome-to-the-jungle/.

[3] K. Gregory, "Overview and C++ AMP approach," September 2011, http://www.gregcons.com/CppAmp/OverviewAndCppAMPApproach.pdf.

[4] D. Dagum, "Introducing C++ accelerated massive parallelism (C++ AMP)," June 2011, http://blogs.msdn.com/b/vcblog/archive/2011/06/15/introducing-amp.aspx.

[5] NVIDIA Corp., "Parallel programming and computing platform," 2012, http://nvidia.com/cuda.

[6] Microsoft Corp., "C++ AMP : Language and programming model," January 2012, http://blogs.msdn.com/b/nativeconcurrency/archive/2012/02/03/c-amp-open-spec-published.aspx.

[7] ISO/IEC 14882:2011, "Information technology – programming languages – C++," 2011.

[8] S. Dower, "ESDL and an abstraction for evolutionary algorithms," Ph.D. dissertation, Swinburne University of Technology, unpublished.

[9] S. Dower and C. J. Woodward, "ESDL: a simple description language for population-based evolutionary computation," in *Proc. 13th Annu. Conf. Genetic and Evolutionary Computation*. Dublin, Ireland: ACM, 2011, pp. 1045–1052.

[10] ——, "Evolutionary system definition language," Swinburne University of Technology, Tech. Rep. TR/CIS/2010/1, 2010.

[11] S. Harding and W. Banzhaf, "Fast genetic programming on GPUs," in *Genetic Programming*. Springer, 2007, pp. 90–101.

[12] O. Maitre *et al.*, "Coarse grain parallelization of evolutionary algorithms on GPGPU cards with EASEA," in *Proc. 11th Annu. Conf. on Genetic and evolutionary Computation*. ACM, New York, United States, 2009, pp. 1403–1410.

[13] P. Krömer *et al.*, "Many-threaded implementation of differential evolution for the CUDA platform," in *Proc. 13th Annu. Conf. Genetic and Evolutionary Computation*. Dublin, Ireland: ACM, 2011, pp. 1595–1602.

[14] S. Debattisti *et al.*, "Implementation of a simple genetic algorithm within the CUDA architecture," in *11th Annu. Conf. Companion Genetic and Evolutionary Computation*. ACM, 2009.

[15] E. Cantú-Paz, "A survey of parallel genetic algorithms," *Calculateurs paralleles, reseaux et systems repartis*, pp. 141–171, 1998.

[16] P. Pospichal *et al.*, "Parallel genetic algorithm on the CUDA architecture," in *Applications of Evolutionary Computation*. Springer, 2010, pp. 442–451.

[17] S. Dower, "Bitonic sort for C++ AMP," Swinburne University of Technology, Tech. Rep. TR/CIS/2012/1, 2012.

[18] K. E. Batcher, "Sorting networks and their applications," in *Proc. April 30–May 2, 1968, spring joint computer conf.* ACM, 1968, pp. 307–314.

[19] G. Bilardi and A. Nicolau, "Adaptive bitonic sorting: An optimal parallel algorithm for shared memory machines," Cornell University, Tech. Rep., 1986.

[20] L. Howes and D. Thomas, "Efficient random number generation and application using CUDA," in *GPU Gems 3*. Addison Wesley, 2007.

[21] K. V. Price *et al.*, *Differential Evolution*. Springer, 2005.

[22] M. H. Halstead, *Elements of Software Science*. Elsevier, 1977.

[23] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. 2, pp. 308–320, December 1976.

[24] S. Wybranski, "How to measure the performance of C++ AMP algorithms?" December 2011, http://blogs.msdn.com/b/nativeconcurrency/archive/2011/12/28/how-to-measure-the-performance-of-c-amp-algorithms.aspx.