Offload – Automating Code Migration to Heterogeneous Multicore Systems

Pete Cooper[1], Uwe Dolinsky[1], Alastair F. Donaldson[2], Andrew Richards[1],
Colin Riley[1], and George Russell[1]

[1] Codeplay Software Ltd., Edinburgh, UK
[2] Oxford University Computing Laboratory, Oxford, UK

Abstract. We present *Offload*, a programming model for offloading parts of a C++ application to run on accelerator cores in a heterogeneous multicore system. Code to be offloaded is enclosed in an *offload* scope; all functions called indirectly from an offload scope are compiled for the accelerator cores. Data defined inside/outside an offload scope resides in accelerator/host memory respectively, and code to move data between memory spaces is generated automatically by the compiler. This is achieved by distinguishing between host and accelerator pointers at the type level, and compiling multiple versions of functions based on pointer parameter configurations using *automatic call-graph duplication*. We discuss solutions to several challenging issues related to call-graph duplication, and present an implementation of Offload for the Cell BE processor, evaluated using a number of benchmarks.

1 Introduction

In this paper, we contribute towards the goal of programming heterogeneous multicore processors like the Cell Broadband Engine (BE) [1] using a familiar threading paradigm. To this end, we present Offload, a programming model and implemented system for offloading portions of large C++ applications to run on accelerator cores. Code to be offloaded is wrapped in an *offload* block, indicating that the code should be compiled for an accelerator, and executed asynchronously as a separate thread. Call graphs rooted in an offload block are automatically identified and compiled for the accelerator; data movement between host and accelerator memories is also handled automatically. The Offload approach allows the development of portable multi-threaded applications for homogeneous *and* heterogeneous platforms: the language extensions we propose are minimal, and preprocessor macros can be used to select between, for example, a POSIX thread and an Offload thread on a homogeneous or heterogeneous platform respectively. The advantages to this approach are evident: large source bases can be incrementally migrated to heterogeneous platforms with relatively little change; portability across heterogeneous and homogeneous platforms is possible, and the burden of writing data movement and accelerator start-up and clear-down code is lifted from the programmer.

After discussing the challenges of heterogeneous multicore programming we make the following contributions. We present the Offload language extensions, and describe

Y.N. Patt et al. (Eds.): HiPEAC 2010, LNCS 5952, pp. 337–352, 2010.

automatic call-graph duplication, where multiple versions of a function[1] are compiled for an accelerator, based on the contexts in which the function is called. We then discuss our solutions to challenging problems associated with call-graph duplication in the presence of pointer types for separate memory spaces, function pointers and virtual methods, and multiple compilation units. Finally, we present experimental results for a Cell BE implementation of Offload, evaluated using several benchmarks.

2 Programming Heterogeneous Multicore Processors

Processor manufacturers are increasingly opting to deliver performance improvements by implementing processors consisting of *multiple* cores due to problems in obtaining further performance increases from single core processors. Multicore processors may be *homogeneous*, consisting of n identical cores, or *heterogeneous*, where some or all of the cores differ in specialisation. In principle, a homogeneous multicore processor with n cores connected to shared memory can offer a factor of n-times execution speedup over a single-core processor at the same clock rate. However, contention for access to shared memory may lead to a performance bottleneck, known as the *memory wall*, where adding further cores quickly leads to diminishing returns.

The memory wall problem has led to a recent mainstream shift towards *heterogeneous* multicore processors in the *host with accelerators* pattern, where a host core connected to main memory coordinates a number of possibly diverse processing element (PE) cores each equipped with private "scratch-pad" memory. Independent calculations can be processed in parallel by separate cores with reduced contention for shared memory. The PE cores need only access shared main memory via direct memory access (DMA) to read input data, write results, and communicate with one another. The Cell BE [1] is one such processor design, consisting of a Power Processor Element (PPE) host with 8 Synergistic Processor Element (SPE) accelerator cores.

The use of scratch-pad memories can boost performance, but increases the complexity of concurrent programming. The programmer can no longer rely on the hardware and operating system to seamlessly transfer data between levels of the memory hierarchy, and must manually orchestrate data movement using DMA. Experience writing and debugging industrial software for heterogeneous multicore processors has identified the following key problems:

Separate programs are required for different cores. Distinct cores may have entirely different instruction set architectures, making it necessary to write, compile and maintain separate versions of functions for each type of core, as well as platform-specific "glue" code, to start up and clear down accelerator threads.

Data movement is untyped and unsafe. Low level data movement primitives operate on untyped bytes and data words. Mistakes in interpretation of untyped data or misuse of DMA primitives can lead to nondeterministic bugs that are hard to reproduce and fix.

Furthermore, many large applications have already been successfully parallelized for homogeneous multicore processors using POSIX or Windows threads. In this case the problem is not to find potential parallelism, but rather exploit already identified potential

[1] We use *function* to refer to functions and methods in general.

by offloading threads to accelerator cores with minimal disruption to an existing codebase. The Offload approach aims to ease the programming of heterogeneous multicore systems via a *conservative* extension to the C++ language.

We focus on systems where there is one type of accelerator core, equipped with a small scratch-pad memory, and with sufficient functionality to be programmed in C. The approach could be naturally extended to support multiple types accelerator.

3 Offload Language Extensions

3.1 Offload Scopes

The central construct of the Offload system is the *offload block*, a block prefixed by the **offload** keyword. Code outside an offload block executes on the coordinating host core; code inside an offload block executes on an accelerator core in a separate thread.

Offload blocks extend the syntax of C++ expressions as follows:

$$\mathit{Expr} ::= \ldots \;\Big|\; \texttt{\textbf{offload}}\ \mathit{Domain}?\ \mathit{Args}?\ \texttt{\{}\ \mathit{Compound\text{-}Stmt}\ \texttt{\}}$$

$$\mathit{Args} ::= \texttt{(}\ \text{list of variable names}\ \texttt{)} \qquad \mathit{Domain} ::= \texttt{(}\ \text{list of function names}\ \texttt{)}$$

An offload block evaluates to a value of type `offload_handle_t`, an opaque type defined in header files supplied with the Offload system. The expression has the side-effect of launching a thread on an accelerator core. This *offload thread* executes *Compound-Stmt*, with standard sequential semantics. Host execution continues in parallel. The host can wait for the offload thread to complete by calling library function `offload_join`, passing as an argument the handle obtained on creating the offload thread. Multiple offload threads can be launched to run in parallel, either via multiple offload blocks, or by enclosing an offload block in a loop.

An offload thread can access global variables, as well as variables in the scope enclosing the associated offload block. Additionally, an offload block may be equipped with an argument list – a comma-separated list of variables names from the enclosing scope. Each variable in this list is copied to a local variable in the offload thread with the same name; references to this name inside the offload block refer to the local variable. An offload block may also be equipped with a *domain*, which we discuss in §5.2.

These concepts are illustrated by the following example:

```
int main() {
  int x = ...;
  int y = ...;
  offload_handle_t handle = offload(y) {
    // runs on accelerator, 'y' passed by value
    ... = x; x = ...; // 'x' accessed in enclosing scope
    ... = y; y = ...; // local copy of 'y' accessed
  };
  ...       // host runs in parallel with accelerator
  y = ...; // changes to 'y' do not affect offload thread
  ...
  offload_join(handle); // wait for offload thread
}
```

For brevity, we omit details of parameters and handles in the examples that follow.

The **offload** keyword can also be used as a function qualifier. A function with the **offload** qualifier is called an *offload function*, and function names can be overloaded using **offload**. We refer to offload functions and offload blocks as *offload scopes*. Offload functions can only be called from offload scopes, but it is permissible to call a non-offload function from an offload scope; this is discussed in detail in §4. We illustrate offload functions using the following example:

```
void f() { ... }         // (1)
offload void f() { ... } // (2)
offload void g() { ... } // (3)
void h() { ... }         // (4)

int main() {
  f();   // calls (1) on host
  g();   // error, 'g' is an offload function
  offload {
    f(); // calls (2) on accelerator
    g(); // calls (3) on accelerator
    h(); // calls (4) on accelerator
  }
}
```

3.2 Outer Pointers and Data Movement

Data declared inside an offload scope resides in accelerator memory. We distinguish pointers to local memory from pointers to host memory, referring to the latter as *outer pointers*. An additional qualifier for pointers and references,[2] the **outer** qualifier, specifies that a pointer refers to host memory. Pointers outside an offload scope have the **outer** qualifier by default. Assignment between outer and non-outer pointers is illegal; this ensures that an outer/non-outer pointer does not refer to data residing in accelerator/host memory.

Dereferencing an outer pointer in an offload scope causes data to be moved between host and accelerator memory. Data movement may be achieved via direct memory access (DMA). However, unless synchronization primitives are used to guard access to host memory, non-volatile data can be cached locally, so data movement may be implemented using a software cache, or via double-buffered streaming DMA if accesses have a regular pattern. Our Cell BE implementation (see §6) uses a software cache by default, and provides library functions to flush or invalidate the cache. These functions can be used in conjunction with mutexes to allow sharing of data between host and offload threads.

Consider the following listing, where a -> b indicates data movement from a to b:

```
offload void f(outer float * p) {
  *p = *p + 42.0f; // host -> accelerator, accelerator -> host
}
```

[2] Henceforth we will only talk about pointers; everything we say about pointers applies to references also.

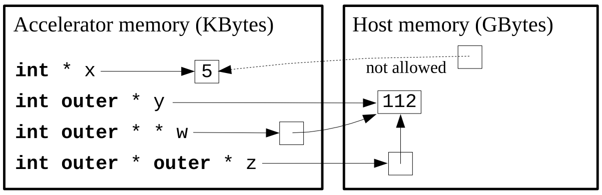


Fig. 1. Examples illustrating the **outer** qualifier for pointers

```
float a;

int main() {
  offload {
    outer float * p = &a;
    float b = *p;    // host -> accelerator
    *p = b + 42.0f; // accelerator -> host
    float c = a;    // host -> accelerator
    a = c;          // accelerator -> host
    p = &b;         // error! '&b' is not outer
    f(p);           // legal function call
    f(&b);          // error! '&b' is not outer
  }
}
```

Taking the address of global variable a obtains an outer pointer p, through which data can be transferred between host and accelerator memory. Accessing host variable a directly from an offload scope also results in data movement. The listing illustrates illegal assignment between outer and non-outer pointers.

Outer pointers allow data transfers to be expressed without exposing the programmer to low-level, non-portable operations. By regarding outer and local pointers as incompatible, the compiler is able to ensure that transfers are well typed. Fig. 1 provides further illustration of the use of outer pointers in memory-space separation.

C++ permits overloading on the basis of **const** and **volatile** qualifiers. The Offload language extends this to the **outer** qualifier, allowing functions to be overloaded with different combinations of outer pointers. For an instance method on a class, the **outer** qualifier can be applied to the **this** pointer by placing **outer** after the closing bracket of the parameter list for the method; an example of this is given in §4.1.

4 Call-Graph Duplication

Suppose we require that *only* offload functions can be called from offload scopes. We call this the *strict* requirement. In this case, the compiler knows exactly which functions to compile for the accelerator (the offload functions) and host (the non-offload functions). Furthermore, the pointer signature for an offload function specifies exactly those pointers for which dereferences correspond to data movement operations (the outer pointers). The drawback is that the programmer may have to manually duplicate functionality to match the contexts in which a function is called. We illustrate the process of manual function duplication with the *strict* requirement, then show how automatic call-graph duplication can be used to handle programs that do not satisfy this requirement.

4.1 Manual Function Duplication

Consider the following class:

```
class SecretKeeper {
  int[SIZE] secrets;
public:
  int getSecret(int * p) const { return secrets[*p]; }
};
```

Listing 1. A simple C++ "secret keeper" class.

The following listing declares a `SecretKeeper` object both outside and inside an offload block, and calls the `getSecret` method on each object with a combination of outer and local pointers:

```
int main() { ...
  SecretKeeper outKeeper;
  int x; ...
  // normal method call on host
  int secretSum = outKeeper.getSecret(&x);
  offload {
    SecretKeeper inKeeper;
    int y;  ...
    secretSum +=
      inKeeper.getSecret(&y)  // (1) local 'this', local 'p'
    + inKeeper.getSecret(&x)  // (2) local 'this', outer 'p'
    + outKeeper.getSecret(&y) // (3) outer 'this', local 'p'
    + outKeeper.getSecret(&x) // (4) outer 'this', outer 'p'
  }; ...
}
```

Listing 2. Calling `getSecret` with various pointer configurations.

To satisfy the *strict* requirement, the programmer must define additional offload versions of `getSecret` for the four contexts in which the method is called inside the offload block:

```
class SecretKeeper {
  // as before, with additional methods:
  offload int getSecret(int * p) {
    return secrets[*p]; // matches context (1)
  }
  offload int getSecret(outer int * p) {
    return secrets[*p]; // matches context (2)
  }
  offload int getSecret(int * p) outer {
    return secrets[*p]; // matches context (3)
  }
  offload int getSecret(outer int * p) outer {
    return secrets[*p]; // matches context (4)
  }
};
```

Listing 3. Manually duplicating the `getSecret` method.

Although the bodies of these methods are syntactically identical, their compilation results in different data movement code. For example, in case (2), dereferencing outer pointer `p` results in a host-to-accelerator data movement operation, while indexing into local member `secrets` is a normal array lookup; in case (4) both dereferencing `p` and indexing into `secrets` require host-to-accelerator data movement operations: the outer **`this`** pointer means that the `secrets` member is located in host memory.

Manual function duplication with the *strict* requirement is time-consuming and results in many similar versions of the same function, which must all be maintained. However, when a program satisfies the *strict* requirement it can be compiled appropriately for the host and accelerator cores. We now show how a program that does *not* satisfy the *strict* requirement can be automatically translated into a form where the *strict* requirement is satisfied, from which it can be successfully compiled.

4.2 Automating the Duplication Process

Suppose a function f has been declared with the following signature:

$T_0\ f(T_1\ p_1, \ldots,\ T_n\ p_n)\ \{\ \mathit{body}\ \}$

Note that f is *not* an offload function. Now suppose f is invoked from an offload scope, violating the *strict* requirement, in the following context:

$e_0\ =\ f(e_1, \ldots,\ e_n);$

For $0 \leq i \leq n$, let U_i denote the type of expression e_i (where e_0 evaluates to an lvalue), and suppose that U_i and T_i are identical if **`outer`** qualifiers are ignored. In other words, the function application is well-typed if we ignore outer pointers. Then we can generate an overloaded version of f as follows:

`offload` $U_0\ f(U_1\ p_1, \ldots,\ U_n\ p_n)\ \{\ \mathit{body}\ \}$

Let f' denote the newly generated version of f. Functions f and f' are identical, except that f' has the **`offload`** qualifier, and pointer parameters of f' may have the **`outer`** qualifier if they are passed outer pointers as actual arguments. The call to f from the offload scope now obeys the *strict* requirement, since it refers to the offload version of f, *i.e.* f'.

If body itself contains calls to non-offload functions then function duplication will be applied to these calls, with respect to the version of body appearing in f', so that f'

only calls offload functions. This process continues until function duplication has been applied to all call-graphs rooted in offload scopes, hence the term automatic *call-graph* duplication. The result is a program which obeys the *strict* requirement, and can thus be compiled appropriately for the host and accelerators. Compilation may fail if duplicated functions misuse outer pointers, as in the following example:

```
void f(int * x, int * y) { x = y; ... }

int main() {
  int a = 5;
  offload {
    int b; f(&a, &b); // '&a' is outer, '&b' is not
  }
}
```

Function duplication produces a duplicate of f where the first parameter is an outer pointer. However, this duplicate is not well-typed as it makes an assignment between an outer pointer and a non-outer pointer:

```
offload void f(outer int * x, int * y)
{  x = y; ... } // type error!  'x' is outer, 'y' is not
```

If outer pointers are used correctly then, in the absence of other general errors, all duplicated functions can be compiled for the accelerators, with data movement code generated corresponding to accesses via outer pointers.

Note that a function is only duplicated with a given signature at most once, meaning that call-graph duplication works in the presence of recursion. Also note that duplication is performed on demand: although a function with n pointer parameters has 2^n possible duplicates, only those actually required by an application will be generated. The above discussion explains how call-graph duplication works for functions; the approach easily extends to instance methods with a **`this`** pointer. In particular, the code of Listing 1 can be compiled with respect to the class definition of Listing 2; the duplicated methods of Listing 3 are generated automatically.

We have presented call-graph duplication as a source-to-source translation, followed by regular compilation. In a practical implementation the technique would most likely be implemented at the call-graph level – this is the case with our implementation (§6). In particular, the programmer never needs to see the duplicated functions generated by the compiler.

4.3 Offload Functions Are Still Useful

If a function should behave differently on the accelerator with a particular configuration of outer pointers, the required variant can be explicitly overloaded using the **`offload`** keyword. Suppose the `getSecret` method of §4.1 should return a pre-defined error constant when called on an outer **`this`** pointer with an outer pointer parameter. This can be specified by adding an offload version of `getSecret` to the `secretKeeper` class of Listing 1:

```
offload int getSecret(outer int * p) outer { return ERR; }
```

This version of `getSecret` will be called whenever the method is invoked on an outer object with an outer pointer parameter; otherwise call-graph duplication will be used to compile the standard version of the method appropriately.

A common use of offload functions is to allow specialised versions of performance-critical functions to be tuned in an accelerator-specific manner, *e.g.* to exploit accelerator features such as SIMD instructions.

5 Issues Raised by Call-Graph Duplication

While call-graph duplication is conceptually simple, its implementation is challenging in the presence of the full complexity of C++. We discuss the way type inference can increase the extent to which call-graph duplication can be automatically applied (§5.1) and our solutions to the issues raised by function pointers and virtual methods (§5.2), and multiple compilation units (§5.3).

5.1 Type Inference for Outer Pointers

The driving factor in the design of Offload is the extent to which existing code can be offloaded to an accelerator without modification. Disallowing assignments between inner and outer pointers in the type system provides a useful degree of type-checking across the host/accelerator boundary. However, when applying call-graph duplication to large examples, it is convenient to design the type system so that the **outer** qualifier is automatically applied in two circumstances:

- When a pointer variable p is initialised upon declaration to an outer pointer, p is given the **outer** qualifier
- If a cast is applied to an outer pointer then the destination type in the cast is automatically given the **outer** qualifier

We present two small examples to illustrate why these methods of inferring outer pointers are useful. The following example finds the smallest element in a list of non-negative integers, where the list is terminated by the value -1:

```
int findMin(int * intList) {
  int result = *intList;
  for(int * p = intList+1; *p != -1; p++)
    if( *p < result ) result = *p;
  return result;
}

int arrayOfIntegers[100] = { ... };

offload { int smallest = findMin(arrayOfIntegers); ... }
```

Because findMin is invoked with the outer pointer arrayOfIntegers, the compiler will attempt to compile a version of findMin which accepts an outer pointer. Without type inference, the compiler would reject the input program for attempting to assign an outer pointer intList+1 to an inner pointer p and the call-graph duplication attempt would fail. With type inference, the initialisation of p to an outer pointer means that p is given the **outer** qualifier implicitly.

The following function, which returns the floating point number corresponding to a machine word given by a pointer, illustrates type inference with casts:

```
float reinterpretInt(int * i) { return *((float *)i); }
```

Without type inference, if `reinterpretInt` is called from an offload scope with an outer pointer, the program would be rejected for attempting to cast an outer pointer into an inner pointer. Automatically adding the **`outer`** qualifier to the cast means that the code compiles un-problematically.

Inference of outer pointers minimizes the extent to which the **`outer`** keyword propagates throughout a large base of source code; in many practical examples, code can be enclosed in an offload block with no **`outer`** annotations whatsoever.

5.2 Function Pointers and Virtual Methods

Consider the following type definition for functions which accept an integer argument and return no value:

```
typedef void (* int_to_void) (int);
```

Assuming multiple functions have been declared with this type, consider a function pointer variable in host memory, followed by an offload block which makes a call via the function pointer:

```
int_to_void f_ptr;
...
offload { f_ptr(25); }
```

The problem is that, assuming `f_ptr` has been initialised to a valid address, the call via `f_ptr` invokes *some* function matching the function type `int_to_void`, but we do not know which one until run-time. For the call to succeed, it is necessary for a version of the function to which `f_ptr` is assigned to have been compiled for the accelerator, and loaded into local store. A similar problem applies when virtual methods are invoked from an offload scope.

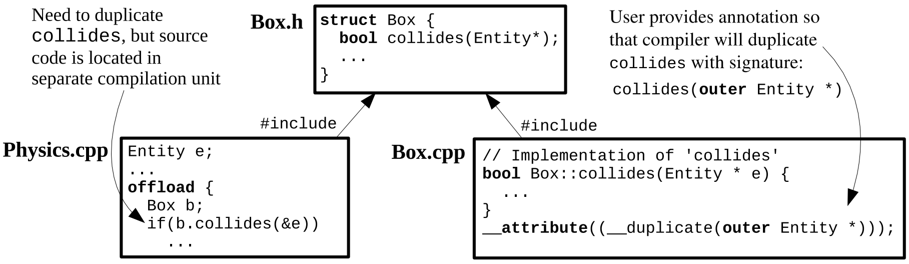
In general, statically determining the precise set of functions to which a given function pointer may refer is intractable. A safe over-approximation would be to compile *all* functions matching the `int_to_void` signature for the accelerator. This would, however, significantly increase compile time and accelerator code size.

Our solution is to use *function domains* – annotations to an offload block listing the names of functions that the block may invoke via function pointers or virtual calls. A domain for an offload block may be specified immediately following the **`offload`** keyword, as shown in the grammar of §3.1.

Function domains are implemented on the accelerator by a lookup table. The value of the function pointer is used to obtain the address of the corresponding duplicated routine on the accelerator, which is then invoked in place of the host routine whose address was taken by the function pointer. An attempt to invoke a function not specified in the domain results in a run-time error and diagnostic message. There is scope for extending the compiler with heuristics to deduce domains automatically in many practical cases.

The following games-related example (derived from industrial source code) uses an array of function pointers for collision response between game entities, and illustrates that domains occur naturally in practical examples:

```
typedef void (* collisionFunction_t) (Entity *, Entity *);
collisionFunction_t collisionFunctions[3][3] =
```


Need to duplicate collides, but source code is located in separate compilation unit
Box.h
User provides annotation so that compiler will duplicate collides with signature:
collides(outer Entity *)
#include
#include
Physics.cpp
Box.cpp

```
struct Box {
  bool collides(Entity*);
  ...
}
```

```
Entity e;
...
offload {
  Box b;
  if(b.collides(&e))
    ...
```

```
// Implementation of 'collides'
bool Box::collides(Entity * e) {
  ...
}
__attribute((__duplicate(outer Entity *)));
```

Fig. 2. Example of call-graph duplication over multiple compilation units

```
  { fix_fix, fix_mov, ..., dead_dead }; // 2d function table
...
// domain annotation on offload block
offload [ fix_fix, fix_mov, ..., dead_dead ] {
  for(...i, j...)
    // apply appropriate function according to status
    collisionFunctions [status[i]] [status[j]] (...);
}
```

Each entity has a status: `fix`, `mov` or `dead`, for fixed, moving or dead entities respectively. The array `collisionFunctions` provides a collision response function for each combination of object statuses, *e.g.* function `fix_mov` is invoked for collision response between a fixed and a moving object. By equipping the offload block with a named list of collision functions, the call via the `collisionFunctions` array will succeed.

5.3 Multiple Compilation Units

Automatic call-graph duplication depends on the compiler having access to the source code for all invoked functions. For large applications this is not the case, as source code is split into multiple files for separate compilation. Suppose source code is not available for method `collides`, called from an offload scope in compilation unit `Physics.cpp`. The compiler cannot perform call-graph duplication and simply generates code to call `collides` with the pointer signature required by the call site. Suppose `collides` is implemented in compilation unit `Box.cpp`. The programmer must mark the implementation with a *duplication obligation*, so that when `Box.cpp` is processed the compiler will duplicate the required version of `collides`, even if `collides` is *not* called from an offload scope in `Box.cpp`. This is illustrated in Fig. 2.

Annotating source code with duplication obligations is not too onerous – if the `collides` method of `Box` calls other functions that *are* defined in `Box.cpp` then since `Box::collides` is marked for duplication, these functions will be automatically duplicated appropriately. Thus, programmer annotations for duplication obligations are restricted to the boundaries of compilation units.

6 A Cell BE Implementation of Offload

We have implemented Offload for the Cell BE processor under Linux.[3] A C++ application with offload blocks is compiled to intermediate C program text targeting the PPE and SPE cores. A makefile and linker script are also generated; these use the PPE and SPE GCC compilers to generate a Cell Linux PPE executable with embedded SPE modules, one per offload block.

A small run-time system implements the target-specific glue code required to use an accelerator, such as handling the transfer of parameters from the host and setup of a software cache through which access to the host memory is provided. The run-time also permits offloaded code to invoke routines on the host, *e.g.* to call malloc/free on host pointers, and for mutex-based synchronization via the POSIX threads API. Our implementation includes header files with definitions of SPE-specific intrinsics, allowing their use in programs where some SPE hand-tuning is desired.

Given a multi-threaded C++ application, we propose the following method for offloading threads to run on SPEs:

1. Profile application to identify a computationally expensive host thread
2. Replace this thread with an offload block, adding outer pointer annotations, function domains and duplication obligations where necessary for correctness
3. Replace performance-critical functions with offload functions, specialised with SPE-specific optimizations where necessary for performance
4. Repeat this process until all appropriate host threads are offloaded

By (2) we mean that a call to create a thread running function `f` should be replaced with an offload block which calls `f`. It is straightforward to define thread creation macros that allow an application to use either POSIX or offload threads, depending on the available support for a particular platform.

Basic offloading achieves the goal of getting code to run on SPEs, freeing the PPE to perform other useful work. This can provide a performance benefit even if performance of offloaded code is non-optimal. To achieve higher performance, it may be necessary to write offload versions of performance-critical functions, hand-optimized for SPEs. The main barrier to performance for the Cell BE is data movement. The Offload system includes a set of header files to optimize access of contiguous data in host memory. These header files define templated iterator classes, to be used in offload functions, that optimize reading/writing of data from/to host memory using double-buffering. The compiler generates advice messages, based on static analysis, to guide the programmer towards refactorings to improve performance.

7 Experimental Results

We have developed a set of examples to investigate the performance improvement over serial code which can be achieved using Offload for the Cell BE processor, and the ease with which the language extensions can be applied:

[3] In addition, an implementation of Offload for Sony PlayStation 3 consoles is available to SCE-licensed game developers.

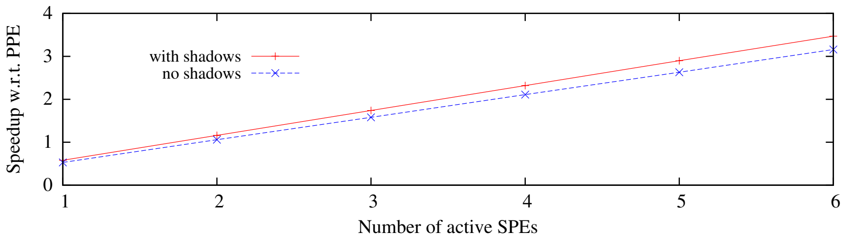


Fig. 3. Scaling of SphereFlake offload across multiple SPEs

- A Mandelbrot fractal generator, generating a 640×480 pixel image
- SphereFlake: a ray tracer generating a 1024×1024 pixel image [2]
- A set of five image processing filters operating on a 512×512 pixel image, performing: embossing, noise reduction, sharpening, Laplacian edge detection, and greyscale conversion

Experiments are performed on a Sony PlayStation 3 console, for which 6 SPEs are available to the programmer. We compare the performance of the computations as follows. The original code executing on a single hardware thread of the Cell PPE is used as a baseline against which to compare successive versions where computation is offloaded to between 1 and 6 SPEs. For a configuration with N SPEs, the benchmarks are multi-threaded to spawn N offload threads, each of which computes $1/N$ of the problem size.

Mandelbrot. The generator computes the Mandelbrot set value for each pixel using scalar operations. Offloading this sequential computation on to a single SPE yields a $1.6\times$ performance increase over the PPE baseline. When 6 SPEs are utilised the performance increase is $13.5\times$. By hand-optimizing the offloaded code to buffer output pixels, writing back to the host a line at a time, a performance increase of $14\times$ over the serial baseline is achieved. This modest improvement over the non-optimized case indicates the program is compute bound.

SphereFlake. SphereFlake [2] is a fractal ray tracer for generating and ray tracing a procedural model of spheres. We have applied Offload to this third party application, offloading parts of the ray tracer to run in parallel across Cell SPEs. Thanks to automatic call-graph duplication, it was possible to use the core of the ray tracer without modification. We applied some modest refactorings to improve performance, ensuring that procedural model generation is performed in SPE local store, and buffering output pixel values in local store, to be written to main memory by DMA a row at a time.

We benchmark the ray tracer with and without support for shadows. Performance scales linearly with the number of SPEs used, as shown in Fig. 3. With one SPE, performance is around $0.5\times$ that of the PPE baseline; with two SPEs, performance slightly exceeds that of the PPE. Maximum performance is achieved with six SPEs, with speedups of $3.47\times$ and $3.16\times$ PPE performance with and without shadows respectively.

Image processing filters. Fig. 4 shows the performance of our image processing filters, offloaded using either a single SPE, or all six available SPEs. For each offloaded

Filter	Emboss no manual opt.	Noise no manual opt.	Sharpen buffered output	Laplacian buffered I/O	Greyscale fully optimized
Speedup: 1 SPE	$0.6\times$	$0.85\times$	$0.76\times$	$3.13\times$	$3.06\times$
Speedup: 6 SPEs	$3.27\times$	$2.76\times$	$2.96\times$	$6.51\times$	$3.44\times$

Fig. 4. Speedups for offloaded image processing filters, with one and six SPEs

benchmark, the figure indicates whether we have performed no additional manual optimization, optimizations to buffer output, input and output, or extensive manual optimizations, including vectorization.

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

Performance scales only modestly for the greyscale benchmark as SPEs are added, due to the lightweight nature of the computation. This is an example where offloading a *single* thread to an accelerator can provide a useful speedup.

Discussion of performance. Our investigation of the performance of offloaded code identifies three categories of benchmarks, distinguished by the ease with which performance increases are obtained, and the steps required to achieve such increases. Computationally intensive algorithms, such as Mandelbrot and SphereFlake, result in increased performance by offloading, requiring little programmer effort, as execution times are dominated by computation rather than data access. Less straightforward are applications such as our image filter examples, that perform relatively little computation per data item on a large volume of data, but access contiguous data using a regular stride. In this case, basic offloading typically results in a performance decrease, which can easily be ameliorated using simple DMA operations which can be hidden in templated classes. A third category of applications, for which we do not present results, access large volumes of input data in an unpredictable manner, or in a manner not amenable to efficient DMA operations. This type of application may require significant restructuring for high performance to be achieved.

8 Related Work

Programming models. Of the recent wealth of programming models for multicore architectures, closest to Offload are Sequoia [3] and CellSs [4]. The Sequoia language abstracts both parallelism and communication through side-effect free methods known as tasks, which are distributed through a tree of system memory modules. When a task is called on a node, input data is copied to the node's address space from the parent's address space, and output is copied back on task completion. This provides a clean way to distribute algorithms that operate on regularly-structured data across heterogeneous multicore processors. However, tasks to be accelerated must be re-written using the bespoke Sequoia language, and the approach is only applicable when the data required by a task (its *working set*) is known statically. The latter requirement has its advantages, allowing aggressive data movement optimizations. CellSs is similar to Sequoia, involving the identification of tasks to be distributed across SPEs, and requiring the working set for a task to be specified upfront.

The idea of optimizing data movement based on regularly structured data is the basis for stream programming languages such as StreamIt [5] and Brook [6], and more recently HMPP [7] and OpenCL [8]. These models encourage a style of programming where operations are described as kernels – special functions operating on streams of regularly structured data – and are particularly suitable for programming compute devices such as GPUs. As with Sequoia and CellSs, exploiting regularity and restricting language features allows effective data movement optimizations. The drawback is that these languages are only suitable for accelerating special-purpose kernels that are feasible to re-write in a bespoke language. In contrast, Offload allows portions of general C++ code to be offloaded to accelerator cores in a heterogeneous system with few modifications. The flexible notion of outer pointers does not place restrictions on the working set of an offload thread. The price for this flexibility is that it is difficult to automatically optimize data movement for offload threads.

Call-graph duplication. Our notion of call-graph duplication is related to function cloning [9], used by modern optimizing compilers for inter-procedural constant propagation [10], alignment propagation [11], and optimization of procedures with optional parameters [12]. Automatic call-graph duplication applies function cloning in a novel setting, to handle multiple memory spaces in heterogeneous multicore systems. Call-graph duplication is related to C++ template instantiation, and faces some of the same challenges. Call-graph duplication across compilation units (§5.3) is similar to template instantiation across compilation units, which is allowed in the C++ standard via the **`export`** keyword, but supported by very few compilers.

Memory-space qualifiers. The idea of using qualifiers to distinguish between shared and private memory originated in SIMD array languages [13], and is used in PGAS languages such as Titanium [14], Co-array Fortran and Unified Parallel C [15]. Similar storage qualifiers are used by CUDA and OpenCL to specify data locations in accelerators with hierarchical memory.

9 Conclusions and Future Work

Our experimental evaluation with a Cell BE implementation of Offload give a promising indication that the techniques presented in this paper allow performance benefits of accelerator cores to be realised with relative ease, requiring few modifications to existing code bases.

While Offload is more flexible than alternative approaches for programming heterogeneous systems, this flexibility means data movement for offload threads is hard to optimize. We plan to extend Offload with facilities for annotating an offload block with information about expected data usage, which the compiler can use to apply more aggressive optimizations.

Call-graph duplication can potentially lead to a significant blow-up in code size, if a function with several pointer arguments is called with many configurations of local/outer pointers. This can be problematic when accelerator memory is limited. We plan to investigate tool support for providing feedback as to the extent to which call-graph duplication is required, and opportunities for reducing duplication.

Acknowledgements

We are grateful to Anton Lokhmotov, Paul Keir, Philipp Rümmer, and the anonymous reviewers, for their insightful comments on an earlier draft of this work. Alastair F. Donaldson is supported by EPSRC grant EP/G051100.

References

1. Hofstee, H.P.: Power efficient processor architecture and the Cell processor. In: HPCA, pp. 258–262. IEEE, Los Alamitos (2005)
2. Hoines, E.: A proposal for standard graphics environments. IEEE Comput. Graph. Appl. 7, 3–5 (1987)
3. Fatahalian, K., Horn, D.R., Knight, T.J., Leem, L., Houston, M., Park, J.Y., Erez, M., Ren, M., Aiken, A., Dally, W.J., Hanrahan, P.: Sequoia: programming the memory hierarchy. In: Supercomputing, p. 83. ACM, New York (2006)
4. Bellens, P., Perez, J.M., Badia, R.M., Labarta, J.: CellSs: a programming model for the Cell BE architecture. In: Supercomputing, p. 86. ACM, New York (2006)
5. Thies, W., Karczmarek, M., Amarasinghe, S.P.: Streamit: A language for streaming applications. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 179–196. Springer, Heidelberg (2002)
6. Buck, I.: Brook specification v0.2., `http://merrimac.stanford.edu/brook/`
7. CAPS Enterprise: HMPP, `http://www.caps-entreprise.com/hmpp.html`
8. Khronos Group: The OpenCL specification, `http://www.khronos.org/opencl`
9. Cooper, K.D., Hall, M.W., Kennedy, K.: A methodology for procedure cloning. Comput. Lang. 19, 105–117 (1993)
10. Metzger, R., Stroud, S.: Interprocedural constant propagation: An empirical study. LOPLAS 2, 213–232 (1993)
11. Bik, A.J.C., Kreitzer, D.L., Tian, X.: A case study on compiler optimizations for the Intel Core 2 Duo processor. International Journal of Parallel Programming 36, 571–591 (2008)
12. Das, D.: Optimizing subroutines with optional parameters in F90 via function cloning. SIGPLAN Notices 41, 21–28 (2006)
13. Lokhmotov, A., Gaster, B.R., Mycroft, A., Hickey, N., Stuttard, D.: Revisiting SIMD programming. In: LCPC, Revised Selected Papers, pp. 32–46. Springer, Heidelberg (2008)
14. Yelick, K.A., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P.N., Graham, S.L., Gay, D., Colella, P., Aiken, A.: Titanium: A high-performance Java dialect. Concurrency - Practice and Experience 10, 825–836 (1998)
15. Coarfa, C., Dotsenko, Y., Mellor-Crummey, J.M., Cantonnet, F., El-Ghazawi, T.A., Mohanti, A., Yao, Y., Chavarría-Miranda, D.G.: An evaluation of global address space languages: Co-array Fortran and Unified Parallel C. In: PPOPP, pp. 36–47. ACM, New York (2005)