Revisiting SIMD programming

Anton Lokhmotov[1]*, Benedict R. Gaster[2],
Alan Mycroft[1], Neil Hickey[2], and David Stuttard[2]

[1] Computer Laboratory, University of Cambridge
15 JJ Thomson Avenue, Cambridge, CB3 0FD, UK
[2] ClearSpeed Technology
3110 Great Western Court, Bristol, BS34 8HP, UK

Abstract. Massively parallel SIMD array architectures are making their way into embedded processors. In these architectures, a number of identical processing elements having small private storage and using asynchronous I/O for accessing large shared memory executes the same instruction in lockstep.
In this paper, we outline a simple extension to the C language, called C^n, used for programming a commercial SIMD array architecture. The design of C^n is based on the concept of the SIMD array type architecture and revisits first principles of designing efficient and portable parallel programming languages. C^n has a low level of abstraction and can also be seen as an intermediate language in the compilation from higher level parallel languages to machine code.

1 Introduction

Massively parallel SIMD array architectures are no longer merely the province of large supercomputer systems but are making their way into embedded processors. What seemed a cautious extrapolation a decade ago [1] has now become a reality.

In the 40 years since the design of Illiac IV [2], the first large-scale array computer consisting of 64 processing elements (PEs), the progress of VLSI technology allows to pack even more PEs into a single-chip microprocessor that in most respects can be considered "embedded". For example, ClearSpeed's CSX600 array processor consisting of 96 PEs has excellent performance per unit power by delivering more than 3 GFLOPS per watt [3].

The CSX is a SIMD array architecture with a control unit and a number of processing elements, with each PE having relatively small private storage and using asynchronous I/O mechanisms to access large shared memory.

In this paper, we describe a data-parallel extension to the C language, called C^n, for programming the CSX architecture. In C^n, parallelism is mainly expressed at the type level rather than at the code level. Essentially, C^n introduces a new multiplicity type qualifier **`poly`** which implies that each PE has its own copy of a value. For example, the definition **`poly int`** `X;` implies that, on the CSX600 with 96 PEs, there exist 96 copies of integer variable `X`, each having the same address within its PE's local storage. The

* This author gratefully acknowledges the financial support by a TNK-BP Cambridge Kapitza Scholarship and an Overseas Research Students Award.

multiplicity is also manifested in conditional statements. For example, the following code alters the value of `X` on every even PE (the runtime function `get_penum()` returns the ordinal number of a PE):

```
if(get_penum()%2 == 0) X = 0;
```

On every odd PE, the assignment is not executed (this is equivalent to issuing a NOP instruction, as the SIMD array operates in lock-step).

We describe designing C^n as an efficient *and* portable language. Efficiency and portability often conflict with each other (especially in parallel languages [4]). We argue that the CSX architecture (§2) is representative of its class and thus can be considered as a *SIMD array type architecture* (§3). Core C^n operations (§4) can be thought of as cheap (in the spirit of C), while more expensive operations are relegated to the standard library. We compare and contrast C^n design decisions with similar approaches (§5), and then argue that C^n can be seen as an *intermediate* language in the compilation from higher level parallel languages (§6). We briefly discuss the C^n compiler implementation (§7) and outline future work in conclusion (§8).

2 CSX architecture

This section outlines ClearSpeed's CSX architecture [3], which largely honours the classical SIMD array organisation pioneered by the Solomon/Illiac IV designs [5], albeit embodied in a single chip.

2.1 CSX family

The CSX architecture is a family of processors based on ClearSpeed's multi-threaded array processor (MTAP) core. The architecture has been developed for high performance, high rate processing. CSX processors can be used as application accelerators, alongside general-purpose processors such as those from Intel and AMD.

The MTAP consists of execution units and a control unit. One part of the processor forms the *mono* execution unit, dedicated to processing scalar (or mono) data. Another part forms the *poly* execution unit, which processes parallel (or poly) data, and may consist of tens, hundreds or even thousands of identical processing element (PE) cores. This array of PE cores operates in a synchronous, Single Instruction Multiple Data (SIMD) manner, where every enabled PE core executes the same instruction on its piece of data.

The control unit fetches instructions from a *single* instruction stream, decodes and dispatches them to the execution units or I/O controllers. Instructions for the mono and poly execution units are handled similarly, except for conditional execution. The mono unit uses conditional jumps to branch around code like a standard RISC architecture. This affects both mono and poly operations. The poly unit uses an *enable register* to control execution of each PE. If one or more of the bits of that PE enable register is zero, then the PE core is *disabled* and most instructions it receives will be ignored. The enable register is a stack, and a new bit, specifying the result of a test, can be pushed onto the top of the stack allowing nested predicated execution. The bit can later be

popped from the top of the stack to remove the effect of that condition. This makes handling nested conditions and loops efficient.

In order to provide fast access to the data being processed, each PE core has its own local memory and register file. Each PE core can directly access only its own storage. (Instructions for the poly execution unit having a mono register operand indirectly access the mono register file, as a mono value gets broadcast to each PE.) Data is transferred between PE (poly) memory and the poly register file via load/store instructions. The mono unit has direct access to main (mono) memory. It also uses load/store instructions to transfer data between mono memory and mono register file. Programmed I/O (PIO) extends the load/store model: it is used for transfers of data between mono memory and poly memory.

2.2 CSX600 processor

The CSX600 is the first product in the CSX family. The processor is optimised for intensive double-precision floating-point computations, providing sustained 33 GFLOPS of performance, while dissipating an average of 10 watts. The poly execution unit is a linear array of 96 PE cores, with 6KB SRAM and a superscalar 64-bit FPU on each PE core. The PE cores are able to communicate with each other via what is known as *swazzle path* that connects each PE with its left and right neighbours. Further details can be found in white papers [3].

2.3 Acceleration example

A C^n implementation of Monte-Carlo simulations for computing European option pricing with double precision performs 100,000 Monte-Carlo simulations at the rate of 206.5M samples per second on a ClearSpeed Advance board. In comparison, an optimised C program using the Intel Math Kernel Library (MKL) and compiled with the Intel C Compiler achieves the rate of 40.5M samples per second on a 2.33GHz dual-core Intel Xeon (Woodcrest, HP DL380 G5 system). Combining both the Intel processor and the ClearSpeed board achieves the rate of 240M samples per second, which is almost 6 times the performance of the host processor alone.

3 C^n design goals and choices

The key design goals of C^n were efficiency and portability. We first discuss these goals in a broader context of programming languages for sequential and parallel computers (§3.1), and then discuss how they affected the design choices of C^n (§3.2).

3.1 Efficiency and portability

Program efficiency and portability are two common tenets of high-level programming languages. Efficiency means that it is possible to write a compiler generating code that is “almost” as fast as code written in assembly. Portability means that software can be adapted to run on different target systems.

Languages like C have been successful largely because of their efficiency and portability on von Neumann machines, having a single processor and uniform random access memory. Efficiency comes from the programmer's clear understanding of the performance implications of algorithm selection and coding style. Portability is achieved because languages like C hide most features of physical machines, such as instruction set, addressing modes, register file, *etc.* Moreover, the hidden features apparently have a negligible effect on performance, so porting often maintains efficiency [4].

Efficiency and portability are even more desired when programming parallel systems. Performance is the most compelling argument for parallel computing; and given the amount of human effort required to develop an efficient parallel program, the resulting program should better have a long useful life [6].

In the world of parallelism, however, no single model accurately abstracts the variability of parallel systems. While it is possible, for example, to program distributed memory machines using a shared memory model, programmers having limited control over data distribution and communication are unlikely to write efficient programs. So in this world, unfortunately, efficiency and portability are no longer close friends.

Snyder introduced [7] the notion of a *type architecture*—a machine model abstracting the performance-important features of a family of physical machines, in the same way as the RAM model abstracts von Neumann machines. He proposed the Candidate Type Architecture (CTA) which effectively abstracts MIMD machines (multicomputers) with unspecified interconnect topology. The key CTA abstraction is that accessing another processor's memory is significantly more expensive than accessing local memory (typically by 2–5 orders of magnitude) [4].

3.2 C^n as a language for the SIMD array type architecture

Our key observation is that the CSX architecture can be considered a *SIMD array type architecture* (SATA), as it respects classical organisation and has typical costs of communicating data between main (mono) and local (poly) memory. Designing a core language based on the type-architecture facilities should provide both efficiency and portability [7]. To match the programmer's intuition, core language operations are cheap, while operations relegated to the standard library are more expensive. For example, arithmetic operations are cheap, while reduction operations (using the inter-PE communication) are somewhat more expensive, although still cheaper than data transfer operations between mono and poly memories.

Efficiency mandates only providing language constructs that can be reliably implemented on the SATA using standard compiler technology. History shows that languages that are difficult to implement are also rarely successful (HPF is a dismal example [8]).

Portability is important because the CSX architecture family (§2.1) does not fix the PE array size. Also, as the number of PE cores increases, other (than linear) interconnect topologies could be introduced.

Designing C^n as a C extension provides a known starting point for a large community of C programmers. In addition, software development tools can be written by making (small) modifications to existing ones, rather than from scratch.

4 C^n outline

In this section we outline the most interesting features of C^n. We give further rationale behind some design decisions in §5.

4.1 Types

C^n extends C with two additional keywords that can be used as part of the declaration qualifier for declarators, *i.e.* logically amend the type system [9]. These keywords are *multiplicity qualifiers* and allow the programmer to specify a memory space in which a declared object will reside. The new keywords are:

- **`mono`**: for declaring an object in the mono domain (*i.e.* only one copy will exist in main memory);
- **`poly`**: for declaring an object in the poly domain (*i.e.* one copy per PE in its local memory).

The default multiplicity is **`mono`**. Wherever a multiplicity qualifier may be used, an implicit **`mono`** is assumed, unless an explicit **`poly`** is provided. A consequence of this is that all C programs are valid C^n programs, with the same semantics. Thus, C^n is a superset of C (but see §5).

Basic types C^n supports the same basic types as C. They can be used together with a multiplicity qualifier to produce declarations (only one of the qualifiers can be used in a basic type declaration). Some example declarations follow:

```
poly int i; // multiple copies in PE local (poly) memory
mono unsigned cu; // a single copy in main (mono) memory
unsigned long cs; // a single copy in main (mono) memory
```

Pointer types The pointer types in C^n follow similar rules to those in C. Pointer declarations consist of a base type and a pointer. The declaration on the left of the asterisk represents the base type (the type of the object that the pointer points to). The declaration on the right of the asterisk represents the pointer object itself. It is possible to specify the multiplicity of either of these entities in the same way as **`const`** and **`volatile`** work in C. For example:

```
poly int * poly sam; // poly pointer to poly int
poly int * frodo;    // mono pointer to poly int
int * poly bilbo;    // poly pointer to mono int
```

Thus, there are four different ways of declaring pointers with multiplicity qualifiers:

- mono pointer to mono object (*e.g.* **`mono int * mono`**);
- mono pointer to poly object (*e.g.* **`poly int * mono`**);
- poly pointer to mono data (*e.g.* **`mono int * poly`**);
- poly pointer to poly data (*e.g.* **`poly int * poly`**).

Note that in the case of declaring a poly pointer, there exist multiple copies of the pointer, potentially pointing to different locations.

As in C, pointers are used to access memory. The compiler allocates named poly objects to the same address within each PE local memory. Thus, taking the address of a named object (whether mono or poly) always yields a mono pointer.

Array types The syntax for array declaration in C^n is similar to that in C. It is possible to use a multiplicity qualifier in an array declaration. Consider the declaration `poly int A[42];`. The multiplicity qualifier applies only to the base type of the array (*i.e.* to the type of array elements). This declaration will reserve a poly space for 42 integers at the same location on each PE. Similar to C, we can say that the array name is coerced to the address of its first element, which is a mono pointer to poly object (*e.g.* `poly int * mono`).

There are some additional restrictions when dealing with arrays, specifically with array subscripting, discussed in §4.3.

Aggregate types Similar to C [9], C^n distinguishes between declaration and definition of objects. A declaration is where an object type is specified but no object of that type is created, *e.g.* a struct type declaration. A definition is where an object of a particular type is created, *e.g.* a variable definition. Structs and unions in C^n match their C equivalents, the only difference being the type of fields one can specify inside a struct type declaration.

Standard C allows essentially any declaration as a field, with the exception of storage class qualifiers. C^n allows the same with the additional restriction that fields cannot have multiplicity qualifiers. This is because a struct type declaration just specifies the structure of memory. Memory is not allocated until an instance of that struct type is created. Thus, putting a poly field in a struct declaration and then defining an instance of that struct in the mono domain would result in having contradictory multiplicity specifications. (Similarly for a mono field in a struct instance defined in the poly domain.) For example:

```
// legal struct declaration
struct _A { int a; float b; }; poly struct _A kaiser;

// illegal struct declaration
struct _B {
    poly int a;   // illegal use of multiplicity
    mono float b; // illegal use of multiplicity
};
mono struct _B king; // where should king.a go?
poly struct _B tsar; // where should tsar.b go?
```

Multiplicity qualifiers, however, can be used on the base type of pointers (otherwise pointers to poly data could not be declared as fields). For example:

```
union _C { // define a union
    poly int * a;
    mono int * poly * b;
```

```
};
poly union _C fred;
mono union _C barney;
```

In the declaration of `fred`, the field `a` is created as a poly pointer to a poly int (**`poly int * poly`**) and `b` is created as a poly pointer to a poly pointer to a mono int (**`mono int * poly * poly`**). In the declaration of `barney`, `a` is created as a mono pointer to a poly int (**`poly int * mono`**) and `b` is created as a mono pointer to a poly pointer to a mono int (**`mono int * poly * mono`**).

4.2 Expressions

C^n supports all the operators of C. Mono and poly objects can usually be used interchangeably in expressions (but see §4.3 for exceptions).

Note that the result of any expression involving a mono and a poly object invariably has a poly type. Thus, mono objects are promoted to the poly domain in mixed expressions. In the following example,

```
poly int x; int y; x = x + y;
```

where `y` is promoted to **`poly int`** before being added to (every copy of) `x`. (Technically, the promotion is done by broadcasting values from the mono register file to the poly execution unit.)

4.3 Assignment statements

Assignment within a domain (*i.e.* poly to poly, or mono to mono) is always legal and has the obvious behaviour.

A mono value can also be assigned to a poly variable. In this case the same value is assigned to every copy of the variable on each PE. For example, **`poly int`** `x = 1;` results in (every copy of) `x` having the value of 1. (Again, the architecture supports such assignments by broadcasting the mono value to the poly unit.)

It is not obvious, however, what should happen when assigning a poly to a mono, *i.e.* when taking data from multiple copies of the poly variable and storing it in the single mono variable. Therefore direct assignment from the poly domain to the mono domain is disallowed in C^n.

Note that certain C^n expressions are disallowed, as otherwise they would require an implicit data transfer from mono to poly memory. One such expression is dereferencing of a poly pointer to mono data (*e.g.* **`mono int * poly`**). Attempting to dereference such a pointer would result in poly data, but the data is stored in the mono domain and would therefore need to be copied to poly memory. Since broadcasting can only send a single mono value to the poly unit at a time, such a copy would involve expensive programmed I/O mechanisms. Therefore, allowing such dereferencing would conflict with the design goal that the core language operations should be cheap. Thus, the compiler reports dereferencing a poly pointer to mono data as an error.

Since, following C, C^n treats `x[i]` as equivalent to `*(x + i)`, it follows that indexing a mono array with a poly expression is also forbidden, because it would implicitly dereference a poly pointer to mono data.

In all the cases when data transfers between mono and poly memories are required, the programmer has to use memory copy routines from the standard C^n library.

4.4 Reduction operations

Many parallel algorithms require reducing a vector of values to a single scalar value. In addition to the core operations defined above, the C^n standard library provides `sum`, `times`, and bit-wise operations defined for basic types for reducing a poly value into a mono value. For example, on an array of n PEs,

```
poly float x; mono float y; ...
y = reduce_mono_sum(x);
```

means

```
y = x(0) + x(1) + ... + x(n-1);
```

where `x(i)` refers to the value of `x` on `i`th PE.

For some algorithms another form of reduction is useful: logically, a poly value is reduced to an intermediate mono value which is then broadcast into a result poly value. Thus,

```
poly float x, z; ...
z = reduce_poly_sum(x);
```

means

```
z(0) = ... = z(n-1) = x(0) + x(1) + ... + x(n-1);
```

Both forms can be efficiently implemented on the CSX using the inter-PE swazzle path. The order in which the result is evaluated is unspecified.

4.5 Control statements

The basic control flow constructs in C^n are the same as in C. Conditional expressions, however, can be of mono or poly domain. Consider the **`if`** statement:

```
if(expression) { statement-list }
```

A mono expression for the condition affects both the mono and poly execution units. If the expression is false, the statement list will be skipped entirely, and execution will continue after the **`if`** statement.

A poly expression for the condition can be true on some PEs and false on others. This is where the PE enable state (described in §2.1) comes in: all the PEs for which the condition is false are disabled for the duration of executing the statement list. The statement list is executed (even if all PEs are disabled by the condition), but any poly statements (*e.g.* assignments to poly variables) have an affect only on the enabled PEs. Mono statements inside a poly **`if`**, however, get executed irrespective of the conditions. Consider the following example:

```
poly int foo = 0; mono int bar = 1;
if(get_penum()%2 == 0) { // disable all odd PEs
  foo = 1; // foo is 1 on even and 0 on odd PEs
  bar = 2; // bar is 2
}
```

Effectively, a poly condition is invisible to any mono code inside that **`if`** statement.[3] This language design choice may seem counterintuitive and is indeed a controversial point in the design of SIMD languages, to which we return in §5.2.

A poly condition in an **`if..else`** statement implies that for the **`if`**-clause all the PEs on which the condition evaluates to true are enabled, and the others are disabled. Then, for the **`else`**-clause, the enable state of all the PEs is inverted: those PEs that were enabled by the condition are disabled and vice-versa.

Conditional statements can be nested just as in C. Poly statements are only executed on PEs when all the nested conditional expressions evaluated to true.

These rules, of course, dictate different compilation of conditional statements. Mono conditional statements result in generating branch instructions, while poly conditional statements result in generating poly instructions enabling and disabling PEs (enable stack operations on the CSX; see §7 for more details).

Similar principles apply to loop constructs **`for`**, **`while`** and **`do..while`**. A loop with a poly control expression executes until the loop condition is false on every PE. Note that a poly loop can iterate zero times, so in that case, unlike the **`if`** statement, even mono statements in its body will not be executed.

4.6 Functions

Multiplicity qualifiers can be specified for the return type of a function, as well as the types of any arguments. We refer to a function as being mono or poly according to whether its return type is mono or poly.

A return statement from a mono function behaves exactly as expected by transferring control to the point after the call statement. A poly function does not actually return control until the end of the function body. A return from a poly function works by disabling the PEs which execute it. Other PEs execute the rest of the function code. When the function returns, all PEs have their enable state restored to what it was on entry. Note that all mono code in the function is always executed (unless branched over by mono conditional statements).

5 C^n design rationale

5.1 Low-level abstraction

Early SIMD programming languages for the Illiac IV computer included Glypnir [10], with syntax based on Algol 60, and CFD [11], based on Fortran. The main design goal of these languages was "to produce a useful, reliable, and efficient programming tool with a high probability of success" [10]. Given the state of compiler technology in the early 1970s, the languages could not *both* be machine independent *and* satisfy this goal.[4] In addition to explicit means for specifying storage allocation and program

[3] It way be worth noting that the block structure of code is still relevant. So, for example, any mono declarations within the body of a poly conditional statement are local to that body.

[4] The compiler for Tranquil—the first Illiac IV language—was abandoned because of implementation difficulties and lack of resources [10].

control as in C^n, these languages even provided means for accessing subwords and registers.

Many vector processing languages have appeared since then, providing higher levels of abstraction for array-based operations (*e.g.* Actus [12], Vector C [13], High Performance Fortran [8], *etc.*). Unfortunately, such languages present greater complexity for the implementors because of data alignment, storage allocation and communication issues on SIMD machines (*e.g.* see [14, 15]).

C^n builds on the C strength of "solving" most implementation efficiency problems by leaving them to the programmers. While not the most desirable solution, it has relieved the programmers from solving the same problems using assembly language. In §6 we argue that C^n can be seen as an intermediate representation for compiling from higher level parallel languages.

5.2 C^n and other SIMD dialects of C

C^n is by no means the first C dialect for SIMD programming. Notable examples include C* [16] for programming the Connection Machine models, MPL [17] for the MasPar models, and 1DC (one-dimensional C) [18] for the IMAP models.

All these languages intentionally reflect their respective architectures. The unifying theme is the use of a keyword to specify multiple instance variables (**`poly`** in C*, **`plural`** in MPL, **`sep`** in 1DC). The language differences stem from the differences between the architectures and design goals.

Communication While the **`poly`** keyword implies the physical data distribution, C* enshrines the viewpoint that no parallel version of C can abandon the uniform address space model without giving up its claim to be a superset of C. Uniform memory means that a PE can have a pointer `p` into the memory of another PE. For example, the statement `*p = x;` means "send message `x` to a location pointed to by `p`". Thus, C* relies on pointers for interprocessor communication.

This C* feature is underpinned by Connection Machine's key capability for any PE to establish a connection with any other PE in the machine (via the global routing mechanism or local meshes). Still, the programmer relies on the compiler writer's ability to implement pointer-based communication efficiently. Perhaps, this is the reason why, in contrast, MasPar's MPL provides explicit communication operators, although the MasPar has connection capabilities similar to the Connection Machine.

The IMAP architectures have a linear nearest neighbour interconnect, as does the CSX. 1DC supports special operators for nearest neighbour communication, while C^n relegates communication functions to the C^n standard library.

Dereferencing pointers Even rejecting the pointer-based communication in C* does not mean that the compiler will be able to optimise poorly written code; in particular, code that makes a heavy use of costly DMA transfers between mono and poly memories. The C^n ban on dereferencing poly pointers to mono data (which would transfer only several bytes at a time) aims to shut the door in front of the abusing programmers. In contrast, MPL does not sacrifice convenience for efficiency and allows all pointer operations.

Poly conditional statements In §4.5, we discussed the behaviour of poly **`if`** statements in C^n: even if the poly condition is false on every PE, the statement list is executed regardless, including all mono statements in the list. This model of execution is easy to implement on the CSX by inserting operations on the hardware enable stack (see §7). The same model was used in MPL [19].

The designers of C* followed the other route by adopting the "Rule of Local Support" [16, §6.2]: if the poly condition is false on every PE, then the statement list is not executed at all. The Rule of Local Support required extra implementation trouble but preserved the usual model of a **`while`** loop in terms of **`if`** and **`goto`** statements. The C* designers, nevertheless, admitted that their rule occasionally also caused inconvenience and confusion.

Deciding on whether to preserve or to change the semantics of familiar statements when extending a sequential language for parallelism is hard, and may even drive the designers to a thought that designing a language from scratch is a better idea (*e.g.* see the history of ZPL [4]). In the case of **`if`** statements, the solution does not need to be that radical and could merely come as using different keywords in a more general language supporting both execution models (for example, **`ifp`** for the C^n/MPL model and **`ifm`** for the C* model).

6 C^n as intermediate language

SIMD array supercomputers went out of fashion in the early 1990s when clusters of commodity computers proved to be more cost effective. Ironically, vector-style instructions reemerged in commodity processors under the name of SIMD (or multimedia) extensions, such as Intel MMX/SSE and PowerPC AltiVec.

The principal difference between "real" (array) SIMD and vector extensions is that the latter operate on data in long registers and do not have local memory attached to each PE as in SIMD arrays. In terms of the C^n language, this means that it would be inefficient (if not impossible) to use pointers to poly data when programming in C^n for vector extensions. This is because it would not be straightforward to compile C^n programs expressing memory operations that are not supported by the hardware.

We argue, however, that the C^n language can be regarded as a "portable assembly" language for both SIMD and vector architectures. Given an architecture description, high-level languages providing more abstract array-based operations can be translated to C^n and then efficiently mapped to machine code by the C^n compiler.

For example, consider a Fortran 90 style statement `B[0:959] = A[0:959] + 42;` where `A[]` and `B[]` are arrays of floats. This statement loads 960 values from `A`, adds 42 to each, and stores 960 results to `B`. Suppose that the target architecture is a vector machine with 96 elements per vector register, hence the statement parallelism needs to be folded 10 times to fit the hardware by *strip mining* [20, 15].

For a vector machine, the declaration **`poly float`** `va, vb;` can be thought of as defining two vector registers in a similar way as **`vector float`** `vc;` can be used to define a (4-element) vector in the AltiVec API [21]. The vector statement can then be strip-mined *horizontally* [15], resulting in:

```
const poly int me = get_penum(); // get PE number
```

```
mono float * poly pa = A + me;
mono float * poly pb = B + me;
for(int i = 0; i < 10; i++, pa += 96, pb += 96) {
  va  = *pa; // load 96-element vector
  vb  = va + 42;
  *pb = vb; // store 96-element vector
}
```

(Note that here we have lifted the C^n ban on dereferencing poly pointers to mono data, which makes sense on distributed memory SIMD arrays.)

The same strategy works on a SIMD machine having 96 PEs: 10 vector indirect loads (one on each iteration) are replaced with 10 DMA transfers from mono to poly memory, 10 vector stores with 10 DMA transfers from poly to mono memory; each transfer moves a single float to/from a PE. It is more efficient, however, to strip-mine *vertically* [15], which in pseudo-vector notation can be written as:

```
poly float pA[10], pB[10];
pA[0:9] = A[10*me:10*me+9]; // DMA mono to poly
pB[0:9] = pA[0:9] + 42;
B[10*me:10*me+9] = pB[0:9]; // DMA poly to mono
```

This requires only two DMA transfers of 10 floats each. Given high DMA start-up costs, vertical strip-mining is several times more efficient. Hence, the compiler should favour the second form on SIMD machines. (This is a trivial example of the decisions we referred to in §5.1 that the compiler needs to make to efficiently compile high-level vector abstractions).

To summarise, C^n code can be seen as an intermediate representation, from which target code can be generated. Once a C^n compiler is written and well-tuned, the quality of target code should depend on the ability of the front-end (using an architecture description to translate a higher level parallel language to C^n) to produce efficient C^n code.

C^n has already been targeted from a subset of OpenMP [22]. We believe that C^n can also be targeted from high-level data parallel libraries such as MSR Accelerator [23].

7 C^n compiler implementation

ClearSpeed has developed a C^n optimising compiler for the CSX architecture using the CoSy compiler development framework [24]. Small modifications to the front-end were needed to support the multiplicity qualifiers in the source code. Each type is supplemented with a flag indicating whether it is mono or poly. A special phase was written to recognise poly conditional statements and transform them into a form of predicated execution. For example, assuming that both `x` and `y` are in the poly domain,

```
if (y > 0) {
  x = y;
}
```

becomes

```
enablestate = push(y > 0, enablestate);
x ?= y;
enablestate = pop(enablestate);
```

where the predicated assignment operator `?=` assigns its *rhs* expression to the *lhs* location only on those PEs where all the bits of the hardware enable stack (represented by the variable `enablestate`) are 1.

Predicated execution requires careful rethinking of standard optimisations. For example, the standard register allocation algorithm via coloring has to recognise that liveness of a virtual poly register is not simply implied by its def-use chain but is also a function of the enable state. This is not, however, a new problem (*e.g.* see [25]).

7.1 C^n compiler performance

ClearSpeed has developed a number of interesting applications in C^n, including functions from the molecular dynamics simulation package AMBER, Monte-Carlo option pricing simulations, implementations of 2D and 3D FFTs, and others.

The C^n design choice of only including features that can be efficiently implemented using standard compiler technology pays off handsomely, since the implementors can concentrate their efforts on optimisations that the programmer expects the compiler to get right. For example, Fig. 1 shows the performance improvement achieved by the current development version of the C^n compiler over the release version 2.51 on 20 benchmarks. Much of the improvement comes from the work on the register allocator and other optimisations becoming aware of poly variables.

Code for the `amber` benchmark generated by the current version of the compiler performs within 20% of hand-coded assembly. (No other similar data is available for comparison, because it makes sense to re-implement assembly programs in C^n only to improve portability.)

We regret that we are unable to present other data because of commercial sensitivity.

8 Future work and conclusion

Programming in C^n tends to require the programmer to restructure programs to expose the parallelism lost when writing in a sequential language. Essentially, the C^n programmer indicates a parallel loop. We believe it is possible to ease the programmer's task of annotating C programs with the **`poly`** qualifier, if the programmer can assert certain properties. We are developing an auto-parallelising compiler module converting C code to valid C^n code using programmer's annotations based on the sieve construct [26]. The resulting C^n code is then fed into the optimisation and code generation phases of the C^n compiler.

ClearSpeed has developed a set of production quality tools targeting the CSX array architecture, including an optimising compiler, assembler, linker and debugger, that make software development in C^n a viable alternative to hand-coding in assembly as loss in performance is far outweighed by the advantages of high-level language programming.

References

1. Behrooz Parhami. SIMD machines: do they have a significant future? *SIGARCH Comput. Archit. News*, 23(4):19–22, 1995.

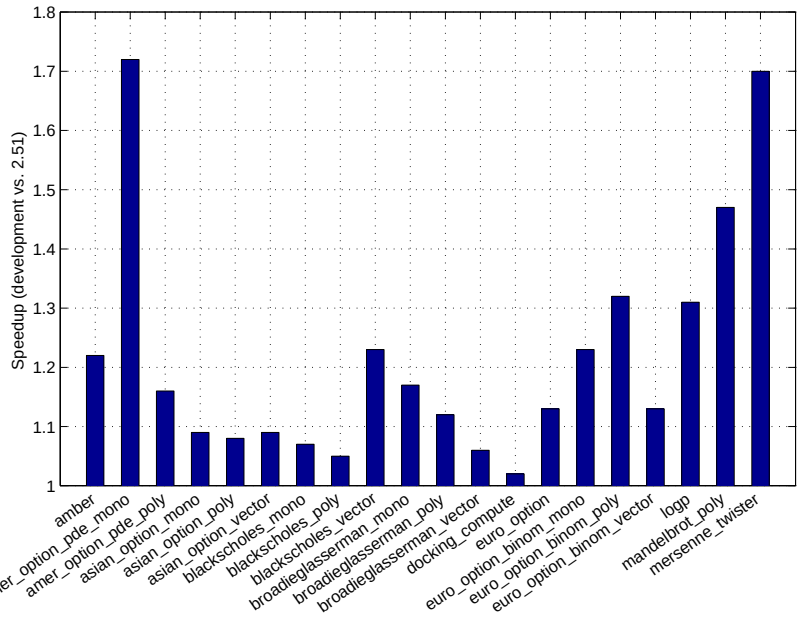

Fig. 1. Performance of the development version against the release version 2.51.

2. George H. Barnes, Richard M. Brown, Maso Kato, David J. Kuck, Daniel L. Slotnick, and Richard A. Stokes. The Illiac IV computer. *IEEE Transactions on Computers*, C-17(8):746–757, August 1968.
3. ClearSpeed Technology. CSX architecture. `http://www.clearspeed.com/`.
4. Lawrence Snyder. The design and development of ZPL. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages (HOPL III)*, pages 8–1–8–37, New York, NY, USA, 2007. ACM Press.
5. D.L. Slotnick. The conception and development of parallel processors—a personal memoir. *IEEE Annals of the History of Computing*, 4(1):20–30, January 1982.
6. Maurice V. Wilkes. The lure of parallelism and its problems. In *Computer Perspectives*. Morgan Kaufmann, San Francisco, CA, USA, 1995.
7. Lawrence Snyder. Type architecture, shared memory and the corollary of modest potential. *Annual Review of Computer Science*, 1:289–317, 1986.
8. Ken Kennedy, Charles Koelbel, and Hans Zima. The rise and fall of High Performance Fortran: an historical object lesson. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages (HOPL III)*, pages 7–1–7–22, New York, NY, USA, 2007. ACM Press.
9. American National Standards Institute. *ANSI/ISO/IEC 9899-1999: Programming Languages – C*. 1999.
10. D. H. Lawrie, T. Layman, D. Baer, and J. M. Randal. Glypnir – a programming language for Illiac IV. *Commun. ACM*, 18(3):157–164, March 1975.
11. K.G. Stevens, Jr. CFD – a Fortran-like language for the Illiac IV. *SIGPLAN Not.*, 10(3):72–76, 1975.
12. R. H. Perrott. A language for array and vector processors. *ACM Trans. Program. Lang. Syst.*, 1(2):177–195, 1979.

13. Kuo-Cheng Li and Herb Schwetman. Vector C: a vector processing language. *Journal of Parallel and Distributed Computing*, 2(2):132–169, May 1985.
14. Kathleen Knobe, Joan D. Lukas, and Guy L. Steele, Jr. Data optimization: allocation of arrays to reduce communication on SIMD machines. *J. Parallel Distrib. Comput.*, 8(2):102–118, 1990.
15. Michael Weiss. Strip mining on SIMD architectures. In *Proceedings of the 5th International Conference on Supercomputing (ICS)*, pages 234–243, New York, NY, USA, 1991. ACM Press.
16. John R. Rose and Guy L. Steele, Jr. C*: An extended C language for data parallel programming. In *Proceedings of the 2nd International Conference on Supercomputing (ICS)*, volume 2, pages 2–16, May 1987.
17. MasPar Computer Corporation. *MasPar Programming Language (ANSI C compatible MPL) Reference Manual*. July 1992.
18. Shorin Kyo, Shin'ichiro Okazaki, and Tamio Arai. An integrated memory array processor for embedded image recognition systems. *IEEE Trans. Computers*, 56(5):622–634, 2007.
19. Peter Christy. Software to support massively parallel computing on the MasPar MP-1. In *Compcon Spring '90: Proceedings of the 35th IEEE Computer Society International Conference*, pages 29–33, March 1990.
20. Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002.
21. Freescale Semiconductor. AltiVec technology programming interface manual, 1999.
22. Con Bradley and Benedict R. Gaster. Exploiting loop-level parallelism for SIMD arrays using OpenMP. In *Proceedings of the 3rd International Workshop on OpenMP (IWOPM)*, June 2007.
23. David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems (ASPLOS-XII)*, pages 325–335, New York, NY, USA, 2006. ACM Press.
24. ACE Associated Compiler Experts. The CoSy compiler development system. `http://www.ace.nl/`.
25. Hansoo Kim. *Region-based register allocation for EPIC architectures*. PhD thesis, Department of Computer Science, New York University, 2001.
26. Anton Lokhmotov, Alan Mycroft, and Andrew Richards. Delayed side-effects ease multicore programming. In *Proceedings of the 13th European Conference on Parallel and Distributed Computing (Euro-Par'07)*, 2007.