Design and Performance of the OP2 Library for Unstructured Mesh Applications[*]

Carlo Bertolli[1], Adam Betts[1], Gihan Mudalige[2], Mike Giles[2], and Paul Kelly[1]

[1] Department of Computing, Imperial College London
[2] Oxford e-Research Centre, University of Oxford

Abstract. OP2 is an "active" library framework for the solution of unstructured mesh applications. It aims to decouple the scientific specification of an application from its parallel implementation to achieve code longevity and near-optimal performance by re-targeting the back-end to different multi-core/many-core hardware. This paper presents the design of the OP2 code generation and compiler framework which, given an application written using the OP2 API, generates efficient code for state-of-the-art hardware (e.g. GPUs and multi-core CPUs). Through a representative unstructured mesh application we demonstrate the capabilities of the compiler framework to utilize the same OP2 hardware specific run-time support functionalities. Performance results show that the impact due to this sharing of basic functionalities is negligible.

1 Introduction

OP2 is an "active" library framework for the solution of unstructured mesh applications. It utilizes code generation to exploit parallelism on heterogeneous multi-core/many-core architectures. The "active" library approach uses program transformation tools, so that a single application code written using the OP2 API is transformed into the appropriate form that can be linked against a target parallel implementation (e.g. OpenMP, CUDA, OpenCL, AVX, MPI, etc.) enabling execution on different back-end hardware platforms.

Such an abstraction enables application developers to focus on solving problems at a higher level and not worry about architecture specific optimisations. This splits the problem space into (1) a higher application level where scientists and engineers concentrate on solving domain specific problems and write code that remains unchanged for different underlying hardware and (2) a lower implementation level, that focuses on how a computation can be executed most efficiently on a given platform by carefully analysing the data access patterns. This paves the way for easily integrating support for any future novel hardware architecture.

[*] This research is partly funded by EPSRC (grant reference numbers EP/I00677X/1, EP/I006079/1), the UK Technology Strategy Board, and Rolls- Royce plc through the SILOET programme.

To facilitate the development of unstructured mesh applications at a higher hardware agnostic level, OP2 provides both a C/C++ and a Fortran API. Currently an application written using this API can be transformed into code that can be executed on a single multi-core and/or multi-threaded CPU node (using OpenMP) or a single GPU (using NVIDIA CUDA). In this paper we introduce the design of the code transformation/compiler framework, which supports OP2's multi-language/multi-platform development capability. We show how, hardware specific optimisations can be utilised independently of the application development language and present key issues we encountered and their performance effects during the execution of a representative CFD application written using the OP2 API.

More specifically we make the following contributions:

1. We present the design of the OP2 compiler framework, which translates an application written using the OP2 API in to back-end hardware implementations. Key design features of this framework are illustrated with a stepwise analysis of this process and the resulting optimisation opportunities, during code transformation.
2. A representative CFD application written using the OP2 C/C++ API is redeveloped using the OP2 Fortran API and the contrasting performance of these two applications are explored on two modern GPU platforms (NVIDIA GTX460 and Fermi M2050).
3. Both C/C++ and the Fortran based applications uses the same hardware specific back-end irrespective of the application language; we show that the impact due to this sharing of basic functionalities is negligible.

The paper is organised as following: Section 2 describes relevant related work. Section 3 gives an overview of the OP2 functions. Section 4 describes the compiler architecture, and its implementation. In Section 5 we provide results of experiments on C/C++ and Fortran programs for CUDA. Finally, Section 6 conlcludes with plans for future research.

2 Related Work

OP2 is the second iteration of OPlus (Oxford Parallel Library for Unstructured Solvers) [3]. OPlus provided an abstraction framework for performing unstructured mesh based computations across a distributed-memory cluster. It is currently used as the underlying parallelisation library for Hydra a production-grade CFD application used in turbomachinery design at Rolls-Royce plc. OP2 builds upon the features provided by its predecessor but develops an "active" library approach with code generation to exploit parallelism on heterogeneous multi-core/many-core architectures.

Although OPlus pre-dates it, OPlus and OP2 can be viewed as an instantiation of the AEcute (access-execute descriptor) [9] programming model that separates the specification of a computational kernel with its parallel iteration space, from a declarative specification of how each iteration accesses its data. The decoupled Access/Execute specification in turn creates the opportunity to apply powerful optimisations targeting the underlying hardware. A number of related research projects have implemented similar programming frameworks. The most comparable of these is LISZT [4, 5] from Stanford University.

LISZT is a domain specific language specifically targeted to support unstructured mesh application development. The aim, as with OP2, is to exploit information about the structure of data and the nature of the algorithms in the code and to apply aggressive and platform specific optimisations. Preliminary performance figures from the LISZT framework have been presented in [5]. The authors report the performance of Joe, a fluid flow unstructured mesh application using a mesh of 750K cells, on a Tesla C2050 (implemented using CUDA) against an Intel Core 2 Quad, 2.66GHz processor. Results show a speed-up of about 30× in single precision arithmetic and 28× in double precision relative to a single CPU thread.

3 OP2

Unstructured meshes are used over a wide range of computational science applications. The are applied in the solution of partial differential equations (PDEs) in computational fluid dynamics (CFD), computational electro-magnetics (CEM), structural mechanics and general finite element methods. Usually, in three dimensions, millions of elements are often required for the desired solution accuracy, leading to significant computational costs.

Unlike structured meshes, they use connectivity information to specify the mesh topology. In OP2 an unstructured mesh mesh problem specification involves breaking down the algorithm into four distinct parts: (1) sets, (2) data on sets, (3) connectivity (or mapping) between the sets and (4) operations over sets. These leads to an API through which any mesh or graph can be completely and abstractly defined. Depending on the application, a set can consist of nodes, edges, triangular faces, quadrilateral faces, or other elements. Associated with these sets are data (e.g. node coordinates, edge weights) and mappings between sets which define how elements of one set connect with the elements of another set.

Fig. 1 illustrates a simple quadrilateral mesh that we will use as an example to describe the OP2 API. The mesh can be defined by two sets, nodes (vertices) and cells (quadrilaterals). There are 16 nodes and 9 cells, which can be defined using the OP2 API as shown in Fig. 2. In our previous works [8] we have detailed the OP2 API for code development in C/C++. Here we will introduce the Fortran API.

The connectivity is declared through the mappings between the sets. The integer array `cell_map` can be used to represent the four nodes that make up each cell, as shown in Fig. 2.

Each element of set `cells` is mapped to four different elements in set `nodes`. The `op_map` declaration defines this mapping where `mcell` has a dimension of

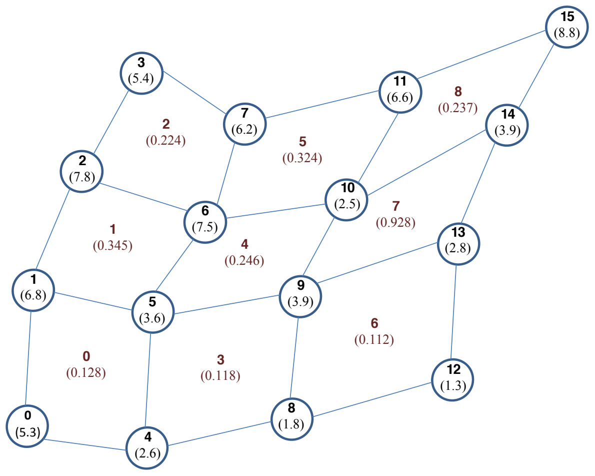

Fig. 1. A mesh example used through the paper

```
integer(4) :: numNodes = 16
integer(4) :: numCells = 9

type(op_set) :: nodes, cells

integer(4), dimension(36) :: cell_map = (/ 0,1,5,4, 1,2,6,5, 2,3,7,6, &
  & 4,5,9,8,5,6,10,9,6,7,11,10, &
  & 8,9,13,12,9,10,14,13,10,11,15,14 /)

type(op_map) :: cellsToNodes

call op_decl_set ( numNodes, nodes )
call op_decl_set ( numCells, cells )

call op_decl_map ( cells, nodes, 4, cell_map, cellsToNodes )
```

Fig. 2. Example of declaration of op_set and op_map variables.

4 and thus its index 0,1,2,3 maps to nodes 0,1,5,4, and so on. When declaring a mapping we pass the source and destination sets (`cells` and *nodes*), and the dimension of each map entry, which for `mcell` it is 4.

Once the sets are defined, data can be associated with the sets; in Figure 3 we show some data arrays that contain double precision data associated with the cells and the nodes respectively. Note that here a single double precision value per set element is declared. A vector of a number of values per set element could also be declared (e.g. a vector with three doubles per node to store the X,Y,Z coordinates).

All numerically intensive computations in the application can be described as operations over sets. This corresponds to loops over a given set, accessing data through the mappings (i.e. one level of indirection), performing some calculations, then writing back (possibly through the mappings) to the data arrays. If

```
real(8), dimension(9) :: cell_data = (/ 0.128, 0.345, 0.224, 0.118, &
  & 0.246, 0.324, 0.112, 0.928, 0.237 /)

real(8), dimension(16) :: nodes_data = (/ 5.3, 6.8, 7.8, 5.4, &
  & 2.6, 3.6, 7.5, 6.2, 1.8, 3.9, 2.5, 6.6, 1.3, 2.8, 3.9, 8.8 /)

type(op_dat) :: dataCells, dataCellsUpdated, dataNodes

call op_decl_dat ( cells, 1, cell_data, dataCells )
call op_decl_dat ( nodes, 1, nodes_data, dataNodes )
```

Fig. 3. Example of data array declaration and OP2 variables.

the loop involves indirection we refer to it as an indirect loop; if not, it is called a direct loop.

The OP2 API provides a parallel loop function which allows the user to declare the computation over sets. Consider the sequential loop in Figure 4, operating over each mesh cell. Each cell updates its data value using the data values held on the four nodes connected to that cell. An application developer declares this loop using, together with the "elemental" kernel function, as shown in Fig. 5. OP2 handles the architecture specific code generation. The elemental kernel function takes six arguments in this case and the parallel loop declaration requires the access method of each to be declared (OP_WRITE, OP_READ, etc). OP_ID indicates that the data is to be accessed without any indirection (i.e. directly). `dnodes` on the other hand is accessed through the `mcell` mapping using the given index

OP2's general decomposition of unstructured mesh algorithms imposes no restrictions on the actual algorithms, it just separates the components of a code. However, OP2 makes an important restriction that the order in which elements are processed must not affect the final result, to within the limits of finite precision floating-point arithmetic. This constraint allows the program to choose its own order to obtain maximum parallelism. Moreover the sets and mappings between sets must be static and only one level of indirection is allowed.

```
subroutine seqLoop ( numCells, cell_map, cellDataUpdated, cellData, &
                   & nodeData )
  integer(4) :: numCells
  integer(4), dimension(:) :: cell_map
  real(8), dimension(:) :: cellDataUpdated, cellData, nodeData

  integer(4) :: i
  do i = 1, numCells
    cellDataUpdated(i) = cellData(i) + nodeData( cell_map(4*i) ) + &
     nodeData( cell_map(4*i +1) ) + nodeData( cell_map(4*i +2) ) + &
     nodeData( cell_map(4*i +3) )  )
  end do
end subroutine seqLoop
```

Fig. 4. Example of sequential loops.

```
subroutine kernel ( cellUpdated, cell, node1, node2, node3, node4 )
  real(8), dimension(1) :: cellUpdated, cell, node1, node2, node3, node4

  cellUpdated(1) = cell(1) + node1(1) + node2(1) + node3(1) + node4(1)
end subroutine kernel

call op_par_loop ( kernel, cells, &
                 & dataCellsUpdated, -1, OP_ID, OP_WRITE,
                 & dataCells, -1, OP_ID, OP_READ,
                 & dataNodes, 1, cellsToNodes, OP_READ,
                 & dataNodes, 2, cellsToNodes, OP_READ,
                 & dataNodes, 3, cellsToNodes, OP_READ,
                 & dataNodes, 4, cellsToNodes, OP_READ )
```

Fig. 5. Example of op_par_loop corresponding to the sequential loops showed above.

The OP2 API targets explicit relaxation methods such as Jacobi iteration; pseudo-time-stepping methods; multi-grid methods which use explicit smoothers; Krylov subspace methods with explicit preconditioning. However, algorithms based on order dependent relaxation methods, such as Gauss-Seidel or ILU (incomplete LU decomposition), lie beyond the capabilities of the API.

4 OP2 Code Generation and Compiler Framework

The OP2 compiler is based on the ROSE framework [10]. ROSE supports front ends for C/C++ and Fortran 77-2003; it generates an Abstract Syntax Tree (AST) of the input program, which we use to analyse, optimise and transform the input OP2 programs. Our compiler infrastructure, illustrated in Fig.6, performs the following set of tasks:

1. Type and consistency checks. For instance, the compiler checks that the basic type of an *op_dat* is the same as a corresponding formal parameter in a user kernel declaration.
2. Host subroutine generation. This subroutine generates the parallel computation to be applied to the different set partitions. In case of indirect loops, it calls the OP2 run-time *plan* generation function. For each partition colour, this subroutine invokes a backend-specific subroutine (see next item) which performs the calculations on a particular subset of the set.
3. Backend-specific subroutine generation. In CUDA, this subroutine is a kernel, while in OpenMP it is a vanilla subroutine invoked in parallel by the threads. Its main task is to call the user kernel to perform the required computations. As partitions are internally coloured, to avoid race conditions and hence the need for locks, this subroutine iterates over such colours, serially invoking the user kernel on all elements with same colours.
4. Transformation of the user kernel. In the case of CUDA some additional labels are needed to inform the back-end compiler which subroutines intend to run on the device. Also, Fortran CUDA [1] requires that all device subroutines be included in the same Fortran module.

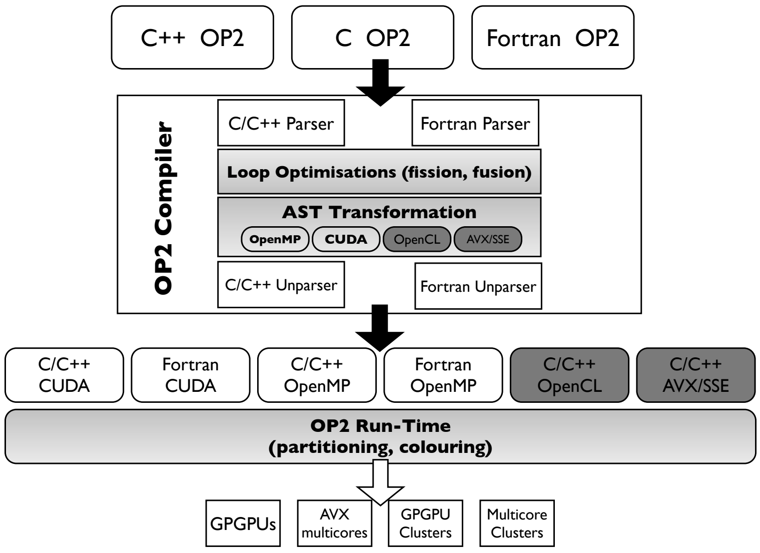

Fig. 6. Architecture of the OP2 ROSE-based compiler. Shaded grey blocks represent the main OP2 functionalities, where darker squared blocks are features currently under development.

Currently, the compiler produces a CUDA implementation of C/C++ and Fortran OP2 programs, while we are developing OpenCL and AVX/SSE back-ends. Further back-ends will include optimised code generation for new heterogeneous architectures, like AMD's APU [2].

Optimisations are the cardinal points of this development: as shown in Figure 6, a set of program transformations, like loop fusion and fission, will be independent of the input language. This independence is easily obtained because ROSE maps input programs of different input languages to an orthogonal AST representation, called Sage III. The ASTs for C and Fortran programs are mainly based on the same AST node types, except for some minor differences which we treat as special cases. However, the need of defining optimisations at this level might require a further abstraction over the AST, to easily manipulate the program without dealing with low-level compiler details.

Other transformations and configurations are instead dependent on the target architecture, and they define the design choices that our compiler targets. For instance, the compiler can select optimal thread numbers in a different way for NVIDIA and AMD GPUs.

In the same figure we also show a further cardinal point in the design of OP2. If we consider different generated back-end programs, originated from different input languages and targeting different architectures, we can see that some of them share a same C-based run-time support. The run-time includes basic OP2 declaration routines (e.g. *op_decl_set*), as well as the colouring and partitioning logics. In other words, a same colouring and partitioning algorithm is used by different back-ends. For instance, both Fortran and C++ generated programs targeting CUDA and OpenMP make use of the same implementation of colouring and partitioning algorithms.

This choice comes at a performance cost for Fortran generated programs. In fact, they need to interoperate with C run-time support functions, and to transform the resulting variables from Fortran to C notation. The compiler generates code to minimise the number of variables which have to be transformed, but in some cases, either due to algorithmic reasons, or to lower level compiler bugs, we are forced to re-execute part of the transformation at each invokation of a op_par_loop . The extent of this cost is targeted in Section 5.

5 Performance

The example application used in this paper, Airfoil, is a non-linear 2D inviscid Airfoil code that uses an unstructured grid [7]. It is a much simpler application than the Hydra [6] CFD application used at Rolls-Royce plc. for the simulation of turbomachinery, but is representative of a production grade unstructured mesh application. The mesh used in our experiments is of size 1200×600, consisting in over 720K nodes, 720K cells and about 1.5 million edges. The code consists of five parallel loops: `save_soln`, `adt_calc`, `res_calc`, `bres_calc`, `update`. The most compute intensive loop `res_calc` has about 100 floating-point operations performed per mesh edge and is called 2000 times during total execution of the application. `save_soln` and `update` are direct loops while the other three are indirect loops.

In this section we show the performance measurements on two GPUs of the Airfoil application, implemented in Fortran and C++, based on double precision floating point numbers. The used GPUs are: a popular consumer graphics card (Nvidia GeForce GTX460) and a high performance computing card (NVIDIA Fermi M2050). For space reasons, we only show results related to the execution on GPUs, while we will target in future work specific optimisations and performance measurements for multicore processors.

Target of these performance measurements is to understand which is the cost of the use of a common run-time support, implemented in C, for both Fortran and C/C++ OP2 generated programs. From the Fortran side this involves: (i) to define a proper interface between C functions and Fortran code, to allow interoperability of function calls; (ii) to translate variables generated in the C functions to Fortran variables. Both points are implemented by using the Fortran 2003 standard binding support, supported by the ISO_C_BINDING Fortran module. At run-time, we employ the *c_f_pointer* function to convert variables from C to Fortran.

Another main difference between C/C++ and Fortran lies in the different implementation of the CUDA kernels, which are language dependent. Practically, while the C implementation makes extensive use of pointers to the GPU shared memory to address specific sub-portion of *op_dat* variables, the Fortran

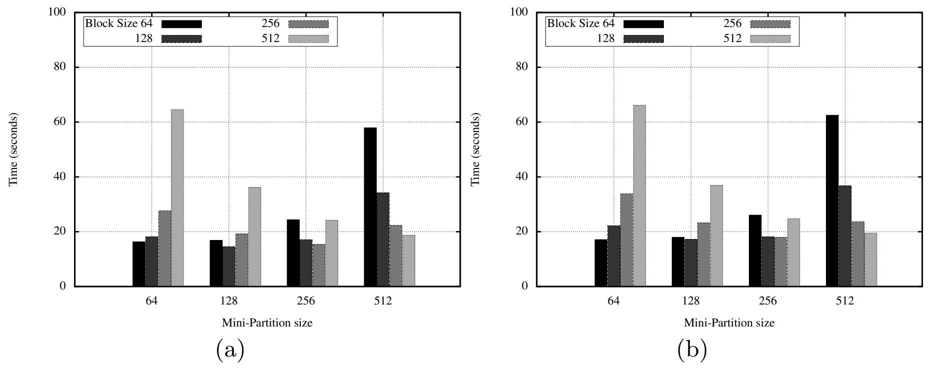

Fig. 7. Performance of C++ (left) and Fortran (right) Airfoil on a M2050.

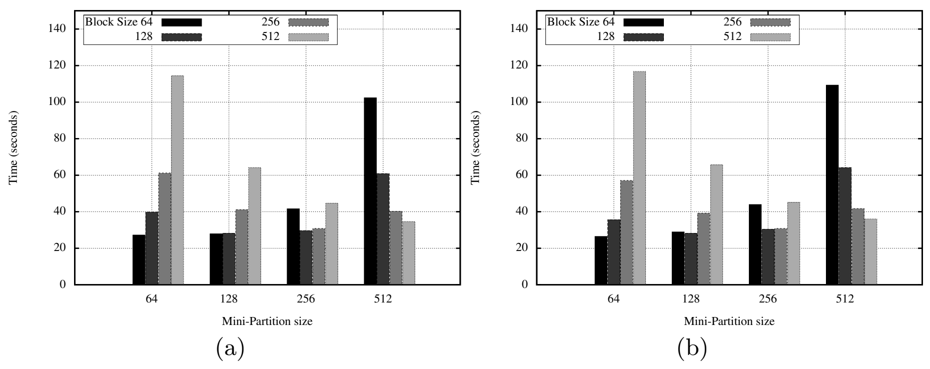

Fig. 8. Performance of C++ (left) and Fortran (right) Airfoil on a GTX460.

implementation lacks of a support for such kinds of pointers, and it is forced to re-compute the offset for shared memory variables.

Unlike the previous interoperability issue, this latter one represents the critical point in the difference between the Fortran and C implementation of OP2 on GPUs. For this reason, our performance measurements specifically target the execution time in the kernel code.

Fig. 7 shows the performance measurements on the M2050 GPU, by varying the number of set elements in each block (*partition size*), and the number of threads in a CUDA block (*block size*). If the partition size is equal to the block size, then each thread is assigned a single set element to which it applies the kernel. If the partition size is a multiple of the block size, then each thread applies the kernel on multiple set elements. Finally, if the block size is a multiple of the partition size, then a section of the threads in a block is not used, leaving executing threads more resources.

Similar results, even if with a lower performance, are shown in Fig. 8 for the execution of the C++ and Fortran program on the GTX 460 GPU.

6 Conclusion

In this paper we have described the source-to-source compiler framework for OP2 applications, targeting unstructured mesh CFD applications. The compiler currently supports different input languages (namely C/C++ and Fortran), and it generates back-end architecture implementation for multicores, using OpenMP, and GPUs, using CUDA. We have shown the specific design choices in the compiler architecture, which are the basis over which we will provide language and back-end independent optimisations, as well as back-end dependent optimal configurations. In addition, the generated code for Fortran and C/C++ makes use of the same set of core run-time OP2 functions, implementing main application logics, like mesh colouring and partitioning.

We have presented performance results of the execution of a CFD application on two GPUs, showing a almost identical performance for Fortran and C/C++ CUDA implementations.

References

1. Pgi cuda fortran (2011), http://www.pgroup.com/resources/cudafortran.htm/
2. The amd fusion family of apus (2011), http://sites.amd.com/us/fusion/apu/Pages/fusion.aspx
3. Burgess, D.A., Crumpton, P.I., Giles, M.B.: A parallel framework for unstructured grid solvers. In: Proc. of the 2nd European Computational Fluid Dynamics Conf. pp. 391–396. John Wiley and Sons, Germany (September 1994)
4. Chafi, H., DeVito, Z., Moors, A., Rompf, T., Sujeeth, A.K., Hanrahan, P., Odersky, M., Olukotun, K.: Language virtualization for heterogeneous parallel computing. In: Proc. of the OOPSLA '10 Applications. pp. 835–847. ACM, USA (2010)
5. DeVito, Z., Joubert, N., Medina, M., Barrientos, M., Oakley, S., Alonso, J., Darve, E., Ham, F., Hanrahan, P.: Liszt: Programming mesh based pdes on heterogeneous parallel platforms (Oct 2010), available at: http://psaap.stanford.edu
6. Giles, M.B.: Hydra (1998-2002), http://people.maths.ox.ac.uk/gilesm/hydra.html
7. Giles, M.B., Ghate, D., Duta, M.C.: Using automatic differentiation for adjoint CFD code development. Computational Fluid Dynamics J. 16(4), 434–443 (2008)
8. Giles, M.B., Mudalige, G.R., Sharif, Z., Markall, G., Kelly, P.H.: Performance analysis of the op2 framework on many-core architectures. SIGMETRICS Perform. Eval. Rev. 38, 9–15 (March 2011)
9. Howes, L.W., Lokhmotov, A., Donaldson, A.F., Kelly, P.H.J.: Deriving efficient data movement from decoupled access/execute specifications. In: High Performance Embedded Architectures and Compilers, LNCS, vol. 5409, pp. 168–182. Springer (2009)
10. Schordan, M., Quinlan, D.: A source-to-source architecture for user-defined optimizations. In: Bszrmnyi, L., Schojer, P. (eds.) Modular Programming Languages, LNCS, vol. 2789, pp. 214–223. Springer Berlin / Heidelberg (2003)