SHADOW3 API: The Application Programming Interface for the ray tracing code SHADOW

Niccolò Canestrari[a,b], Dimitris Karkoulis[a] and Manuel Sánchez del Río[a]

[a]European Synchrotron Radiation Facility, 6, rue Jules Horowitz, 38000 Grenoble, France
[b]Institut Néel, CNRS-UJF, 38042 Grenoble, France

ABSTRACT

We developed the third version of SHADOW, a ray tracing software widely used to design optical system in the synchrotron world. SHADOW3 is written in Fortran 2003 and follows the new computer engineering standards. The users can always execute the program in the traditional file oriented approach. Moreover, advanced users can create personalized scripts, macros and executables using the new Application Programming Interface SHADOW3-API. It also allows binding of SHADOW3 with several popular programming languages such as C, C++, python and IDL. We describe the SHADOW3 API structure, and illustrate its use with some examples.

We analyze the possibilities of running SHADOW3 in parallel machines under different environments. A version using the Open Message Parsing Interface has been implemented. A SHADOW3 postprocessor has been accelerated with the use Graphics Processing Units. This will open new possibilities to extend the already very popular ray tracing tool to applications simulating 2D and 3D experiments (like imaging, tomography)

Keywords: SHADOW3, Application Programming Interface, parallelization, ray tracing, X-ray optics, GPU, OpenMPI

1. INTRODUCTION

SHADOW3 is the new version of the popular ray-tracing software SHADOW. SHADOW,[1] developed by F.Cerrina, has been in use for three dacades and helped many scientists in designing synchrotron beamlines around the world. Over the years SHADOW has grown thanks to several contributions: B. Lai[2],[3] K. Chapman,[4] G. J. Chen[5],[6] S. Singh,[7] Welnak[8],[9] among others. At the origins of SHADOW, the software engineering discipline was just starting and the present guidelines to keep a code maintainable were not yet raised. At that time the scientific comunity used mostly FORTRAN77 (F77) in Digital VAX-VMS environment with extensive use of common blocks, implicit variables and instructions like `goto`. Nowadays, these are techniques to be avoided. Among many other upgrades found in SHADOW3, the common blocks and implicit variables are replaced by derived types and explicit variables.

The new version SHADOW3[10] has been developed in Fortran 2003 (F03), adopting a modular structure and using new techniques of software engineering. A modular structure allows the users to import SHADOW3 functionality into their programs. Adding new features comes easy in this frame: a new module can be added if one wants to extend a project. The new version also makes use of new data types such as the derived types. A derived type is a container of variables, the counterpart of a `struct` in C. The latest standard of Fortran, Fortran 2008 (F08), will allow to implement methods in the derived types, making Fortran a fully object oriented programming language. We avoided using new F08 features as they are not yet implemented in the most popular free compilers, i.e. `gfortran` and `g95`.

SHADOW is a file oriented code, it launches an executable, `trace`. At each step of the tracing process, `trace` reads the beam, a collection of rays or photons from a file, moves it up to the next element and writes the result

Further author information: (Send correspondence to C.N.)
C.N. : E-mail: niccolo.canestrari@esrf.eu, Telephone: +33 (0)4 76 88 29 52
K.D. : E-mail: dimitris.karkoulis@esrf.eu, Telephone: +33 (0)4 76 88 17 49
S.R.M.: E-mail: srio@esrf.eu, Telephone: +33 (0)4 76 88 25 13

to a new file. Reading and writing files are among the slowest operations, so this strategy, indeed simple and reliable, comes with a problem. What happens if one wants to run several million of rays? What if one wants to repeat the tracing process over a long loop? The program spends more time in reading and writing then in performing the calculations, making it less robust. In order to overcome this important limitation it was deemed necessary to transform the kernel of SHADOW into a library. A set of functions are available for the advanced users who want to program their own executable. However the SHADOW executables are still provided with full backward-compatibility with the previous versions.

A consistent way to bind SHADOW3 with C is put in place. A list of F03 functions are made available (exposed) to C, which forms a C Application Programming Interface (API). Moreover the C layer permits to go further and embed SHADOW3 in `python` and `IDL`. We tested parallelization techniques, such as MPI and OpenCL to speed up part of the code. The status of the parallelization of SHADOW3 is discussed in the section 4.

2. DESCRIPTION OF THE FORTRAN MODULES

The new SHADOW3 is presented as a F03 set of modules (see Table 1). It is meant to provide the same functionality of the previous version of SHADOW and to be backward-compatible. The code has been rearranged following modern standards in computer engineering to allow the developers to maintain and extend the software.

Table 1. Modules description, each module is listed following the compilation order.

Module	Task
shadow_globaldefinitions	definition of the kind of integer, real, complex and character
stringio	string manipulation routines.
gfile	definition of type `gftype` and methods used for input-output source and optical element files `start.XX`
shadow_beamio	subroutines to read and write rays from and to files
shadow_math	set of mathematical subroutines and functions
shadow_variables	types for source and optical elements
shadow_kernel	the fundamental routines of SHADOW for ray-tracing.
shadow_synchrotron	subroutines to generate synchrotron sources: bending magnets, wigglers and undulators
shadow_pre_sync	preprocessors for synchrotron sources
shadow_preprocessors	preprocessors for SHADOW
shadow_postprocessors	postprocessors for the analysis and visualization of SHADOW results
shadow_bind	F03 subroutines to be called by C

The use of global variables has been reduced drastically, replaced by local variables in modules, with the idea of removing them completely in the next releases. Indeed using global variables is considered bad practice in computer science. It is preferable to adopt other strategies, like passing more arguments to a function or subroutine, or to organize variables logically in structures. These approaches are safer and ensure a better maintainability of the code. Implicit variables also have been partially removed. Again, the use of implicit variables is not recommended and the new F03 keeps them only for backward-compatibility with F77. Each module has been designed to do a specific job, so developers can easily find where to search what they need to make their own extensions or their own executables. A new main program called `shadow3` has been developed, being an interactive command line environment similar to the collection of main programs of the previous version.

3. APPLICATION PROGRAMMING INTERFACE (API)

In this section the API for five languages, F03, C, C++, python and IDL are presented together with examples that can be used as a guide. Each subsection will describe the available routines, their tasks, their input arguments and their outputs.

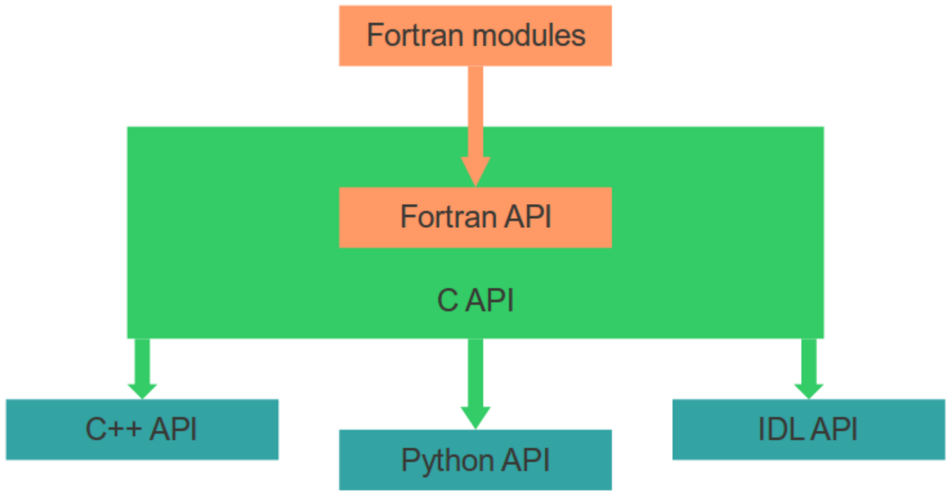

Figure 1. Schematic structure of the API and its relationship with the Fortran code

The module `shadow_bind` contains the FORTRAN API, and a set of bind-to-C subroutines using the `ISO_C_BINDING` module of F03. These subroutines can be easily called by C. A C-Layer has been written, it provides to advanced users a typical interface to develop main programs with SHADOW functionality using the C language. Moreover, almost all the scripting languages used in science, such as python, IDL, matlab have an interface with C. So one can use this layer as a basis to embed SHADOW3 to these script languages. SHADOW3 has already been successfully embedded in `IDL` and `python`, which are included in the package. A schematic representation of SHADOW3 API is in Figure 1

3.1 Fortran API

A Fortran developer can potentially use all the procedures of the library, but the Fortran API is defined by a smaller set of them. The principal derived types used in SHADOW3 are two, `poolSource` and `poolOE`. `poolSource` holds all the variables necessary to represent the source. The user of SHADOW will recognize the variables listed in the file `start.00`. `poolOE` holds instead the variables necessary for the optical element which in shadow were listed in files like `start.XX`.

The two derived types mentioned above are declared public in the module `shadow_variables`. They perform the basic operations for reading from file and writing to file. In the module `shadow_variables` four subroutines helps for that purpose: `PoolSourceLoad`, `PoolSourceWrite`, `PoolOELoad`, `PoolOEWrite`. All of them have the same prototype so only one will be described. `PoolSourceLoad` takes two arguments, a `poolSource` type “src” and a string “filename” with the name of the file to be read.

Rays are stored in a double precision array of 18 x NPOINT. The user decides the number of rays in the field `NPOINT` of the derived type `poolSource`. The 18 components of the ray are: 3 coordinates for the position; 3 for the direction; the 3 components of the σ-polarized electric field; a check flag; the wave-number expressed in $2\pi/\lambda$; a flag with the ID number of the ray; the optical path; the phase of the σ-polarized electric field; the phase of the π-polarized electric field; and the 3 components of the π-polarized electric field.

In the previous version of SHADOW, a ray could have 12, 13 or 18 components. That was due to the fact that in the 70’s the amount of memory available was low and very expensive. Nowadays, memory availability is no longer an issue and for uniformity we decided to fix number of components to 18. This array can be read and written into files using three subroutines in the module `shadow_beamio`: `beamGetDim`, `beamLoad` and `beamWrite`. To preserve the compatibility with the previous version the reading routines (`beamGetDim`, `beamLoad`) can work even if the number of components per ray is 12 or 13, but `beamWrite` only writes in the new format. Therefore it is possible to import and export files from previous calculations. The user of SHADOW is familiar with these files using standard names such as `begin.dat` for the rays of the source, `star.01` for the rays after the first optical element, or `screen.0101` for the rays on the first screen in the first optical element. All these files have the same structure and can be read and written with the procedures listed in Table 2.

Table 2. Types and routines exposed regarding file input/output, source generation and ray tracing. Each subroutine is followed by the arguments it needs and their kind.

Name	Task	arguments	module
PoolSource	description of the source		shadow_variables
PoolOE	description of the optical element (OE)		shadow_variables
PoolSourceLoad	read source from file	src (poolSource), fname (string)	shadow_variables
PoolSourceWrite	write source to file	src (poolSource), fname (string)	shadow_variables
PoolOELoad	read OE from file	src (poolOE), fname (string)	shadow_variables
PoolOEWrite	write OE to file	src (poolOE), fname (string)	shadow_variables
beamGetDim	inspect a file containing the rays	fname (string), ncol (integer), npoint (integer), iFLag (integer), iErr (integer)	shadow_beamio
beamLoad	read rays from a file	ray (real(18,npoint)), iErr (integer), ncol (integer), npoint (integer), fname(string)	shadow_beamio
beamWrite	write rays to a file	ray (real(18,npoint)), iErr (integer), ncol (integer), npoint (integer), fname(string)	shadow_beamio
SourceGeom	generate a geometrical source	src (poolSource), ray (real), npoint (integer)	shadow_kernel
SourceSync	generate a synchrotron source	src (poolSource), ray (real), npoint (integer)	shadow_synchrotron
TraceOE	trace rays to the next image plane	src (poolSource), ray (real), npoint (integer), icount (integer)	shadow_kernel

`beamGetDim` inspects the file to find the number of fields, NCOL, and the number of rays, NPOINT. It is necessary before reading the actual data to allocate the right amount of memory, `beamLoad` fills a previously allocated array with the data of the file and `beamWrite` writes the array to the file.

The pricipal routines in the ray tracing library are dedicated to create the source, and to trace it. The source routines in SHADOW3 are two. One used for geometrical sources, it is called `SourceGeom` and it belongs to the module `shadow_kernel`. The other is `SourceSync` used for bending magnets, wigglers and undulator sources (synchrotron) in the module `shadow_synchrotron`. The trace routine is called `TraceOE` and it is part of the module `shadow_kernel`. It trace the rays through the optical element described by the derived type `poolOE`. The subroutine `traceOE` can take into account only one optical element at a time. The output array has the informations about the rays in the image plane. SHADOW3 is able to perform the Fresnel-Kirchhoff integral. This feature was present in the previous version of SHADOW for 1D-detector (`FFresnel`) and has been extended to support 2D ones. `FFresnel2D` is found in the `shadow_postprocessors` module. We have extensively used `FFresnel2D` for parallelization tests (see subsection 4.2.4)

An example of a main program using the library SHADOW3 can be found in Listing 1. It creates and writes a source and then traces an optical system with one single optical element. In the first lines the necessary modules are imported and variables are declared. The program starts loading `start.00` into `src`, changes the number of rays, and allocates memory. It generates the source and prints it in `begin.dat`. In the next step it loads `start.01` in `oe1`, runs the tracing subroutine `traceOE` and writes the result in `star.01`.

```
program example
  use shadow_globaldefinitions, only  : skr, ski
  use shadow_beamio, only             : beamWrite
  use shadow_variables, only          : poolSource, poolOE, &
                                        PoolSourceLoad, PoolOELoad
  use shadow_kernel, only             : TraceOE
  use shadow_synchrotron, only        : SourceSync

  implicit none
  type(poolSource)                               :: src
  type(poolOE)                                   :: oe1
  real(kind=skr), allocatable, dimension(:,:)   :: ray
  integer(kind=ski)                              :: iErr

  call PoolSourceLoad(src,"start.00")
  src%npoint=100000

  allocate( ray(18,src%npoint) )

  call SourceSync(src,ray,src%npoint)
  call beamWrite(ray,ierr,18,src%npoint,"begin.dat")

  call PoolOELoad(oe1,"start.01")

  call TraceOE(oe1,ray,src%npoint,1)
  call beamWrite(ray,ierr,18,src%npoint,"star.01")

  deallocate(ray)

  stop
end program example
```

Listing 1. Simple main program showing the use of the SHADOW Fortran API

3.2 C API

The Fortran API described before has been linked with C. The first step to build this layer was to choose the part of the Fortran code to expose. We opted for the routines listed in the previous section. Each one of these routines (e.g., `TraceOE`) has a template in the module `shadow_bind` (eg. `BindShadowTraceOE`). This module makes use of `ISO_C_BINDING`, a Fortran internal module, to override the difficulties in the interoperability with C. This module provides interoperable kind and pointers and it can ensure the name of a subroutine in the symbols of the library, which is necessary to link it to any C code. Interoperability of kind is simply given by a list of kind types, e.g., an integer of kind `C_INT` corresponds in C to an int, and `C_DOUBLE` to a double. For a routine to be interoperable all its arguments must be interoperable. The same is true for a derived type, it is possible to expose it if all its members are interoperable. C arrays are passed by pointer, but if one wants to pass the dimension, she has to give an additional argument to the function. Instead, F03 arrays have shape, which describes the number of its dimensions and their magnitude. `ISO_C_BINDING` makes available to developers `C_F_POINTER`, a subroutine to associate a C pointer to its F counterpart. As an optional argument it is possible to give also the shape of the array. An example used in the code is:

```
call c_f_pointer(ray_c_ptr,ray_f_ptr,[18,npoint1])
```

Some comments regarding strings, in C strings are arrays of characters with the ending character '\0'. In Fortran instead they are structures with a reference to a reserved memory and an integer with the length of the string. Unfortunately, this structure has other fields as well, and their order and number are compiler dependent.

To successfully pass string to F03 one can use the same tecnique of the arrays. Subroutines and functions can be exposed using the keyword `bind` in the following F03 way:

```
subroutine SubExample(arg1,arg2) bind (C,name='SubExample')
```

where all the arguments are interoperable. The keyword "name" forces the compiler to use exactly the word "SubExample" in the table of symbols of the library.

The C API provides functions for C programmers. These functions are bound with the routines of the F03 modules `shadow_bind`. The Listing 2 is the translation of the Listing 1 into the C programming language.

Table 3. Functions of the C API. Each function is followed by its description and the underlying Fortran routine.

C function	Task	Fortran routine
AllocateBeamFromPool	allocates a chunck of memory according to NPOINT in src. It returns the new pointer	none
AllocateBeam	allocates a chunck of memory according to NPOINT. It returns the new pointer	none
CShadowPoolSourceLoad	reads the content of a file like start.00 into the struct src	PoolSourceLoad
CShadowPoolSourceWrite	write the content of src in the file fname	PoolSourceWrite
CShadowPoolOELoad	read the content of a file like start.01 into the struct oe	PoolOELoad
CShadowPoolOEWrite	write the content of oe in the file fname	PoolOEWrite
CShadowGetDimRay	retrieve the dimension of the rays written in a binary file like begin.dat	beamGetDim
CShadowBeamLoad	read the rays from a file like begin.dat and store them in the previously allocated memory pointed by ray	beamLoad
CShadowBeamWrite	write the rays in a binary file	beamWrite
CShadowSourceGeom	call SourceGeom to generate the rays according to the structure src	SourceGeom
CShadowSourceSync	call SourceSync to generate the rays according to the structure src	SourceSync
CShadowTraceOE	call TraceOE for the data in ray according to oe	TraceOE

```
#include <stdio.h>
#include <string.h>
#include "shadow_bind_c.h"

int main()
{
  poolSource src;
  poolOE     oe1;
  double     *beam=NULL;

  CShadowPoolSourceLoad(&src, "start.00");
  src.NPOINT=100000;
  beam = AllocateBeamFromPool(&src,beam);
  CShadowSourceSync(&src, beam);
  CShadowBeamWrite(beam,18,src.NPOINT,"begin.dat");

  CShadowPoolOELoad(&oe1,"start.01");
  CShadowTraceOE(&oe1,beam,src.NPOINT,1);
  CShadowBeamWrite(beam,18,src.NPOINT,"star.01");

  return EXIT_SUCCESS;
}
```

Listing 2. Simple main program showing the API in C

3.3 C++ API

C++ is the object oriented successor of C and it includes almost all the features of C, which means that a C++ compiler can process almost all C sources. C++ introduces the concept of class and inheritance and provides a new library for strings. It is also equipped with a library of standard containers, such as vector, deque, list, map or queue, and a set of highly optimized algorithms e.g., to sort the elements of containers. Having a C bind done for SHADOW3, it was easy to extend it to C++. Three classes have been introduced, `Source`, `OE` and `Beam`. The first two of them correspond to the structures `poolSource` and `poolOE`, and the last one corresponds to the array. The class is often described as an entity holding a set of data (members) and functions (methods). The classes Source and OE have been directly inherited by their C counterparts:

```
  class Source : public poolSource {...};
  class OE : public poolOE {...};
```

These two classes are thought as variables containers, therefore they provide only two methods one to load a file of variables and one to write it. The class `Beam` contains a pointer to the block of memory, which will be passed to the routines of SHADOW for processing. the pointer is passed internally through the methods defined within the class, these methods are called `genSource` and `trace`. The first one is bound to both `SourceGeom` and `SourceSync`, since it calls one or the other depending on the characteristics of the source, while the second one corresponds to `TraceOE`. The class `Beam` provides also a way to store the data to a file and to recover the data from a file. Table 4 shows the list of classes with their methods, while the Listing 3 shows the precedent C example in the new C++ paradigm.

Table 4. Methods of the C++ API per class, each function is followed by its arguments and description.

C++ class	C++ method	arguments	Task
Source	load	char *fname	load the file named fname
Source	write	char *fname	write Source to the file fname
OE	load	char *fname	load the file named fname
OE	write	char *fname	write OE to the file fname
Beam	load	char *fname	load the binary file named fname
Beam	write	char *fname	write Beam to the binary file fname
Beam	genSource	Source *src	generate the rays using the src definitions
Beam	trace	OE *oe, int id	trace the beam to the idth optical element oe

```
#include "shadow_bind_cpp.hpp"

int main()
{
  Source  src;
  OE      oe;
  Beam    beam;

  src.load("start.00");
  src.NPOINT=100000;
  beam.genSource(&src);
  beam.write("begin.dat");

  oe.load("start.01");
  beam.trace(&oe,1);
  beam.write("star.01");

  return 0;
}
```

Listing 3. Simple main program using the API in C++

3.4 python API

C is a very flexible language, and it has been in used for a long time to develop operating systems, kernels, as well as many popular applications, and among them python. Many methods exist to interface python with C code. One can use auto-generating code like: swig, pyrex, CXX, weave and others. Another way is to follow the Python/C API [*]. We found convinient to use the latter to create `Shadow`, the python module of SHADOW3. It makes use of `numpy` a scientific computing module for python. `numpy` provides the container "array", very useful in scientific software, because of its robustness and built-in functionality.

The module `Shadow` is composed, as in C++, by three classes: `Beam`, `Source` and `OE`. `Source` and `OE` correspond to `poolSource` and `poolOE`, and they have two methods, `load` and `write`. These methods perform the reading and writing of file like `start.00` and `start.01`. The members of these classes are directly accessible, and their type is fixed. Whenever the user instantiates the class, the instance will have default values. These values are the default ones for source and optical element in SHADOWVUI.

[*]docs.python.org/c-api/

Beam has a single member, rays, a numpy array of 64 bits floats. This array counts 18 columns and npoint rows to hold the data for the photons. Four methods are associated to Beam, read, write, genSource and traceOE. read and write are the usual input-output from binary file like begin.dat. genSorce takes a Source instance in input and fill the member rays with the generated data for the photons. It works both for geometric and synchrotron sources. traceOE takes OE instance and an integer, a counter for the optical element. It fills the member rays with the data of the photons in the image plane. An example of the python code is in the Listing.4.

```
import Shadow as sd
from numpy import *

ray = sd.Ray()

src = sd.Source()
src.load("start.00")
src.NPOINT = 100000

ray.genSource(src)
ray.write("begin.dat")

oe1 = sd.OE()
oe1.load("start.01")

ray.traceOE(oe1,1)
ray.write("star.00")
```

Listing 4. Simple main program showing the API in python

3.5 IDL API

IDL is a scientific programming language developed by ITTVIS [†]. It was choosen to develop a file oriented Graphical User Interface (GUI) for SHADOWVUI. Many users indeed use SHADOW as a plugin of XOP[11] called SHADOWVUI. An IDL interface to the modules of SHADOW3 has been built. Since several IDL functions were developed around SHADOW, this API is already rich in features. Many of them related to the GUI, such as windows were the user can define sources and optical elements, all the plotting system and the post-process, but also functions to read and write definitions files (like start.00 and start.XX) and binary files for rays. All these have been developed natively in IDL.

IDL structures of data are created in order to prepare the environment for shadow to run. Afterwards, when this process is complete, SHADOWVUI is able to read the results and plot them. In order to skip the use of files we need to pass these IDL structures directly to the subroutines of SHADOW3 and to expose these subroutines to IDL. The C layer already developed was used for this purposes. The binding of IDL to the library are minimalist (see Table 5), only three functions are actually exposed, SourceGeom, SourceSync and TraceOE. The rest of the tasks (load, write, pre- and post-processors) are done directly in IDL using the rich existing SHADOWVUI library.

[†] www.ittvis.com

Table 5. Functions of the IDL API, each function is followed by its description and the underlying Fortran function.

Routine	Task	F90 Routine
GENSOURCEGEOM	call SourceGeom to generate the rays according to the structure src	SourceGeom
GENSOURCESYNC	call SourceSync to generate the rays according to the structure src	SourceSync
TRACEOE	call TraceOE for the data in ray according to oe	TraceOE

4. PARALLELIZATION

Ray-tracing falls in the family of embarrassingly parallel problems. The problem of solving the trajectory for two rays can be completely decoupled, hence the problem is data independent and each ray may be computed independently. By Amdahl's law,[12] such problems are expected to scale near linearly. There are two distinctive approaches to parallelization:

1. Task-parallelism, that suggests the concurrent execution of different or same tasks on different processes.
2. Data-parallelism, the application of data-segmentation on the problem and performing the same task in one process on all or groups of the data segments concurrently.

Both approaches are studied for the parallelization of SHADOW3. Data-parallelism of a specific SHADOW3 post-processor, the `FFresnel2D`, is studied and its performance demonstrated on Graphics Processing Units (GPU), since GPUs are ideal for massively data-parallel problems.

4.1 Parallelization of SHADOW3. Implementation of task-parallelism using OpenMPI

We first studied the feasibility of using the Open Multi-Processing (OpenMP), an API for shared memory multiprocessing, for data-parallelism for SHADOW3. Upon the creation of the rays by source, they were segmented in groups of rays and traced in parallel. The problem with this implementation relies on the fact that there are still global variables present in the SHADOW3 modules. Such variables are shared to each thread, unless explicitly stated otherwise in each module. As a result, considerable intervention would be required on various levels of the SHADOW3 source code, which deems this approach costly, so this alternative was left for a future planned reorganization of SHADOW3 variables.

For the task-parallel implementation, OpenMPI is used. OpenMPI relies on spawning processes which can communicate with each other, but a shared memory mechanism is not present. In this approach, the distribution of work among the processors is done by performing a common task on different input data sets, one for each process. Each process uses a unique sequence of random numbers and input data set, which it traces to build its unique trajectories. The results can then be gathered by one process and analysed further. Due to its minimal requirement for alterations in SHADOW3, this approach was found to be the most suitable.

Using this approach, an OpenMPI version of the `trace3` program is created using the F90 API. This version, called `trace3_mpi` reads the source definition from the `start.00` file and OE's definitions from the `start.xx` and `systemfile.dat`, as commonly done with SHADOW. Depending on the `N` number of processors available, `NPOINT` is rescaled to `NPOINT*N`, and each block of `NPOINT` rays is calculated by a different processor. The results can be:

1. analyzed to create a common histogram,
2. written to separate files `star.01-1,...,star.01-N`, or
3. be combined into a single `star.01` file with `NPOINT*N` rays.

4.2 GPU acceleration of SHADOW

A common approach to accelerating a program with the use of GPU devices, or even traditional parallelization, is to profile the program and identify the bottlenecks. This is very practical especially on large packages like SHADOW, as it ensures that the limited development resources are allocated where the most performance benefit can be achieved. In contrary to traditional parallelization, GPU acceleration requires that the part of the program of interest is completely rewritten and many times even the algorithm itself redesigned from scratch. These difficulties, along with the limited resources, justify that in most cases GPU acceleration targets only the most important bottlenecks. Not all of them can be tackled by GPU devices, since, as discussed later, they have to be able to follow strict models to be efficient. In the case of SHADOW3, `trace` and `source` are not compute-intensive, while the new `FFresnel2D` postprocessor requires some hours even for moderate resolution detectors. For that reason, it is a good candidate for GPU acceleration.

4.2.1 General Purpose GPU Programming

The evolution of General Purpose GPU programming (GPGPU) through the last years has made the use of GPU devices an asset to scientific computing. The ever increasing need for processing power, has attracted the interest of High Performance Compiting (HPC) to GPUs as they feature an order of magnitude higher peak floating point operations per second (FLOPS) than commercial multicore CPU processors, while maintaining similar pricing.

GPU devices are by design massively multithreaded and optimised in every aspect of the parallel execution of data independant vectorizable problems. The assumption of such problems requires minimal branch prediction which allows the GPUs to devote more silicon to be used for Shader Processors (SP or shaders). An aditional interesting feature is that they can efficiently hide latency in computations, meaning the clock cycles required by the issued memory and arithmetic operations to complete.

Internally, a GPU device is mainly composed by its processor, the Dynamic Random-Access Memory (DRAM) and the bridge (interconnection) to the main system. Inside the processor, the vast number of shaders is physically organized in the Streaming Multiprocessors (SM), sharing a number of instruction units and Level 1 (L1) cache memory amongst other resources. SMs may also be organized in Clusters, sharing Level 2 (L2) cache memory. The DRAM can achieve access speeds an order of a magnitude higher than conventional RAM memory, but with the downside of strict access patterns for such performance to be achieved. The bridge, currently Peripheral Component Interconnect Express version 2.0 (PCIe 2), is essential since, as a peripheral device, the GPU device is controlled by the CPU. Moreover, only through it the data can be transfered to the GPU. The bridge is considered the most serious bottleneck in a GPU device and thus data transfers between CPU and GPU should be minimized.

The programming model of the GPU, which can be distinguished in the execution and memory access models, introduce the steep learning curve of GPU programming and is critical to performance. The code that is executed on a GPU is called a kernel. The kernel is in fact a loop over all the defined threads. The execution model defines how these threads are organized and executed in a GPU. The threads are executed in groups of few that are called warps and their size depends solely on the device and is not user-definable. The threads are also organized in blocks and grids, whose size is defined in the program. At any given point of the execution the scheduler may load many warps per SM, depending on the available resources. There is no guarantee to execution order of the warps. The scheduler may shuffle active warps for execution as it sees fit in order to hide arithmetic and memory access latency. In few words, the warps follow the Single Input Multiple Data (SIMD) model, while the blocks and grids the Single Instruction Multiple Threads (SIMT). SIMT allows for flow divergence with no penalty, while such divergence in a warp may cause partial or total seriallization of the execution within that warp.

The memory model defines the hierarchy of the memory, the access patterns that should be used, and the scope. Memory is organized in global, shared and local memory, implying the scope. Global is visible to all the threads, shared has the scope and lifetime of a block and local the scope and lifetime of a thread. The data must be transfered from RAM to the DRAM in allocated global memory in order to be able to be used in a kernel. Physically, the global memory resides on the DRAM, the shared memory is on the on-chip cache and the local memory may be on the DRAM or, on the on-chip cache in which case it is also called a register. Typically, shared memory and registers are very limited in resources, but very fast featuring performance in the TB/sec scale, and

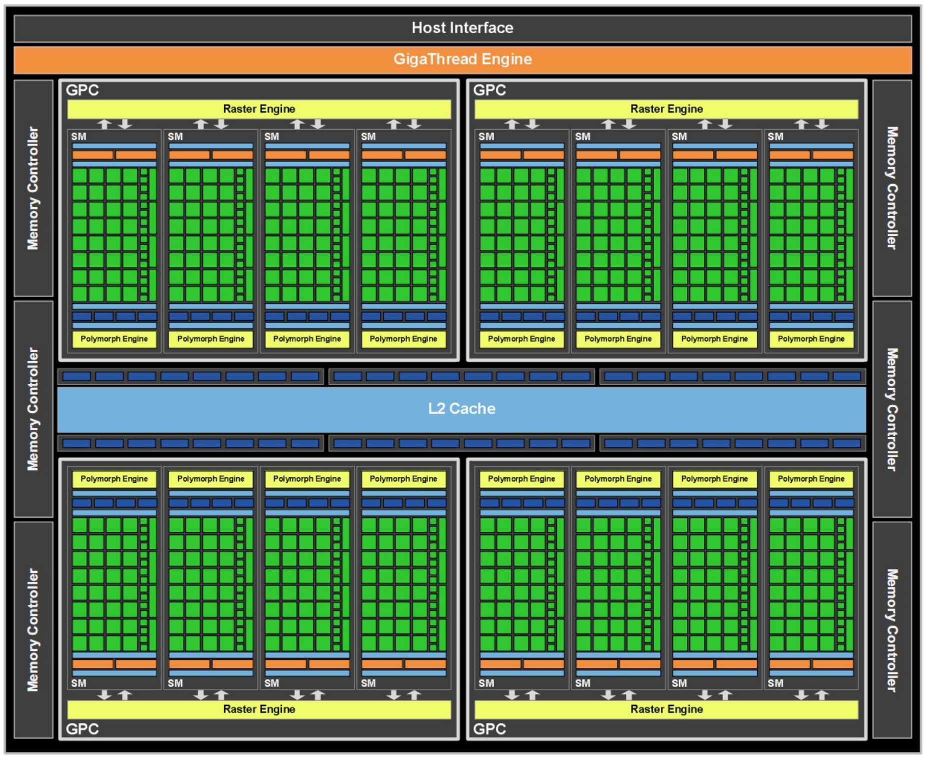

Figure 2. Schematic of the NVIDIA Fermi GPU architecture, found in the GF110 GPU processor.

their saturation can greatly affect performance. Moreover, global and shared memory have strict access patters that must be obeyed in order to achieve good performance. The programming model, in most cases, makes it necessary that algorithms are restructured or reinvented before they can be ported to the GPU. In most cases, it is necessary to refactor algorithms before they can be ported to GPU.

4.2.2 The choice of programming language for GPGPU

A GPU is only accessible through its device driver. Specific APIs have been developed that allow the control of a GPU in a program. Most notable propriety APIs are, the Compute Unified Device Architecture (CUDA) Runtime and Driver API by NVIDIA and the Stream API by AMD, but they are designed only for the GPU devices of the according vendor. These APIs are accompanied by specific programming languages for the programming of kernels. Typically, these languages are an extension of a subset of C. There is however the open-source Open Computing Language (OpenCL) API which was designed to unite the different APIs and programming languages provided by each vendor. It currently supports INTEL and AMD processors, NVIDIA and AMD GPU devices, VIA processors and the IBM CELL Broadband engine processor. The OpenCL API is very similar to the CUDA Driver API. The programming language used to develop OpenCL kernels, whether they are to be executed in a CPU, GPU or CELL, is an extension to a subset of the C(99) language.

The NVIDIA CUDA APIs are the most established in GPGPU, due to the early development of these APIs. They are feature rich and a great amount of computational libraries and tools are already available. The Runtime API is high level, making the control and execution of a kernel straightforward, while the Driver API is low level and allows for full control over the device, at the cost of longer code and increased complexity.

The OpenCL API is low level and in logic similar to the CUDA driver API. Yet it is still rather new, lacking the extensive libraries found in CUDA. Its main advantage is the ability to compile and execute a kernel on devices of different types and vendors.

All the APIs provide the necessary functions to scan for suitable devices, create a control context, allocate necessary global memory, copy data from the system to the device and configure and execute kernels. Such a sequence is a typical flow of a program that requires to execute a kernel. Moreover, new scalar and vector datatypes are typically defined.

GPU acceleration of SHADOW3 was done as part of the European Union Linksceem-2 project. One of the main goals of this work package is the porting of algorithms used at synchtrotrons to GPUs using OpenCL.

4.2.3 GPU parallelization

As mentioned before, the complete porting of SHADOW3, like any other large package, is a tremendous task that cannot always be fulfilled with the current resources available. To study the possible use of GPUs, we have selected a compute-intensive routine that makes wave propagation, `FFresnel`. In the original version, it calculates for each point on the detector (considered `1D`) the electric field and phases originates by propagating (Fresnel-Kirchhoff integral) each ray from a SHADOW file (`star.01`, `screen.0101`, etc). The `FFresnel` routine has been extended to 2D-detectors, thus calculating electric fields and phases (as well as intensity) produced by `NPOINT` rays into a 2D detector of `Nx*Ny` pixels. `FFresnel2D` calculates 6 `Nx*Ny` images (real and imaginary parts of the Electric fields $\vec{E_\sigma}$,$\vec{E_\pi}$) and combines them to the intensity image.

4.2.4 FFresnel2D GPU implementation

The Fresnel-Kirchhoff integral problem is by nature data-independent and non-divergent on the image side, the contribution of each ray (source plane) must be taken into account on each pixel of the image (target plane). However, on the source side it is data-dependent, due to memory collisions caused by many threads trying to add their contribution to the same pixel of the image. Data-independency can be achieved however, if the algorithm is restructured in such way that memory collisions are avoided. The ray-based approach is complex in comparison to the image-based approach, but it should perform better when the size of the image is small. Both approaches were studied and implemented into GPU algorithms.

Ray-based algorithm In the ray-based algorithm each thread maps to a ray. Data-independency is achieved and memory collisions are avoided by applying cyclic updates of the coefficients on the image. Cyclic update is an iterative process, where the distribution of the subimages to the blocks is rotated with each new iteration. First (see top Figure 3), the image is segmented into $\mathtt{N_s}$ subimages and these subimages are distributed to the blocks accordingly. After $\mathtt{N_s}$ updates all subimages will be updated with the contributions of all the rays on all the blocks, as shown on Figure 3 (*a* to *d*). Inside each block (Figure 3 bottom), the same method is applied to avoid memory collisions. One subimage is loaded in the shared memory and iteratively each thread calculates the Fresnel-Kirchhoff integral for a unique pixel at any given time and updates it (Bottom Figure 3 *a* to *d*). When the update is complete the results are stored back in the global memory in order to be accessible by the next block that will load this subimage. There are two disadvantages with this algorithm, first it requires $\mathtt{N_s}$ kernels to be launched. $\mathtt{N_s}$ can be in the order of thousands which results in added overhead. Secondly more complexity is introduced by the cyclic updates when padding needs to be taken into account.

Image-based algorithm The image-based algorithm maps each thread to a unique image pixel. In figure 4 for example, $pixel_i$ is mapped to $thread_i$. The 2D image is mapped to 1D space and handled as such internally, where the $\{x, y\}$ coordinates of any pixel can be retrieved by decomposition of the global thread index. Each thread is required to loop over all the available rays and add their contribution. Ideally for the GPU algorithm, the rays would be organized in column per component and row per ray, since then the rays and their components can be accessed in a way that maximizes global memory read efficiency, but that is not the case. If we consider that in each line of the algorithm, one element of a given ray is used in an arithmetic operation and that warps are implicitly synchronized, this means that all the threads of a warp would read the same location on the global memory and then they would all read one memory location that is padded 18 elements away (`ray` has 18 components). This is not efficient performance wise, since in the same operation global memory is able to provide 128 bytes width of data in one memory fetch and we are only utilizing a small fraction at a time. Such problems are commonly solved with the use of shared memory, as initially we can have all the threads read the global memory sequentially, component by component and store the data in the shared memory. This way of reading data from the global memory is efficient as it follows the memory model and fully utilizes the available

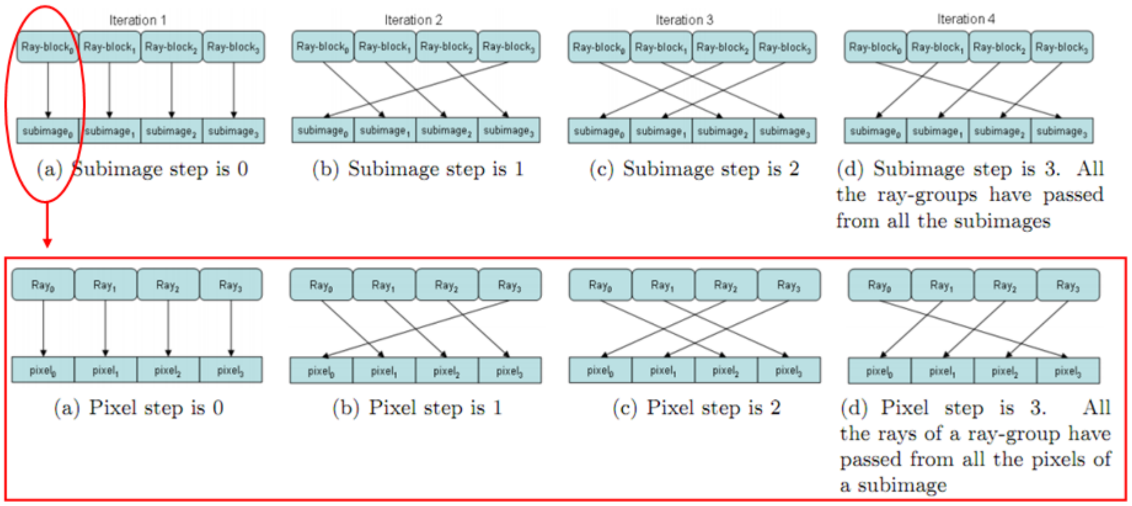

Figure 3. Top: Simple example of the ray-based algorithm cyclic image updates on the complete image level, for a system of 4 ray-groups. All the subimages are updated concurrently. The subimage steps are wrapped to image width (if next subimage is out of bounds, we continue from the first subimage) Bottom: Details of the cyclic update performed on the block (subimage) level.

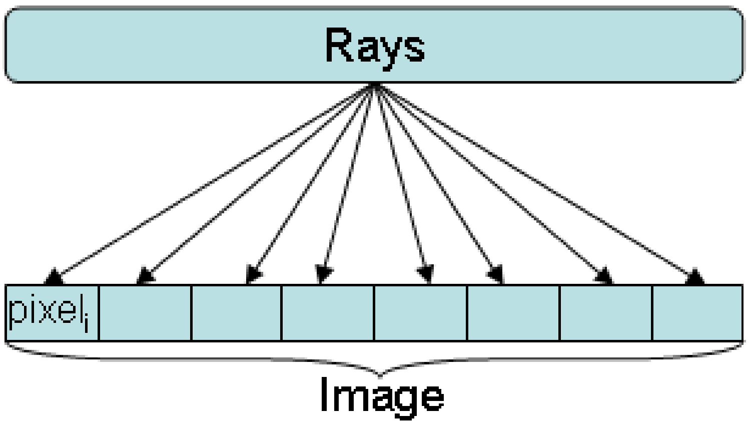

Figure 4. Image-based algorithm. Each thread updates one pixel only, using all the rays. All the pixels are updated in parallel.

memory bandwidth. Then each thread of a warp will be reading the same address (bank) on the shared memory, a process called broadcast, which yields maximum performance for the shared memory.

4.2.5 Performance results

The main goal for GPGPU programming is performance and a demonstration of an algorithm typically is accompanied by benchmark results. For the benchmarks a GPU TESLA computing node of the ESRF HPC facilities was used. The computing node consists of two quad-core Intel XEON E5640 processors with operating frequency 2,67 GHz and 48GB of RAM and two TESLA M2050 GPU devices with 448 shaders and 3Gb of DRAM each. A single CPU core (sequential `FFresnel2D`) is tested against a a single GPU (`FFresnel2D` GPU), as well as the ray-based algorithm against the image-based on the M2050 GPU. Benchmarks of the two GPU algorithms showed than even with an image size of merely $128 * 128$ pixels, sufficient GPU utilization is achieved when using the image-based algorithm which is faster by roughly 10% that the ray-based one. This difference in performance between the two GPU algorithms is mostly due to the extensive use of the expensive `modulo` operations in the ray-based algorithm, in order to create the cyclic memory access behavior. As such the image-based algorithm was selected for the benchmarks against the sequential `FFresnel2D`.

The elapsed times for the execution of the sequential and the GPU image-based algorithm are shown in Table 6 and 7 respectively. The acceleration factor for a given configuration of beam and image size can be found in Table 8. A specific number of rays is depicted by a line, while the image size (number of pixels) by a column. All elapsed times are in seconds. In Table 6 we can see that by moving by a column or by row the execution time quadruples, which is expected due to the $O(n^2)$ complexity of the algorithm. We decided to not go above 512^2 pixels and 512^2 rays, since it is straightforward to estimate the execution time of consequent increases in size, in example a configuration of 1024^2 pixels and 1024^2 rays would take roughly two days to complete on our system.

Some of the characteristics of GPUs can be analyzed from Table 7. If we move across a column, we can see that the execution time increases according to the complexity. In that case the number of pixels is fixed and consequently so are the threads defined in the GPU. Across a row however things are very different. Now every time the image size, and thus the problem size, is increased so is the number of threads. Notice that by increasing the pixel size from 32^2 to 64^2 the elapsed time remains roughly the same, which means that at 32^2 we are far from utilizing the device efficiently. This is a fine example of the scaling behavior of GPU devices and how they are able to handle hundreds of thousand of threads efficiently, while maintaining only a few thousands active at a time and swapping them with inactive (always in warps) to hide latencies. In fact we can see that, regardless the number of rays, only for image size of 1024^2 and up the execution time truly quadruples, while at 512^2 is close to that limit of complete saturation of the GPU device.

Table 6. `FFresnel2D` CPU execution times $(t_{CPU})(s)$ for various beam (number of rays) and image sizes.

Rays\Pixels	$32*32$	$64*64$	$128*128$	$256*256$	$512*512$
$32*32$	0.16	0.63	2.61	10.34	41.47
$64*64$	0.63	2.50	10.37	41.37	165.84
$128*128$	2.51	10.06	41.36	165.62	663.10
$256*256$	10.02	40.14	165.49	662.52	2652.52
$512*512$	40.15	160.59	662.16	2649.95	10613.45

Table 7. `FFresnel2D` GPU (image-based) execution times $(t_{GPU})(s)$ for various beam (number of rays) and image sizes.

Rays\Pixels	$32*32$	$64*64$	$128*128$	$256*256$	$512*512$	$1024*1024$	$2048*2048$
$32*32$	0.0263	0.0276	0.0829	0.289	1.12	4.43	17.70
$64*64$	0.0995	0.103	0.321	1.14	4.44	17.62	70.40
$128*128$	0.392	0.407	1.28	4.54	17.69	70.34	281.19
$256*256$	1.57	1.62	5.10	18.15	70.80	281.32	1124.33
$512*512$	6.26	6.48	20.39	72.58	283.14	1125.23	4496.93
$1024*1024$	25.02	25.93	81.54	290.27	1132.93	4500.51	17987.69
$2048*2048$	N/A	N/A	N/A	N/A	N/A	18004.66	71948.99

Table 8. `FFresnel2D` GPU acceleration factor (t_{CPU}/t_{GPU}).

Rays\Pixels	$32*32$	$64*64$	$128*128$	$256*256$	$512*512$
$32*32$	6.08	22.8	31.5	35.6	37.0
$64*64$	6.33	24.3	32.3	36.3	37.4
$128*128$	6.40	24.7	32.3	36.5	**37.5**
$256*256$	6.38	24.8	32.4	36.5	**37.5**
$512*512$	6.41	24.8	32.5	36.5	**37.5**

The main limiting factors in the performance of `FFresnel2D` GPU is the use of double-precision complex numbers, as standard in SHADOW. GPUs are optimized for single-precision operations and even though the performance of double precision has improved dramatically in the last generation of GPU architectures, is still at best 1/4 the performance of single-precision. Also, the scarce shared memory and register resources are saturated quickly by the increased need of memory of double-precision complex numbers, limiting the utilization of all the GPU shaders. It is however possible to introduce some approximations which will allow the use of single-precision for the calculation of the intensity image and could be studied for a later version of SHADOW.

OpenCL is not yet mature and lacks some tools like the support for complex numbers and the ability to initialize the global memory directly through the API. As a result we had to implement complex numbers as vectors of width 2 and implement all the required complex number mathematical operations as well kernels to initialize the memory. Finally, we note that while a kernel may be executed on many different kinds of devices, that does not mean it will perform efficiently on all of them. An algorithm written for an NVIDIA GPU most probably will not perform as it should on an AMD GPU and vice-versa. We measured, that when given one physical core, the kernel executed on the CPU via OpenCL performed twice as slow than the reference algorithm.

5. ACKNOWLEDGEMENTS

The SHADOW3-API has been funded by the European Union FP7 Infrastructures LABSYNC (grant number 213126). The GPU acceleration work was done as part of the European Union FP7 programme, under the LinkSCEEM-2 (INFRA-2010-1.2.3) Work Package 12 project (grant number RI-261600). The computing infrastructure required for GPU programming was provided by the ESRF HPC center, in Grenoble, France.

REFERENCES

[1] Cerrina, F. and Sanchez del Rio, M., "Ray-tracing of x-ray optical systems," in [*Handbook of Optics Volume V. Third Edition*], Bass, M., ed., Mc Graw Hill (2010).

[2] Lai, B., Chapman, K. and Cerrina, F., "SHADOW: New developments," *Nuclear Instruments Methods in Physics Research Section a-Accelerators Spectrometers Detectors and Associated Equipment* **266**, 544–549 (1988).

[3] Lai, B., Chapman, K., Runkle, P. and Cerrina, F., "SHADOW: New developments," *Review of Scientific Instruments* **60**, 2127–2127 (1989).

[4] Chapman, K., Lai, B., Cerrina, F. and Viccaro, J., "Modeling Of Undulator Sources," *Nuclear Instruments and Methods in Physics Research Section a-Accelerators Spectrometers Detectors and Associated Equipment* **283**, 88–99 (1989).

[5] Chen, G. J., Cerrina, F., Voss, K. F., Kim, K. H. and Brown, F. C., "Ray-tracing of X-ray focusing capillaries," *Nuclear Instruments and Methods in Physics Research Section a-Accelerators Spectrometers Detectors and Associated Equipment* **347**, 407–411 (1994).

[6] Chen, G. J., Guo, J. Z. Y. and Cerrina, F., "Applications Of Faceted Mirrors To X-Ray-Lithography Beamlines," *Nuclear Instruments and Methods in Physics Research Section a-Accelerators Spectrometers Detectors and Associated Equipment* **347**, 238–243 (1994).

[7] Singh, S., Solak, H. and Cerrina, F., "Multilayer roughness and image formation in the Schwarzschild objective," *Review of Scientific Instruments* **67**, 3355–3355 (1996).

[8] Welnak, C., Anderson, P., Khan, M., Singh, S. and Cerrina, F., "Recent Developments In Shadow," *Review of Scientific Instruments* **63**, 865–868 (1992).

[9] Welnak, C., Chen, G. J. and Cerrina, F., "Shadow - A Synchrotron-Radiation And X-Ray Optics Simulation Tool," *Nuclear Instruments and Methods in Physics Research Section a-Accelerators Spectrometers Detectors and Associated Equipment* **347**, 344–347 (1994).

[10] Sanchez del Rio, N. Canestrari, F. J. and Cerrina, F., "Shadow3: A new version of the synchrotron x-ray optics modelling package," *Journal of Synchrotron Radiation* **In Press**, xxx (2011).

[11] Sanchez del Rio, M. and Dejus, R. J., "Status of XOP: an x-ray optics software toolkit," *SPIE Proc.* **5536**, 171–174 (2004).

[12] Amdahl, G., "Validity of the single processor approach to achieving large scale computing capabilities," *AFIPS Conference Proceedings* **30**, 483–485 (1967).