

Exploiting Loop-Level Parallelism for SIMD Arrays Using *OpenMP*

Con Bradley and Benedict R. Gaster

ClearSpeed Technology Plc
3110 Great Western Court
Hunts Ground Road
Bristol BS34 8HP UK
{ceb,brg}@clearspeed.com

Abstract. Programming SIMD arrays in languages such as C or FORTRAN is difficult and although work on automatic parallelizing programs has achieved much, it is far from satisfactory. In particular, almost all ‘fully’ automatic parallelizing compilers place fundamental restrictions on the input language.

Alternatively *OpenMP* provides an approach to parallel programming that supports incremental improvements to applications that places restrictions in the context of preserving semantics of a parallelized section. *OpenMP* is limited to a thread based model that does not define a mapping for other parallel programming models.

In this paper, we describe an alternative approach to programming SIMD machines using an extended subset of *OpenMP* (*OpenMP SIMD*), allowing us to model naturally the programming of arbitrary sized SIMD arrays while retaining the semantic completeness of both *OpenMP* and the parent languages.

Using the CSX architecture we show how *OpenMP SIMD* can be implemented and discuss future extensions to both the language and SIMD architectures to better support explicit parallel programming.

1 Introduction

With the rise of both SIMD processors, such as ClearSpeed’s SIMD Array Processor [1], and the current unstoppable trend to put more and more cores on to a single die, parallel programming is back with a bang! For anyone who has tried to write a reasonably large parallel program then this will be seen with at least a slight frown. Why? In a nutshell parallel programming is hard and a choice must be made as to which of the many parallel programming languages to use.

While there is no magic wand to wave that can easily resolve the problem of parallel programming, there has been a number of recent proposals for programming languages, including Brook [2], Cilk [3], and C^n [4], and a number of simple language extensions for parallel programming, including *OpenMP* [5] and X10 [6]. Other than Brook, which introduces a programming model based on Streams, C^n , *OpenMP*, and X10 are all based upon existing programming

languages C¹ [7] and Java [8], respectively, and as such provide parallel programming in the context of well understood and widely used programming paradigms. It is only *OpenMP* that has seen wide adoption in the marketplace and is the only major parallel programming extension to C that is supported by the major compiler vendors, including Intel (Intel C Compiler) and Microsoft (Visual Studio).

OpenMP as defined is intended as a light-weight approach to programming threads. In this paper, we describe an alternative interpretation of *OpenMP*, called *OpenMP SIMD*, that rather than mapping to the standard runtime parallelism of threads looks instead at how it can be deployed for programming SIMD machines. In particular, we show that the loop level parallelism described by *OpenMP* maps well to data-parallelism as an alternative to the runtime approach described traditionally by *OpenMP*.

While we believe this alternative semantics for *OpenMP* is valid for any SIMD architecture, including Intel's Streaming SIMD Extensions [9], we focus here on programming the CSX architecture [1]. The CSX architecture is a SIMD array with a control unit and a number of processing elements (PE), with each PE having a relatively small private storage and supports an asynchronous I/O mechanism to access larger shared memory.

To date the CSX architecture would have been programmed using the explicit parallel programming language *Cⁿ* but such an approach has some drawbacks:

- Any code must be completely ported to use explicit parallelism before it can be run; and
- Users have large amounts of C and *OpenMP* code that they do not want to re-write.

A mapping from *OpenMP* to either *Cⁿ* or directly to the CSX architecture provides an alternative approach as *OpenMP* provides a piecemeal approach to parallelizing a given program and by definition supports any legacy C and *OpenMP* code.

The remaining sections of this paper are as follows; Section 2 introduces the CSX architecture, including how it is intended to be programmed; Section 3 is broken into a number of sub-sections describing the translation of *OpenMP SIMD* to the CSX architecture; Section 4 gives two examples of *OpenMP SIMD* in practice, highlighting both the translation and the small but necessary extensions to *OpenMP*; and finally, Section 5 concludes.

2 The CSX Architecture

The CSX family of co-processors [1] are based around a multi-threaded array processor (MTAP) core optimized for high performance, floating point intensive applications. The most recent product includes an MTAP processor, DDR2 DRAM interface, on-chip SRAM and high speed inter-processor I/O ports.

¹ *OpenMP* provides models for C++ and FORTRAN but we choose to focus on the C++ variant.

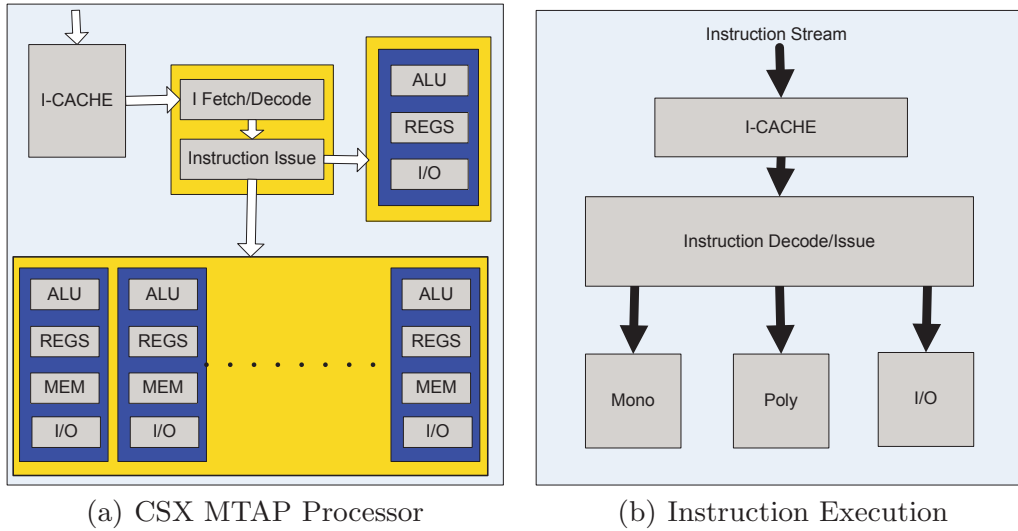
**Fig. 1.** CSX Architecture

Figure 1(a), visualizes the MTAP architecture showing a standard, RISC-like, control unit with instruction fetch, caches, and I/O mechanisms. A single instruction stream comprising of fairly standard, three operand instructions, is made up of mono (non-parallel) instructions executed by a single RISC-style processor and SIMD instructions (poly) executed, in lock-step, by a number of poly execution (PE) cores².

As noted above the processor executes a single instruction stream; each instruction is sent to one of the functional units, which may be in the mono, or poly execution units or one of the I/O controllers. Figure 1(b) shows this visually.

The main change from programming a standard processor is the concept of operating on parallel data. Data to be processed is assigned to ‘variables’ which have an instance on every PE core and are operated on in parallel: we call this poly data. Intuitively, this can be thought of as a data vector which is distributed across the PE core array.

Variables which only need to have a single instance (e.g. loop control variables) are known as mono variables; they behave exactly like normal variables on a sequential processor and can be ‘broadcast’ to all PEs if required in the poly domain.

Code for the mono and poly execution units is handled similarly, except the handling of conditional code 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 perform as though a *NOP* has been issued. 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

² The current CSX600 processor has 96 PEs.

remove the effect of the condition. This makes handling nested conditions and loops very 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 can directly access only its own storage and additionally can communicate with its left and right neighbors via what is known as the swizzle path, forming a network of PEs. Data is transferred between PE (poly) memory and registers via load/store instructions while a family of instructions is provided for swizzle communication.

In the following mapping of *OpenMP* to the CSX architecture it is important to note that mono memory (i.e. card memory) and local PE (poly) memory are allocated separate address spaces which require 32 and 16 bit pointers, respectively.

3 *OpenMP SIMD*

As discussed in the introduction we are looking at using *OpenMP* to exploit loop level parallelism and as such the translation described below supports only one parallel construct, namely, the work-sharing loop. For this construct, we retain the fork-join ‘threading’ model. The master thread runs on the mono processor and is responsible for creating the ‘thread’ gang to execute the iterations of the work-sharing loop. Where the translation differs from the multi-core threading model traditionally associated with *OpenMP* is that each ‘thread’ is modeled as a PE element. At the end of the loop there is an implicit barrier. *OpenMP* expects the master thread to participate in the loop calculations but in our model this is not the case due to the asymmetry between the PE and mono processor architecture. We assume that all housekeeping operations are carried out on the mono processor.

3.1 Work-Sharing Constructs

The *OpenMP SIMD* translation actually provides two alternative mappings described in the following to sections.

Poly Loop Translation. *OpenMP* states that the index variable of the loop to be parallelized becomes private for the duration and thus a straightforward translation of the *OpenMP* loop pragma to *OpenMP SIMD* is to make the control variable poly and to increment by the number of PEs each time around the loop. Such a translation has the effect of unrolling the loop by the number of PEs. Mapping the test expression of the loop to use the poly control variable and increment with the defined increment plus the number of PEs in the final clause of the loop definition, provides all that is needed for the loop control definition. All that is remaining is to translate the body of the loop. The mapping is defined as follows:

```
#pragma omp parallel for clause clause ...
for (i = e1; i op e2; i = i ops incr) { stmts; }
```

is translated to³:

```
{ poly int  $i_p = e_1$ ;
  for ( $i_p = i_p \text{ } op_s \text{ } get\_penum() * incr$ ;  $i_p \text{ } op \text{ } e_2$ ;  $s_{1_p}$  )
    {  $stmts_p$ ; }
}
```

Here *OpenMP* requires that *op* is one of $<$, $>$, $\leq$, or $\geq$ and *op_s* is $+$ or $-$. With this s_{1_p} becomes⁴ $i_p = i_p \text{ } ops \text{ } (incr * ___NUM_PES_)$.

Finally, we must provide a translation for $stmts_p$. Firstly, all automatics defined locally to the scope of the loop become poly, i.e. privates in *OpenMP*, and any reference to the index variable *i* is replaced with i_p . Secondly, the translation must replace all accesses to arrays indexed by newly introduced poly expressions with poly temporaries, and insert calls to runtime routines that copy data between memory spaces, accounting for the strided access implied by parallelizing by the number of PEs. For example, consider the following loop statement which uses *i* to index into an array of mono doubles:

```
double tmp = V[i];
```

In the first part of the translation this becomes:

```
poly double  $tmp_p = V[i_p]$ ;
```

Assuming that *V* is defined globally and thus in *OpenMP* terminology shared, then this expression becomes the list of statements⁵:

```
poly double  $tmp_p$ ;
double * poly  $V_p = ((double * poly) V) + i_p$ ;
memcpym2p(& $tmp_p$ ,  $V_p$ , sizeof(double));
```

where *memcpym2p* is one of a family of memory transfer routines forming part of the *OpenMP SIMD* runtime, with this routine in particular performing data transfers from mono memory into local PE memory.

Mono Loop Translation. The poly loop translation given above is both obvious and straightforward but in practice may not be the most effective approach to implementing the loop on a SIMD style architecture.

To illustrate this consider the simple poly loop⁶:

```
for ( $i_p = 0$ ;  $i_p < bound$ ;  $i_p++$ ) { ... }
```

As all control flow is handled in the mono processor a poly loop must be implemented by evaluating the conditional $i_p < bound$ such that all, some, or none of the PEs are disabled. Then a mono instruction is issued to determine if all of the PEs are disabled and if so then exit the loop, otherwise continue with

³ The *OpenMP SIMD* runtime function `poly int get_penum(void)` returns the unique number assigned to each PE, given in the sequence 0..95.

⁴ `__NUM_PES__` is defined statically as the number of PEs for a given implementation.

⁵ The type expression `double *poly` should be read as a poly pointer to a mono double.

⁶ Assuming *bound* to be mono.

the next iteration. The overhead of checking wheather the PEs are disabled or not is expensive and is often not necessary, as a mono loop can be used instead.

To explain the mono loop translation we consider again the loop from the previous section:

```
#pragma omp parallel for clause clause ...
for (i = e1; i op e2; i = i op incr) { stmts; }
```

which can be translated into the mono loop:

```
for (i = e1; i op (e2 - (e2%__NUM_PES__));
     i = i op (incr * __NUM_PES__))
{ stmtsp }
```

Finally a translation for *stmts_p* is required, which is the same as in the previous section except in the case when the control variable *i* is referenced. In this case the translation needs to account for the fact that *OpenMP*, and thus the translation, requires that for the duration of the loop *i* is marshaled as a private and to account for the unrolling factor the PE number must be added into any expression referencing *i*. Again considering the loop defined statement:

```
double tmp = V[i];
```

the first part of the translation gives:

```
poly double tmpp = V[i];
```

As *i* must be treated as a private, accounting for the stride due to loop unrolling, the final translated code becomes:

```
poly double tmpp;
double * poly Vp = ((double * poly) V) +
                    (i + get_penum() * incr);
memcpym2p(&tmpp, Vp, sizeof(double));
```

If one is not careful it is easy to think at this point that the translation for mono loops provided above is enough, but we must consider the case when the number of PEs is not a multiple of the number of PEs. To see that there may in fact be a problem, consider again the translated test condition for the loop *i op (e₄ - (e₄%__NUM_PES__))*.

The expression *(e₄ - (e₄%__NUM_PES__))* always results in an expression that is a multiple of the number of PEs. It is straightforward to see that any remaining number of iterations will not be performed by the loop, and as is the case with optimizations like software-pipelining [10] the mono loop translation requires some additional post loop code to account for these missing iterations. This is easily achieved by augmenting the translation with the post loop code:

```
if (get_penum() < (e4%__NUM_PES__)) { stmtsp; }
```

Where *stmts_p* is the same set of statements from the translated loop body and the conditional simply disables any PEs not required for the remaining calculation. An important consideration is to note that as the conditional is

poly then $stmts_p$ will always be executed. If the condition $get_penum() < e_4\%_NUM_PES_$ evaluates to false this could be expensive depending on the size of the sequence $stmts_p$. This overhead can be easily avoided by adding in an additional mono conditional of the form:

```
if (e4%_NUM_PES_)
```

3.2 Data Environment

OpenMP defines a set of constructs to describe how data is shared in parallel regions and we must capture how these are mapped into *OpenMP SIMD*.

OpenMP defines storage as being either private to a thread or shared, i.e. globally shared for all threads. The obvious translation into *OpenMP SIMD* is to map private variables to PE memory (poly) and shared to mono memory.

The remainder of this section considers the consequences of this simple translation and provides some extensions to *OpenMP* when the mapping does not quite capture the required semantics.

Sharing Attribute Rules. *OpenMP* defines a set of rules for describing sharing attributes of variables within parallel constructs. These are variables without explicit sharing attributes, i.e. the default rules, and we need to confirm their semantics for *OpenMP SIMD*. These rules apply in two cases:

1. Variables explicitly declared within the work sharing loop. For our translation the default rules work well and do not need modification.
2. Variables within called functions. *OpenMP* refers to these as being within a region but not a construct, meaning that syntactically a variable does not appear in a parallel construct but at runtime the function is called from a parallel region. The rules for data sharing are as in the explicit case but with one major addition: Formal arguments passed by reference inherit the data sharing attributes of the actual parameters.

This implies that at compile time, it is impossible for a compiler to know if a function will be called from a parallel region and thus what the data sharing attributes of its formal parameters will be. In order for the translation to *OpenMP SIMD* to work efficiently the user is required to provide this information. This leads to the introduction of one of three extensions to *OpenMP* as it stands:

```
#pragma omp private_function <name>
```

This *private_function* clause marks the function $< name >$ as being called from a parallel region and hence the default data sharing rules are to be applied. The default is that all functions are serial, i.e. mono, but this can be overridden via a default clause. In a typical program, parallel functions can be grouped thus reducing the amount of explicit clauses needed.

Unlike standard *OpenMP* the formal arguments to a function for the CSX architecture can be either mono or poly but not both. This leads to the observation

that data-sharing attributes of formal parameters need to be specified at compile time as they affect code generation. This is captured in *OpenMP SIMD* by introducing a second extension to *OpenMP*:

```
#pragma omp param <name> (list)
```

Where *list* is defined as a comma separated list, one entry per formal parameter to *< name >*; each entry can be of the form:

- private - formal is private (i.e. poly)
- shared - formal is shared (i.e. mono)

Variable argument parameters and function pointers are not supported.

The final extension needed for the translation is analogous to that for a function's formal arguments but applies to the return value as it can be either mono or poly but not both:

```
#pragma omp private_return <name>
```

This *private_return* clause marks the function *< name >* as having a private return value, i.e. poly. The default is that functions return a shared value, i.e. mono.

Any function can have any number of sharing directives applied to describe all possible calling conditions. This defines a set of 'signatures' describing the set of possible calling conditions. Given these extra annotations a compiler might choose to use a C++ [11] style name mangling scheme at function definition time and at function call sites to generate the required function signature. For example, a simple void function, `void foo(void)`, with no parameters which can be called from both a parallel and non-parallel region could have two instances, *_POLY_foo* and *_MONO_foo*. In a serial context, the mono version is called, and in a poly context the poly version is called. The linker will then select the appropriate versions. This scheme is also applied to a function's formal arguments to generate C++ style mangled names.

It is important to note that for this system to be sound, it must at all times be known if a piece of code is in a parallel region or not. This is where *OpenMP SIMD* differs slightly from *OpenMP* as it requires this to be enforced by the user through application of the above extensions.

Data-Sharing Attribute Clauses. *OpenMP* assumes that multiple threads execute within the same shared address space and thus inter-thread communication is straightforward. Data is sent to other threads by assigning to shared variables. *OpenMP* also provides support for a variable to be designated as private to each thread rather than shared, causing each thread to get a private copy of a particular variable for the duration of the parallel construct.

This notion of shared and private fits well within our SIMD model of OpenMP, producing the mapping of *private* to *poly* and *shared* to *mono*. The *firstprivate* and *lastprivate* clauses are also supported with extra runtime functions provided to perform the initialization and finalization.

The reduction clause describes the application of a reduction operator to be applied to a list of private variables. The reduction operator comes from the base

language and is applied to each (private) copy of the variable at the end of the parallel construct. This construct is supported by introducing a private copy of the reduction variable, initialized to the particular reduction operator's identity, at the start of the parallel construct. The reduction is then performed across all PEs. The swizzle path is used to reduce into either the top or bottom PE, where the data can be quickly transferred into mono space and assigned to the original reduction variable. For example, consider the following reduction clause and loop:

```
#pragma omp parallel for reduction(+:acc)
for (...) { ... ; acc += ...; }
```

which becomes⁷:

```
poly double acc_p = 0.0;
for (...) { ... ; acc_p += ... ; }
acc += reduce_sum(acc_p);
```

4 An Example

As an example of *OpenMP SIMD* we consider performing a Monte-Carlo simulation to calculate European option prices [12]. The intention here is to show the extensions in practice. Due to space restrictions we choose not to show the translated code but rather concentrate our discussion of the extensions themselves.

Monte-Carlo simulation problems include derivative pricing, portfolio risk management and Markov chain simulation. In financial modeling, the problem of finding the arbitrage-free value of a particular derivative can be expressed as the calculation of a given integral. As the number of dimensions grows, today's financial mathematicians use Monte-Carlo simulation methods to solve problems in a tractable manner.

Monte-Carlo simulations can be easily parallelized, since tasks are largely independent and can be partitioned among different processors, and the need for communication is limited. Due to the simple but highly computational nature of Monte-Carlo it is clear that any performance increase in their implementation can lead to substantial speed improvements for the entire system and significant advances in time to result. The following function implements Monte-Carlo option pricing in *OpenMP SIMD*:

```
void MCOption(double S, double E, double r,
              double sigma, double T,
              int nSimulations,
              double *Pmean, double *width) {
```

⁷ The function *double reduce_sum(poly double)* is one of a family of reduction operators provided as part of the *OpenMP SIMD* runtime, with this function performing a double precision sum across the PEs placing the scalar result into mono.

```

double sumx, sumx2, tmp1, tmp2, stdev;
int i;

sumx=0; sumx2=0;
tmp1=(r-0.5*sigma*sigma)*T; tmp2=sigma*sqrt(T);

#pragma omp for reduction(+:sumx,+:sumx2)
for ( i = 0; i< nSimulations; i++)
{
    double Svals, Pvals;
    Svals = S* exp (tmp1 +tmp2*gaussrand());
    Pvals = Max( Svals - E, 0);
    sumx  +=  Pvals;
    sumx2 += (Pvals*Pvals);
}
stdev = sqrt(((double)nSimulations*sumx2-sumx*sumx)/
              ((double)nSimulations*(double)(nSimulations-1)));

*width = exp(-r *T) * 1.96 * stdev /
          sqrt((double)nSimulations);
*Pmean = (exp(-r *T) * (sumx / nSimulations));
}

```

The above code calls the function *gaussrand* from within a parallel region and as a consequence the programmer must tell the compiler, through use of the *OpenMP SIMD* extensions, to produce an instance suitable for calling from within the parallel construct:

```

#pragma omp private_function gaussrand private(V1,V2,S)
#pragma omp private_return gaussrand
double gaussrand(void) {
    static double V1, V2, S; static int phase = 0;
    double X;
    ...
    return X;
}

```

Here the application of clause *private_function* expresses the requirement that *gaussrand* is intended to be called from within a parallel region or loop and *private_return* implies that the function will produce a private, i.e. poly, result. This highlights an important difference between the threaded approach of *OpenMP*'s out-of-step execution when compared with *OpenMP SIMD*'s lock-step execution.

Using our prototype compiler early benchmarking of the Monte-Carlo simulation for European Option pricing has shown excellent results for the OpenMP SIMD implementation running on ClearSpeed's Advance Board. These benchmarks were performed on a 2.33GHz dual-core Intel Xeon (Woodcrest, HP DL

Table 1. Monte-Carlo - ClearSpeed Board vs. host-only performance comparison

GNU C Compiler with Math Kernel Library	Intel C Compiler with Math Kernel Library	ClearSpeed Advance Board	ClearSpeed Advance Board plus Intel Math Kernel Library
36.6M	40.5M	206.5M	240M

380 G5 system) and the results are shown in Table 1. The results show a 5.15x performance improvement, for 100,000 Monte-Carlo simulations performed using a single ClearSpeed Advance board when compared to simply using an Intel Xeon processor, the Intel C compiler and Math Kernel Library (MKL). When compared against the GNU C compiler and the Intel MKL the performance improvement increases to 5.7x. However, combining both the general purpose Intel processor and MKL with the Advance board achieves the best result at 240M samples per second and a performance factor of up to 6x compared to using just the host processor, Intel C compiler and MKL.

5 Conclusion

In this paper, we have introduced the programming extensions *OpenMP SIMD* for programming arbitrary-sized SIMD arrays. This language is understood by providing an alternative semantics for *OpenMP*. While there have been a number of extensions to existing programming languages for accessing the features of SIMD machines, including C^n and C^* [4,13], to our knowledge we are the first to propose using *OpenMP* to program SIMD machines.

We have developed a set of prototype tools targeting the CSX array architecture and are evaluating using *OpenMP SIMD* as an alternative to the explicit parallel programming language C^n . Early results indicate that it provides at least the same functionality and ability for performance.

OpenMP has a relaxed-consistency shared-memory model that allows threads to have their own view of global data which only required to be synched up at specific points including the explicit flush operation. Any compiler implementing OpenMP SIMD has the ability to promote shared variables to registers and even cache them in PE memory if needed. The flexibility this model allows provides a compiler with the chance to optimize through data migration between mono memory, PE memory and host memory.

In conclusion we believe that there is an important and fascinating concept at the heart of *OpenMP SIMD*. Essentially we are translating runtime parallelism into data parallelism. The model can only work if this fundamental difference in the model is not observable. We believe that this is sound as long as we can only observe a program's behavior by its effect on variables. Since we believe we have correctly defined the data parallelism then there should be no observable difference in behavior.

Acknowledgments

We would like to thank Ray McConnell who originally introduced us to OpenMP and David Lacey for his useful feedback on the translation.

References

1. ClearSpeed Technology Plc: White paper: CSX Processor Architecture (2004)
2. Buck, I.: Brook: A Streaming Programming Language. Stanford University (2001)
3. Frigo, M.: Portable High-Performance Programs. PhD thesis, MIT Department of Electrical Engineering and Computer Science (1999)
4. Gaster, B.R., Hickey, N., Stuttard, D.: The Cⁿ Language Specification. ClearSpeed Technology Plc. (2006)
5. OpenMP Architecture Review Board: OpenMP Application Program Interface (2005)
6. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. In: OOPSLA 2005. Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, pp. 519–538. ACM Press, New York (2005)
7. International Standards Organization: ISO/IEC 98899 - Programming Language C. (1999)
8. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Specification, 3rd edn. Addison-Wesley, Reading (2005)
9. Ramanathan, R.: Intel white paper: Extending the world's most popular processor architecture. Intel Corp. (2006)
10. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures. Morgan Kaufmann, San Francisco (2002)
11. Stroustrup, B.: The C++ Programming Language, 2nd edn. Addison-Wesley, Reading (2000)
12. Wilmott, P.: Quantitative Finance, 2nd edn. John Wiley & Sons Ltd., Chichester (2000)
13. Rose, J., Steele, G.: C*: An extended C language for data parallel programming. In: International Conference on Supercomputing, vol. 2, pp. 72–76 (1987)