

The FPGA High-Performance Computing Alliance Parallel Toolkit

Rob Baxter¹, Stephen Booth, Mark Bull, Geoff Cawood, James Perry, Mark Parsons, Alan Simpson, Arthur Trew
EPCC and FHPCA

Andrew McCormick, Graham Smart, Ronnie Smart
Alpha Data ltd and FHPCA

Allan Cantele, Richard Chamberlain, Gildas Genest
Nallatech ltd and FHPCA

¹ communicating author: r.baxter@epcc.ed.ac.uk; 0131 651 3579;
University of Edinburgh, James Clerk Maxwell Building, King's Buildings, Edinburgh EH9 3JZ

Abstract

We describe the FPGA HPC Alliance's Parallel Toolkit (PTK), an initial step towards the standardization of high-level configuration and APIs for high-performance reconfigurable computing (HPRC). We discuss the motivation and challenges of reaping the performance benefits of FPGAs for memory-bound HPC codes and describe the approach we have taken on the FHPCA supercomputer Maxwell.

1. Introduction

Perhaps the number one challenge to improving the uptake of FPGAs in high-performance computing (HPC) is programmability.

The FPGA High Performance Computing Alliance (FHPCA) [1] is focused on the development of computing solutions using FPGAs to deliver new levels of performance. It will enable a range of applications to be able to exploit the benefits that such an approach will deliver.

This will be achieved through the implementation of a large scale demonstrator or 'supercomputer' – Maxwell – and supported by a structured programme of knowledge diffusion designed to raise industry awareness, interest and create commercial advantage.

The alliance partners are Algotronix, Alpha Data, EPCC, the Institute for System Level Integration, Nallatech and Xilinx.

The project to build Maxwell and port three key demonstrator applications to it is described in [2]; this paper focuses on the programming model and generic software approach taken in the project. One of the primary goals of the project was to build a software layer – the Parallel Toolkit (PTK) – which would, as far as possible, abstract away the details of the underlying

FPGA hardware *from an application perspective*. This was deliberately driven as a top-down approach, taking the view of the HPC code owner as a starting point and attempting to 'push down' as far as possible a common interface layer.

Maintaining the portability and maintainability of HPC codes on new architectures is these days essential. Many years of effort have been invested since the early 1990s on standardizing HPC codes to run cleanly and portably across a wide range of parallel hardware. Standards such as the Message Passing Interface [3] and OpenMP [4] are used almost exclusively; gone are the days of vendor-specific codes and proprietary languages.

High-level FPGA programming faces the same challenges now as parallel computing did 15 years ago. Addressing these challenges requires a consensus among tool providers and a standardization activity by the users of high-performance reconfigurable computing – a process organizations like OpenFPGA [5] hope to catalyse.

The PTK is a potential step along the road towards standardizing both high-level APIs and job and machine configuration methods for HPRC. It is a set of practices and infrastructure intended to address key HPRC acceleration issues such as: how to associate processes with FPGA resources; how to associate FPGAs with bitstreams; how to manage contention for FPGA resources within a process; and how to managing code dependencies to facilitate re-use. To these ends the PTK comprises a library of C++ classes providing abstract interfaces to FPGA hardware components; classes providing standard ways to configure arbitrary FPGA hardware; and a standard way of launching parallel FPGA jobs.

This paper is structured as follows. Section 2 describes the starting point for hardware acceleration of a standard HPC code and describes the basic problem the PTK was set to solve. Section 3 further motivates the

PTK and describes the various design decisions that went into it. Section 4 describes the PTK's architectural levels and describes by example how the architecture aims to solve the basic problem as posed. Section 5 offers more details on the internals of the PTK, and Section 6 describes the challenge of hardware configuration. Finally Sections 7 and 8 describe the current limitations of the PTK and offer thoughts for further work.

2. Basic acceleration strategy

Our approach to designing the PTK began with the assumed starting position of an existing parallel application running entirely in software over multiple processes (and typically this means multiple processors). We assume a fairly standard single program-multiple data (SPMD) model, with inter-process communication handled by the standard message-passing interface (MPI) library. Our other basic assumption is that the fully parallelized code is not delivering the necessary performance.

The PTK was also designed as the acceleration and programming approach for the FHPCA Supercomputer *Maxwell*. Maxwell is described in detail elsewhere but logically it comprises a number of equivalent 'nodes'. A node is defined as a software process running on a host CPU, together with some FPGA acceleration hardware.

A node's application process runs on a host CPU and communicates with an 'accelerator' via the PTK. The physical contents of an accelerator are deliberately kept fluid to avoid constraining the system unnecessarily. The number of FPGAs per node or even FPGA boards per node may vary depending on the application. The 'soft' configuration of an accelerator in terms of the hardware functions programmed into its FPGAs will also vary, as different applications will require different functionality.

To accelerate an application, its 'hotspot' function F is firstly identified. A corresponding hardware function F' is then designed to perform the same task (but at a higher speed). A bitstream representing F' is programmed into the accelerator, and F is then accelerated by replacing its innards with a call out to F' .

At runtime, F typically copies the relevant input data to a memory component on the accelerator before invoking F' . F' then processes the data directly from this local memory. Once F' signals its completion, F copies any output data back to the host.

The execution of F' may involve external communication with neighbouring nodes to implement the message-passing model of parallel computation commonly used in HPC applications.

3. Motivation and design

Our primary aim in building the PTK was to provide a way to transform a starting code into a hardware-accelerated parallel application. 'Hotspot' logic would be migrated from software to FPGA hardware, and custom configuration of the FPGA would deliver the necessary performance. Other key design requirements were driven by our experiences at EPCC in parallel computing and its standardisation.

3.1 'Clean' hardware interfacing

Hardware abstraction was a key driver. The PTK allows the same application code to work with FPGA boards from multiple vendors (notably Alpha Data and Nallatech).

The PTK also provides a clean mechanism for client application code to access hardware resources without introducing undesirable dependencies, or necessitating client code changes whenever the hardware is revised.

The call out to a hardware function could potentially occur anywhere in an application. For example, suppose the call needs to happen deep in a reusable module, eg. some matrix multiplication code. Inevitably a dependency will be introduced from the matrix code to the acceleration hardware, but for reusability that dependency should be made as 'narrow' as possible. It should be a dependency only on the particular multiplier component required, not on the wider accelerator. This should allow the same matrix code to work with any accelerator that offers the appropriate multiplier component.

The PTK also had to ensure that different parts of an application do not attempt to utilise the same component at the same time.

3.2 Parallel functions

One feature of FPGAs that allow them to deliver significant performance advantage over CPUs is the capacity for naturally functional and pipeline parallelism within each device. The PTK therefore had to allow multiple hardware functions to be executed in parallel, and allow the application to continue processing while a hardware function is executing.

For example, it may be the case that an application process can do useful work while waiting for a hardware function to complete; so the invocation of a hardware function should not cause the application to be blocked.

3.3 Copying efficiency & memory flexibility

The key to our acceleration strategy is one of copying efficiency. The PTK was thus designed to provide scope to avoid unnecessary data-copying between host and FPGA memory.

For example, suppose an application involves two accelerated functions, F and G. If G is called soon after F and operates on the same data then it is wasteful for F to copy the data back to the host, only for G to copy it onto the accelerator again immediately after. So the PTK supports the concept of data that is resident on an accelerator and has a lifetime longer than that of a single hardware function call.

FPGA memories physically have independent address buses. They are only logically linked when accessed from the PCI interface firmware. The parallel components in the FPGA will access them in as parallel a manner as possible.

Typically FPGA memory is split into multiple parallel independently addressable banks. These banks will have independent functionality specific to the parallel accelerator components in the FPGA which can access each bank. The PTK supports a software model of these independent banks, to simplify the transfer and conversion of data between the host and the FPGA.

3.4 Potential for 'intelligent' tools

A final, but very important, consideration was to ensure that PTK accelerators provide scope for high-level toolkit functions that can adapt to the quantity of hardware resources available. Ultimately this should offer further decoupling between the application software and the FPGA hardware.

4. PTK architecture

This section describes the overall architecture of the PTK as implemented on the FHPA Supercomputer *Maxwell*. The PTK is written as a series of C++ libraries and interfaces; thus this section uses the language of object orientation in general and C++ in particular.

4.1 Software layers

There is a loose layering of classes in the PTK, as shown in Figure 1.

Each layer can be thought of as a client of the layer below; the application is the ultimate client of the PTK, and the PTK is a client of the vendor C APIs.

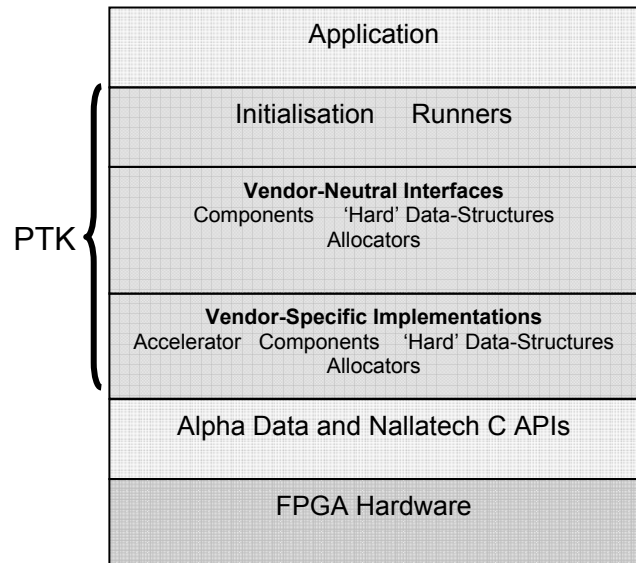


Figure 1. PTK layered architecture

4.2 Acceleration using the PTK

So, how does the PTK facilitate the basic acceleration strategy noted above?

Suppose that some algorithm F is to be accelerated. There may be many ways to implement F in FPGA hardware and different vendors' implementations may be radically different depending on the capabilities of their particular hardware and preferred design tools etc.

Rather than coercing vendors into producing similar implementations, the approach taken in the PTK is to define the top level interfaces only and give vendors complete freedom over how they choose to implement those interfaces.

The interfaces are defined using abstract superclasses in C++. Concrete realizations of these abstract classes are then defined as required in hardware-specific ways.

Three categories of PTK object may be involved in modelling F: Components; Hard Data-Structures; and Allocators.

A component is essentially a computing device performing some calculation. A hard data-structure is data residing in FPGA hardware (as opposed to host memory), and allocators provide controlled access to both these types of resources.

For example, in a simple image processing application, F is an image processing algorithm and it is modelled in terms of these abstract objects:

- Jacobi – a Component that processes data in an ImageHard.

- ImageHard – a Hard Data-Structure representing image data.
- JacobiAllocator – an Allocator for accessing Jacobi components.
- ImageHardAllocator – an Allocator for accessing ImageHard data structures.

A specific implementation must provide suitable concrete subclasses and a new Accelerator object that will manage the initialisation of these allocators and configure the FPGA hardware (bitstreams etc.). At runtime, the sequence of events in a given process of the accelerated image processing application would be as follows:

1. The PTK is initialised (for this node):
 - a. The desired Accelerator for this implementation is created.
 - b. The FPGA hardware is prepared.
2. When execution reaches F:
 - a. An ImageHard of appropriate size for the input data is requested from ImageHardAllocator.
 - b. The input data is copied into the resulting ImageHard.
 - c. A Jacobi suitable for use with the ImageHard is requested from JacobiAllocator.
 - d. The resulting Jacobi component is invoked on the ImageHard.
 - e. On completion, output data is copied out of the ImageHard.
 - f. The Jacobi component is released.
 - g. The ImageHard is released.
 - h. The output data is returned.
3. The PTK is shutdown.

Note that these interactions between the application and the PTK are completely independent of the underlying implementation, ie. none of this needs to change should a different implementation be used.

In practice, steps (a) to (h) are performed by a fourth type of object, a ‘Runner’ in the PTK (JacobiRunner in this case), to help minimise complexity in the application code.

5. PTK internals

The following sections describe the main classes involved in the PTK.

5.1 Hard data-structures

As an example, suppose that the data structure used in the (unaccelerated) image processing application is of class Image. A new ImageHard class would be

introduced to represent an image when stored in the FPGA hardware.

ImageHard would be an abstract superclass declaring methods to copy-in and copy-out image data that is passed as an instance of class Image eg.

```
void ImageHard::receiveFrom(const Image* image);
void ImageHard::sendTo(Image* image) const;
```

Depending on the nature of Image, it might not be possible to fill an existing Image with data (as the sendTo method here implies). It might be necessary to create and return a new Image instead eg.

```
Image* ImageHard::getNewImage() const;
```

ImageHard would be subclassed for a specific implementation eg. ImageHardAd1 for the first Alpha Data implementation.

Note that to prevent an undesirable dependency from the PTK to the application (as a result of ImageHard methods referencing Image), the Image class is deemed to come not from the application, but from a separate independent library. The application might need some refactoring accordingly.

5.2 Components

An abstract superclass is created for each type of component (eg. Jacobi) and a concrete subclass is created in each implementation (eg. JacobiAd1). The methods defined on the superclass should allow client code to achieve the following:

- pass parameters in at the beginning;
- receive data out at the end;
- start the component running (non-blocking);
- wait for it to finish running.

For example, the interface to the abstract Jacobi superclass might be as follows:

```
void Jacobi::setImage(ImageHard* imageHard);
void Jacobi::setMaxIter(int iterations);
void Jacobi::setTolerance(float tolerance);
void Jacobi::startNonBlock();
void Jacobi::waitForCompletion();
```

The purpose of the setImage(...) method is to tell the Jacobi where the image data lives (on the FPGA hardware). However, this information may not be accessible via the ImageHard interface as the nature of this information is specific to the implementation. For one implementation the information might be a single address. For a different implementation that splits the data over multiple memory banks, the information might be a number of addresses on a number of memories.

Consequently, the `setImage(...)` method on an implementation-specific subclass may need to downcast the `ImageHard` to the specific data class expected eg.

```
void JacobiAd1::setImage(ImageHard* imageHard) {
    // AD1-specific Jacobi only meant for use with
    // AD-specific image.
    ImageHardAd* imageAd =
        dynamic_cast<ImageHardAd*>(imageHard);

    if (imageAd == NULL) {
        throw runtime_error("AD-specific Jacobi was
            passed a non-AD image");
    }

    int address = imageAd->getAddress();
    // Then pass the address to the Jacobi
    // hardware so it knows where to
    // start processing...
    ...
}
```

An alternative solution would be to define the external interface to the Jacobi functionality as methods on the `ImageHard` class. This would have the advantage of simplifying the top-level interface and avoiding downcasting, but at the expense of flexibility in invoking a component.

5.3 Allocators

For each type of component or hard data-structure, a corresponding Allocator class is required that can serve up those resources to clients. For example, in the image processing application there is a `JacobiAllocator` for serving up Jacobi components and an `ImageHardAllocator` for serving up `ImageHard` objects.

These allocator classes are also made abstract since the actual type of resource they should return depends on the implementation. The interface of `ImageHardAllocator` is straightforward:

```
virtual ImageHard*
    requestImageHard(int sizeX, int sizeY) = 0;
virtual void
    releaseImageHard(ImageHard* imageHard) = 0;
```

As the names suggest, these methods allow clients to request `ImageHards` of a given size, and release them again when their use is over.

A concrete implementation of `ImageHardAllocator` would override these methods to return an object of the appropriate type for that implementation. Eg. `ImageHardAllocatorNt1` might return an object of type `ImageHardNt1`.

As well as defining an interface for subclasses to implement, Allocator superclasses are also ‘singletons’ in that they provide a globally-accessible home for whichever concrete allocator is used in an

implementation. For example, `ImageHardAllocator` has the following static methods for global access.

```
static ImageHardAllocator* getInstance();
static void
    setInstance(ImageHardAllocator* allocator);
static void clearInstance();
```

Assuming that `setInstance(...)` has been called at initialisation time, client code can access the current allocator as follows:

```
ImageHard* imageHard =
    ImageHardAllocator::getInstance()->
    requestImageHard(sizeX, sizeY);
```

The client code here is completely independent of the actual implementation being used, which means that no changes are necessary when using a different implementation. Also, the client code depends only on the resources that it absolutely needs.

It is the responsibility of the concrete implementation to make sure that physical constraints are not violated. For example, if there is only one Jacobi component in an implementation then the corresponding allocator must refuse any request which comes in while the sole Jacobi is in use.

Some allocators require information about other resources before being able to make a decision. For example, suppose there are multiple Jacobi components available. If an `ImageHard` has already been allocated, the implementation-specific `JacobiAllocator` will need to know which memory the `ImageHard` occupies in order to select a Jacobi which is connected to it.

5.4 Accelerators

An Accelerator object models all the relevant acceleration hardware for a given implementation of a given algorithm for a given node.

The main role of an Accelerator is configuration. It must configure its associated FPGA hardware with the appropriate bitstream(s), and initialise the relevant allocators so that they are ready to serve up the resources that have been designed for use with this implementation. This happens in the `configure()` method.

It is usually convenient to have a dedicated Accelerator subclass for each implementation of an algorithm. So if Nallatech produce two implementations ‘Nt1’ and ‘Nt2’ of the image processing application then there might be two accelerator classes `AcceleratorImgProcNt1` and `AcceleratorImgProcNt2` (ideally with more meaningful names).

If implementation Nt1 uses a particular `ImageHard` class (`ImageHardNt1`, say) then a corresponding allocator would be set up in `AcceleratorImgProcNt1` eg.

```
void AcceleratorImgProcNt1::configure() {
    ImageHardAlloc_ = new
        ImageHardAllocatorNt1();
    ImageHardAllocator::setInstance
        (imageHardAlloc_);
    ...
}
```

Any subsequent references from the client code to `ImageHardAllocator::getInstance()` will produce the special Nt1-specific instance created here.

In their destructors, Accelerator classes should delete any objects that they created from the heap, and reset any allocator singletons that were set up eg.

```
AcceleratorImgProcNt1::~AcceleratorImgProcNt1() {
    if (ImageHardAllocator::getInstance() ==
        imageHardAlloc_) {
        ImageHardAllocator::clearInstance();
    }
    delete imageHardAlloc_;
    ...
}
```

5.5 Handling Output Data

In the example interface of the Jacobi component it was assumed that the hardware operates ‘in-place’, ie. the output image replaces the input image in FPGA memory. The client code then extracts the output image by copying the data back out of the original `ImageHard` eg.

```
ImageHard* input =
    ImageHardAllocator::getInstance()->
        requestImageHard(image->size());

// Send the image data to the FPGA.
input->receiveFrom(image);

Jacobi* jacobi =
    JacobiAllocator::getInstance()->
        requestJacobiFor(input);
jacobi->setImage(input);
jacobi->setMaxIter(100);
jacobi->setTolerance(0.01);
jacobi->startNonBlock();
jacobi->waitForCompletion();

// Copy data back via the original ImageHard.
Image* result = input->getNewImage();

// Release resources.
ImageHardAllocator::getInstance()->
    releaseImageHard(input);
JacobiAllocator::getInstance()->
    releaseJacobi(jacobi);

return result;
```

However, for some components this will not be the case, and they will generate output data elsewhere eg. in a different part of the memory, or in a different memory altogether. This raises the question of how the client code should access output data. One option is to provide a method on `Jacobi` that returns an `Image` eg.

```
Image* Jacobi::getOutputImage();
```

However, this is not ideal if the output data is to be used as the input data for another component, since the information about where the output data currently lives on the FPGA hardware will be lost.

To solve this, there could be a method on `Jacobi` for returning an `ImageHard` instead of an `Image`. Client code could then either extract an `Image` from the `ImageHard`, or use the `ImageHard` as an input to another component eg.

```
ImageHard* Jacobi::getOutputImageHard();
```

The snag with this approach is that the client code does not know whether this output `ImageHard` should be released via the `ImageHardAllocator` or not. Ordinarily, the client code uses the `ImageHardAllocator` to create `ImageHards` and is therefore duty-bound to use the `ImageHardAllocator` to release them again. But this new method would be an alternative source of `ImageHards`, and there would have to be a convention about how to dispose of them.

A third and perhaps cleaner solution is for the client to use the `ImageHardAllocator` to allocate both the input and output data at the start and pass them both into the `Jacobi`. It is then clear that the client code must use the `ImageHardAllocator` to release both the input and output data at the end. The additional challenge in this case is knowing how much output data space to allocate, but this can be handled via another method on `ImageHardAllocator` eg.

```
ImageHard* input =
    ImageHardAllocator::getInstance()->
        requestImageHard(image->size());
ImageHard* output =
    ImageHardAllocator::getInstance()->
        requestJacobiOutputImageFor(input);
...

// Tell Jacobi where the input data is.
jacobi->setInputImage(input);

// Tell Jacobi where to store output data.
jacobi->setOutputImage(output);
...

jacobi->waitForCompletion();

Image* result = output->getNewImage();

// Release resources if not re-using.
ImageHardAllocator::getInstance()->
    releaseImageHard(input);
ImageHardAllocator::getInstance()->
    releaseImageHard(output);
JacobiAllocator::getInstance()->
    releaseJacobi(jacobi);

return result;
```

Here `requestJacobiOutputImageFor(...)`, the new method, has to work out how much output space is required by the Jacobi component when processing the given input image. If this is difficult to determine beforehand, it can naively allocate some arbitrarily large amount to be on the safe side. Imposing an upper bound on output size before the algorithm starts is not really a new burden as the size of the memory itself is already an upper bound.

Finally, what should happen if implementation X of Jacobi works in-place, while implementation Y creates output data elsewhere? The behaviour of these two implementations is sufficiently different that they are arguably not implementing the same functionality and should therefore be implementations of two different components eg. `JacobiInPlace`, `JacobiOutOfPlace`.

6. Machine configuration in the PTK

As described above, an Accelerator object is responsible for configuring a given node of an application. However, to do this it may require certain information such as the identities of its neighbour nodes, bitstream file names, addresses or other parameters. Also, before the node's software process has even started, MPI must be told which executable to launch for that node.

To handle these issues, a configuration file is used in combination with a launcher script. The syntax of a PTK config file is similar to a Java properties file, *vis* a sequence of `name = value` pairs.

At runtime, the config file is read into the PTK and stored within the `NodeProps` singleton which may be accessed via `NodeProps::getInstance()`. It is possible to set properties programmatically, though this will only have any tangible effect if the properties are set prior to the point at which they are queried by other parts of an implementation.

In general, properties are set in the config file and then queried (eg. by Accelerators) using the following methods on `NodeProps`:

```
string getStringProp(string key, string def="");
int   getIntProp  (string key, int   def=0);
float getFloatProp (string key, float def=0.0);
```

For example, use the following to discover the value of the `app.node.3.fpgaId` property:

```
int fpga = NodeProps::getInstance()->
  getIntProp("app.node.3.fpgaId");
```

For convenience, client code can supply a default value that should be used in the event that the property has not been set, eg.

```
int fpga = NodeProps::getInstance()->
  getIntProp("app.node.3.fpgaId",-1);
if (fpga < 0) ...
```

7. Current limitations

As it currently stands the PTK has a number of elements which assist in the manual FPGA acceleration of application codes. It solves various software side-issues (primarily software engineering issues rather than hardware acceleration problems per se) and promotes flexibility and reusability where practicable. It offers a standardized way of launching parallel FPGA jobs and configuring FPGAs with bitstreams.

However it is by no means a panacea. It does not begin to address automated FPGA acceleration, nor does it push standardized interfaces very deep into the software stack. Interfaces tend to be high-level and application-specific – in the language of HPC library interfaces, more PETSc [6] than BLAS [7].

8. Further work

The obvious route for the PTK to take is towards generalization and deeper standardization. Both these goals depend on pushing standard interfaces deeper into the software stack, and this cannot be achieved without a close dialogue with FPGA C-to-gates tools vendors.

Close dialogue is also required with existing HPC code owners to understand what generic features would be most useful, and what level of 'code mangling' they are willing to accept to reap the performance rewards offered by FPGAs.

FPGA programmability is still the major challenge for HPRC, and the PTK is only a start along the road ahead.

9. References

- [1] The FHPCA, www.fhpc.org
- [2] R.M. Baxter *et al*, "Maxwell – a 64 FPGA Supercomputer", *Proc. AHS2007 Conf.*, Second NASA/ESA Conference on Adaptive Hardware and Systems, Edinburgh, 2007.
- [3] The Message Passing Interface forum, www.mpi-forum.org
- [4] The OpenMP Architecture Review Board, www.openmp.org
- [5] OpenFPGA, www.openfpga.org
- [6] Portable, Extensible Toolkit for Scientific Computation, <http://www-unix.mcs.anl.gov/petsc/petsc-as/>
- [7] The Basic Linear Algebra Subprograms library, <http://www.netlib.org/blas/>