A Domain-Specific Optimizing Compiler for Solving Partial Differential Equations

Fabio Luporini*, Florian Rathgeber*, Ana Lucia Varbanescu†, Gheorghe-Teodor Bercea*,
Paul H. J. Kelly*, and David Ham*
*Imperial College London, Department of Computing,
†TU Delft, Faculty of Engineering, Mathematics and Computer Science (EWI)
*{f.luporini12|f.rathgeber10|gheorghe-teodor.bercea08|p.kelly|d.ham@imperial.ac.uk}@imperial.ac.uk,
†a.l.varbanescu@uva.nl

***Abstract*—The abstract goes here.**

I. Introduction

In many fields, like computational fluid dynamics, computational electromagnetics, and structural mechanics, programs model phenomena by means of partial differential equations (PDEs). Numerical techniques, like Finite Volume Method and Finite Element Method, are widely employed in real-world applications to approximate solutions of PDEs. Unstructured meshes are often used to discretize the equation domain, since their geometric flexibility allows solvers to be extremely effective. The solution is sought in each cell of the discretized domain by applying suitable numerical kernels. Iteration over mesh components is also expensive, due to the presence of indirect memory accesses. As the number of cells can be of the order of millions, a major issue is the time required to execute the computation, which can be hours or days. To address this issue, domain-specific languages (DSLs) have been developed. The successful porting of Hydra, a computational fluid dynamics industrial application devised by Rolls Royce for turbomachinery design (based on Finite Volume Method, roughly 50000 lines of code and mesh sizes that can be over 100 millions edges), to OP2 [3], demonstrates the effectiveness of the DSL approach for implementing PDEs solvers [**?**].

OP2 adopts a kernel-oriented programming model, in which the computation semantics is expressed through self-contained functions ("kernels"). A kernel is applied to all elements in a set of mesh components (e.g. edges, vertices, elements), with an implicit synchronization between the application of two consecutive kernels. On commodity multi-cores, a kernel is executed sequentially by a thread, while parallelism is achieved partitioning the mesh and assigning each partition to a thread. Together with the indirections problem, kernel optimization is one of the major concerns in unstructured mesh applications. In this paper, we tackle this problem by proposing a domain-driven optimization strategy for a class of kernels used in Finite Element Methods.

We focus on Local Assembly ("assembly", in the following), a fundamental step of a Finite Element Method that covers an important fraction of the overall computation runtime, typically in the range 30%-60%. During the assembly phase, the solution of the PDE is approximated by executing a suitable kernel over all elements in the discretized domain. A kernel's working set is usually small enough to fit the L1 cache; it might need L2 cache when high-order methods are employed to improve the accuracy of the solution. However, we do not consider the latter case. An assembly kernel is characterized by the presence of an affine, usually non-perfect loop nest, where individual loops are rather small (the trip count rarely exceeds 40, with a minimum value of 3, depending on the order of the method). With such small kernels, we focus on aspects like minimization of floating-point operations, register allocation and instruction-level parallelism, especially in the form of SIMD vectorisation.

Since loop nests are affine, one possibility is to rely on polyhedral compilation for improving performance. There are, however, several limitations. The first problem is that a polyhedral compiler reasons on iteration spaces, whereas some of the transformations we propose rely on analysis of the data space. Our attempts at using PoCC's optimizations [**?**], especially register tiling, were unsuccessful. A domain-specific transformation of the iteration space, which is not expressible in the polyhedral model because of its non-affine nature, also showed a significant performance gain due to improved register locality.

Assembly kernels could be turned into a sequence of calls to BLAS routines, although it is complicated to achieve and, potentially, ineffective. One issue concerns automating identification and extraction of matrix-matrix multiplications from the code, which requires a non-trivial analysis of the memory access pattern. In addition, register locality would be affected, since the same matrix can appear in a number of products. Finally, it is known that BLAS routines perform far from peak performance when the dimension of the matrices is small [5]. Hand-made implementations of the simple Helmholtz assembly kernel as a sequence of BLAS $dgemm$ calls, for a range of polynomial orders ($p = 1, 2, 3, 4$), has showed that the achieved performance is worse than that exposed in Section V.

Given the constraints on polyhedral compilation and linear-algebra-specialized libraries, we address assembly optimization by studying a set of domain-specific code transformations, applicable to a wide class of problems. We have developed a domain-specific optimizing compiler and we have integrated it with Firedrake, a system for solving PDEs through the Finite Element Method [**?**] based on the OP2 abstraction. This allows us to evaluate our code transformations in a range of real-world problems, varying parameters that impact both solution accuracy and kernel cost, namely the polynomial order of the

```
void helmholtz(double A[3][3], double **coords) {
  // K, det = Compute Jacobian (coords)

  static const double W3[3] = {...}
  static const double X_D10[3][3] = {{...}}
  static const double X_D01[3][3] = {{...}}

  for (int i = 0; i<3; i++)
    for (int j = 0; j<3; j++)
      for (int k = 0; k<3; k++)
        A[j][k] += ((Y[i][k]*Y[i][j]+
           +((K[1]*X_D10[i][k]+K[3]*X_D01[i][k])*
           *(K[1]*X_D10[i][j]+K[3]*X_D01[i][j]))+
           +((K[0]*X_D10[i][k]+K[2]*X_D01[i][k])*
           *(K[0]*X_D10[i][j]+K[2]*X_D01[i][j])))*
           *det*W3[i]);
}
```

Algorithm 1: Local assembly code generated by Firedrake for a Helmholtz problem (2D mesh, Lagrange $p = 1$ elements).

method (from $p = 1$ to $p = 4$) and the mesh type (2D, 3D, 3D hybrid structured-unstructured).

Performance results show that Firedrake-generated code for non-trivial assembly kernels, if not adequately optimized, is sub-optimal. A cost-model-driven sequence of source-to-source code transformations, aimed at improving register allocation and register data locality, can result in performance improvements up to 33% over "softly-optimized" code (i.e. where only basic transformations are performed, such as auto-vectorizable loop-invariant code motion, padding and data alignment), and up to 77% over the original kernels. The contribution of this paper is therefore twofold:

- An optimisation strategy for a class of kernels widely-used in scientific applications, namely Local Assembly in the context of the Finite Element Method. Our approach exploits domain knowledge to go beyond the limits of both vendor (e.g. *icc*) and research (e.g. those based on the polyhedral model) compilers.
- Automation of such code optimizations in Firedrake. This enables performing an in-depth performance evaluation of the proposed optimization strategy in real-world simulations.

The paper is organized as follows. ...

II. Background

Local assembly consists of evaluating so called element stiffness matrix and vector; in this work, we focus on computation of matrices, which is the costly part of the process. A stiffness matrix can be intuitively thought as an approximated representation of the PDE solution in a specific cell of the discretized domain. Numerical integration algorithms are widely-used in scientific codes to implement assembly [4], [**?**].

Given a mathematical description of the input problem, expressed through the domain-specific Unified Form Language [**?**], Firedrake generates C-code kernels implementing assembly using a numerical integration algorithm. At the core of Firedrake [**?**] lies PyOP2 [3], which triggers compilation of such kernels using an available vendor compiler, and

```
void burgers(double A[12][12], double **c, double **w) {
  // K, det = Compute Jacobian (c)

  static const double W5[5] = {...}
  static const double X1_D001[5][12] = {{...}}
  static const double X2_D001[5][12] = {{...}}
  //It follows 11 other basis functions definitions.
  ...

  for (int i = 0; i<5; i++) {
    double F0 = 0.0;
    //It follows 10 other declarations (F1, F2,...)
    ...
    for (int r = 0; r<12; r++) {
      F0 += (w[r][0]*X1_D100[ip][r]);
      //It follows 10 analogous statements (F1, F2, ...)
    }
    ...
    for (int j = 0; j<12; j++)
      for (int k = 0; k<12; k++)
        A[j][k] += (..(K[5]*F9)+(K[8]*F10))*Y1[i][j])+
         +(((K[0]*X1_D100[i][k])+(K[3]*X1_D010[i][k])+
         +(K[6]*X1_D001[i][k]))*Y2[i][j]))*F11)+
         +(..((K[2]*X2_D100[i][k])+...+(K[8]*X2_D001[i][k])*
         *(K[2]*X2_D100[i][j])+...+(K[8]*X2_D001[i][j]))..)+
         + <roughly a hundred of sum/muls go here>)..)*
         *det*W5[i]);
  }
}
```

Algorithm 2: Local assembly code generated by FFC for a Burgers problem (3D mesh, Lagrange $p = 1$ elements).

eventually manages parallel execution over the mesh. In our work, we have enhanced this execution model by adding an optimization stage prior to the generation of C code. The code transformations described next are also generalizable to non-Firedrake assembly kernels, provided that numerical integration is used. Notable examples are [**?**], [**?**], [**?**], in which the kernel's loop nest and memory access pattern are reducible to that generated by Firedrake.

The complexity of Firedrake-generated kernels depends on the mathematical problem being solved. In simpler cases, the loop nest is perfect, it has short trip counts (in the range 3-15), and the computation reduces to a summation of a few, small outer-product vector-operations. An example is provided in Figure 1, which shows an assembly kernel for a Helmholtz problem, using Lagrange basis functions on 2D elements with polynomial order $p = 1$. In other scenarios, for instance when solving a non-linear problem like Burgers (see Figure 2), the number of arrays involved in the computation of A can be much larger: in this case, 14 unique arrays are accessed, and the same array can be referenced multiple times within the expression. Also, constants evaluated in outer loops (called F in the code), acting as scaling factors of arrays, may be required; trip counts can be larger (proportionally to the order of the method); arrays may be block-sparse. Despite such a big space of possible assembly implementations, it is still possible to identify a domain-specific structure common to all kernels, which can be turned into effective code transformations and simd-vectorization.

The class of kernels we are considering has, in particular, some peculiarities. 1) The computation of the Jacobian, which

```
void helmholtz(double A[3][4], double **coords) {
  // K, det = Compute Jacobian (coords)

  static const double W3[3] = {...}
  static const double X_D10[3][4] = {{...}}
  static const double X_D01[3][4] = {{...}}

  for (int i = 0; i<3; i++) {
    double LI_0[4];
    double LI_1[4];
    for (int r = 0; r<4; r++) {
      LI_0[r] = ((K[1]*X_D10[ip][r])+(K[3]*X_D01[ip][r]));
      LI_1[r] = ((K[0]*X_D10[ip][r])+(K[2]*X_D01[ip][r]));
    }
    for (int j = 0; j<3; j++)
      for (int k = 0; k<4; k++)
        A[j][k] += (Y[i][k]*Y[i][j]+LI_0[k]*LI_0[j]+
                    +LI_1[k]*LI_1[j])*det*W3[i]);
  }
}
```

Algorithm 3: Local assembly code generated by Firedrake for a Helmholtz problem (2D mesh, Lagrange $p = 1$ elements). The padding, data alignment and $licm$ optimizations are applied. Data alignment and padding relate to an AVX machine. In this specific case, sub-expressions invariant to j are identical to those invariant to k; in general, this could not be the case.

```
void helmholtz(double A[3][4], double **coords) {
  // K, det = Compute Jacobian (coords)
  // Declaration of basis function matrices

  for (int i = 0; i<3; i++) {
    double LI_0[4];
    double LI_1[4];
    for (int r = 0; r<4; r++) {
      LI_0[r] = ((K[1]*X_D10[ip][r])+(K[3]*X_D01[ip][r]));
      LI_1[r] = ((K[0]*X_D10[ip][r])+(K[2]*X_D01[ip][r]));
    }
    for (int j = 0; j<3; j++)
      for (int k = 0; k<4; k++)
        A[j][k] += (Y[i][k]*Y[i][j]+LI_0[k]*LI_0[j])*det*W3[i];
    for (int j = 0; j<3; j++)
      for (int k = 0; k<4; k++)
        A[j][k] += LI_1[k]*LI_1[j]*det*W3[i];
  }
}
```

Algorithm 4: Local assembly code generated by Firedrake for a Helmholtz problem (2D mesh, Lagrange $p = 1$ elements). The padding, data alignment, $licm$ and $split$ optimizations are applied. Data alignment and padding relate to an AVX machine. In this specific case, the split factor is 2.

is the first step of the assembly, is independent of the loop nest; this is not true in general, since the unstructured mesh might have bended elements that require the Jacobian be re-computed every i iteration. 2) The stiffness matrix A is always the result of an outer-product computation along the j and k dimensions. 3) Memory accesses along the three loop dimensions are always 1-stride .4) The j and k loops are interchangable, whereas to permute i it might be necessary to precompute F values and to introduce temporaries. 5) Basis functions arrays (denoted by X, Y, ...) are constants, and many sub-expressions on the right hand side of the stiffness matrix computation depend on just two loops (either i-j or i-k). In Section III we show how to exploit these observations to define a set of systematic, composable optimizations.

III. Domain-Driven Code Transformations

A. Auto-vectorizable Loop-invariant Code Motion

From inspection of the codes in Figures 1 and 2, it can be noticed that the computation of A involves evaluating many sub-expressions that depend on two iteration variables only. Since symbols in most of these sub-expressions are read-only variables, there is ample space for loop-invariant code motion ($licm$). Vendor compilers apply $licm$, although not in the systematic way we need in our assembly kernels. We want to overcome two deficiencies that both icc and gcc have. First, these compilers can identify sub-expressions that are invariant with respect to the innermost loop only. This is an issue for sub-expressions depending on i-k only, which are not automatically lifted. Second, the hoisted code is scalar, i.e. it is not subjected to auto-vectorization. We work around these limitations by doing source-level $licm$. In addition, we pre-compute all values that an invariant sub-expression assumes along the fastest varying dimension. This is implemented by introducing temporary arrays and new loops in the nest. At the price of both extra memory for storing temporaries and reduced register locality, the gain is that lifted terms can be auto-vectorized, because part of an inner loop. Note that given the short trip counts of our loops, it is important to achieve auto-vectorization of hoisted terms in order to minimize the percentage of scalar instructions, which can be otherwise significant.

B. Padding and Data Alignment

Auto-vectorization of assembly code that computes A can be less effective if data are not aligned and if the length of the innermost loop is smaller than the vector length (vl). Data alignment is enforced in two steps. Initially, both arrays and matrices are allocated to addresses that are multiples of vl. Then, matrices are padded by rounding the number of columns to the nearest multiple of vl. For example, assume the original size of a matrix is 3×3, and the underlying machine possesses AVX, which implies $vl = 4$ since a vector register is 256 bits long and our kernels use double-precision floating-point values (64 bits). Then, a padded matrix on this architecture will be 3×4. The compiler is explicitly informed about data alignment using a suitable pragma. Padding of all matrices involved in the evaluation of the stiffness matrix allows us to safely round the loop trip count to the nearest multiple of vl. This avoids the introduction of a remainder (scalar) loop from the compiler, which would be responsible for inefficient vectorization.

C. Toward Improving Register Locality

One notable problem assembly kernels are exposed to concerns register allocation and register locality. The critical situation occurs when loop trip counts and number of accessed matrices along the inner dimensions are such that available vector registers are under-utilized. Due to a kernel's working set fitting the L1 cache, it is remarkably important to optimize register management in order to maximize performance.

Canonical optimizations, such as loop interchange, unroll, unroll-and-jam and register tiling are typically employed to deal with this problem. In our compiler we support these optimizations either by means of explicit source code transformation (interchange, register tiling, unroll-and-jam) or indirectly advising the compiler through standard pragmas (unroll). With register tiling, in particular, we slice the innermost loop into rectangular blocks of identical size (except the "reminder" block), and we apply unrolling to individual blocks to expose ILP.

D. Outer-product Vectorization

Register tiles along both inner dimensions can be determined as well. Our domain-driven strategy, based on the observation that the evaluation of the stiffness matrix can be reduced to a summation of outer products along the j and k dimensions, aims at optimizing register reuse, in terms of both minimization of load operations and achievement of larger tiles, over a canonical implementation. If we consider the code snippet in Figure 3, we can notice that the computation of A is abstractly expressible as

$$A_{jk} = \sum_{\substack{x \in B' \subseteq B \\ y \in B'' \subseteq B}} x_j \cdot y_k \qquad j,k = 0,...,4 \tag{1}$$

where B is the set of all matrices (or temporaries) accessed in the kernel, whereas B' and B'' are generic problem-dependent subsets. Without loss of generality, the presence of other constant or i-dependent variables can be momentarily neglected. Note that regardless of the specific input problem, the assembly computation is always reducible to this form. Figure III-D illustrates how we can evaluate 16 elements ($j,k = 0,...,4$) of the stiffness matrix with just 2 vector registers. Once the first four entries are evaluated, values in a register are shuffled and the same product is performed again. Standard compiler auto-vectorization (*gcc* and *icc*), instead, would employ 5 registers, 4 of them target of a broadcast operation (i.e. "splatting" a single value over the four locations of an AVX register) along the outer dimension, and 1 of a load operation along the inner dimension. In our outer-product vectorization (*op-vect*), the 4 broadcast accesses to the L1 cache are avoided by an initial load and by shuffling. More importantly, we avoid using vl-1 registers per each outer-loop-dependent variable (i.e. all x terms in Equation 1), with vl being the vector length. This is a considerable gain, which allows us to slice the iteration space into bigger tiles, implemented directly by vector registers.

The storage layout of A, however, is incorrect after the application of *op-vect*. We efficiently restore it with a sequence of vector shuffle following the pattern highlighted in Figure III-D, executed just once outside of the ijk loop nest. The generated pseudo-code for the simple Helmholtz problem when using *op-vect* is shown in Figure 5.

E. Assembly splitting

In complex kernels, like Burgers in Figure 2, and on certain architectures, achieving effective register allocation can be challenging. If the number of variables independent of the innermost-loop dimension, i.e. basis function matrices, temporaries introduced by $licm$ and constants, is larger than the available registers on the CPU, it is likely to obtain poor register reuse. For example, applying $licm$ to Burgers on a 3D mesh needs 33 temporaries for the ijk loop order, and compiler's hoisting of invariant loads out of the k loop can be inefficient on architectures with a relatively low number of registers. One potential solution to this problem consists of suitably "splitting" the computation of the stiffness matrix A into multiple sub-expressions; an example, for the simpler Helmholtz problem, is given in Figure 4. Splitting an expression has, in general, several drawbacks. Firstly, it increases the number of accesses to A proportionally to the "split factor", which is the number of sub-expressions produced. Secondly, depending on how the split is executed, it can lead to redundant computation (e.g. the product $det * W3[i]$ is performed times number of sub-expressions in the code of Figure 4). Finally, it might affect register locality, although this is not the case of the Helmholtz example: for instance, the same matrix could be accessed in different sub-expressions, requiring a proportional number of loads be performed. Nevertheless, as shown in Section V, the performance gain from improved register reuse along inner dimensions can still be greater, especially if the split factor and the splitting itself use heuristics to minimize the aforementioned issues.

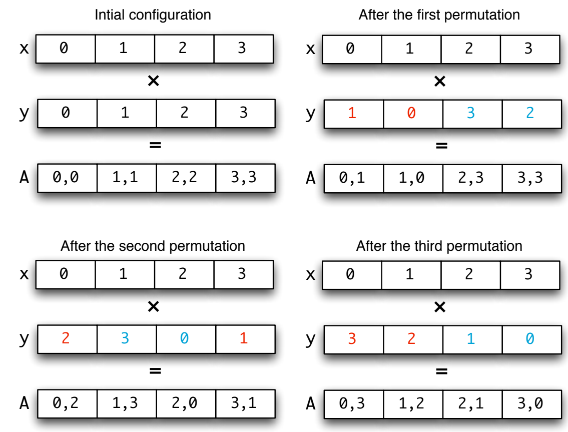

Fig. 1. Outer-product vectorization by permuting values in a vector register.

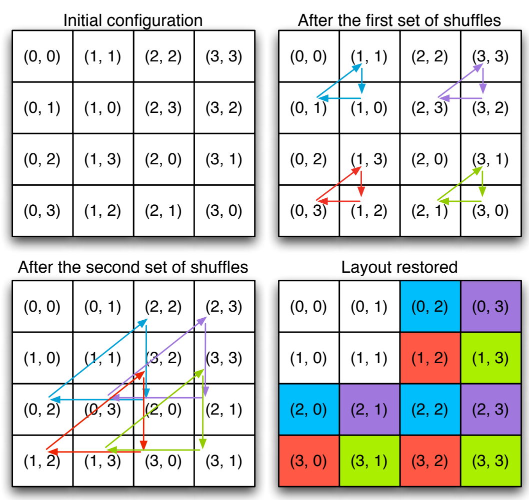

Fig. 2. Restoring the storage layout after *op-vect*.

```
void helmholtz(double A[8][8], double **coords) {
  // K, det = Compute Jacobian (coords)
  // Declaration of basis function matrices

  for (int i = 0; i<6; i++) {
    // Do loop-invariant code motion
    for (int j = 0; j<4; j+=4)
      for (int k = 0; k<8; k+=4) {
        // Call Load and set intrinsics
        // Compute A[1,1],A[2,2],A[3,3],A[4,4]
        // One permute_pd intrinsics per k-loop load
        // Compute A[1,2],A[2,1],A[3,4],A[4,3]
        // One permute2f128_pd intrinsics per k-loop load
        // ...
      }
    // Do Remainder loop (from j = 4 to j = 6)
  }
  // Restore the storage layout:
  for (int j = 0; j<4; j+=4) {
    __m256d r0, r1, r2, r3, r4, r5, r6, r7;
    for (int k = 0; k<8; k+=4) {
      r0 = _mm256_load_pd (&A[j+0][k]);
      // Load A[j+1][k], A[j+2][k], A[j+3][k]
      r4 = _mm256_unpackhi_pd (r1, r0);
      r5 = _mm256_unpacklo_pd (r0, r1);
      r6 = _mm256_unpackhi_pd (r2, r3);
      r7 = _mm256_unpacklo_pd (r3, r2);
      r0 = _mm256_permute2f128_pd (r5, r7, 32);
      r1 = _mm256_permute2f128_pd (r4, r6, 32);
      r2 = _mm256_permute2f128_pd (r7, r5, 49);
      r3 = _mm256_permute2f128_pd (r6, r4, 49);
      _mm256_store_pd (&A[j+0][k], r0);
      // Store A[j+1][k], A[j+2][k], A[j+3][k]
    }
  }
}
```

Algorithm 5: Local assembly code generated by Firedrake for a Helmholtz problem (2D mesh, Lagrange $p = 2$ elements). The padding, data alignment, $licm$ and *op-vect* optimizations are applied. Data alignment and padding relate to an AVX machine. In this specific case, the unroll-and-jam factor is 1.

Table I summarizes the code transformations described so far. Given that many of these transformations depend on some parameters (e.g. tile size), we need a mechanism to prune such a large space of optimization. This aspect is treated in Section IV

Name (Abbreviation)	Parameter
Auto-vectorizable Loop-invariant code motion ($licm$)	
Padding	
Data Alignment	
Loop interchange	loops
Loop unrolling	unroll factor
Register tiling	tile size
Outer-product vectorization (op-$vect$)	tile size
Assembly splitting ($split$)	split point, split factor

TABLE I. OVERVIEW OF CODE TRANSFORMATIONS FOR FIREDRAKE-GENERATED ASSEMBLY KERNELS.

IV. THE PYOP2 COMPILER

Firedrake provides users with the Unified Form Language to write problems in a notation resembling mathematical equations. This high-level specification is translated by the domain-specific Fenics Form Compiler [?] into an Abstract Syntax Tree representation of a Finite Element assembly kernel. ASTs are then passed to PyOP2, where parallel execution over the unstructured mesh is managed. Our compiler, capable of applying the transformations described in Section III, is integrated with PyOP2: it can manipulate ASTs introducing optimizations, and then it generates C code, which is eventually just-in-time compiled on the underlying architecture.

The compiler structure is outlined in Figure 6. Initially, an AST is inspected, looking for the presence of iteration spaces and other domain-specific information provided by the higher layer. If the kernel lacks an iteration space, then so-called inter-kernel vectorization, in which the non-affine loop over mesh elements is vectorized, can be applied. This feature, currently under development, has been proved to be useful in several Finite-Volume-based applications [?]. The second transformation step is applied if the backend is a GPU: the compiler tries to extract parallelism from inside the kernel, by partitioning loop iterations among different threads, if these are found to be independent [?]. Then, a sequence of optimization steps are executed. A cost model is queried to estimate the potential gain if the $split$ and op-$vect$ transformations are applied. Optimization passes must follow a specific order. Application of $licm$ must precede padding and data alignment, due to the introduction of temporary arrays. Then, $split$ might take place, depending on the outcome of the cost model. The op-$vect$ optimization is introduced lastly. Note that the compiler can handle all possible corner cases: for example, if op-$vect$ is to be applied but the size of the iteration space is not a multiple of the vector length, then a reminder loop, amenable to auto-vectorization, is inserted.

Our experience... unrolling/unroll and jam

The cost model ..... TODO

```
The PyOP2 Compiler
Input: ast, wrapper, isa
Output: code
// Analyze ast and build optimization plan
it_space = analyze(ast)
if not it_space then
    ast.apply_inter_kernel_vectorization(wrapper)
    return wrapper + ast.from_ast_to_c()
end if
if isa.backend == gpu then
    if it_space then
        ast.extract_iteration_space(wrapper)
    end if
    return wrapper + ast.from_ast_to_c()
end if
plan = cost_model(it_space.n_inner_arrays, isa.n_regs)
// Optimize ast based on plan
ast.apply_licm()
ast.padding()
ast.data_align()
if plan.sz_split then
    ast.split(plan.sz_split)
end if
if plan.uaj_factor then
    ast.vr_tile(plan.uaj_factor)
end if
return wrapper + ast.from_ast_to_c()
```

Algorithm 6: The PyOP2 compiler.

```
Algorithm: ApplyCostModel
Input: n_inner_arrays, n_regs
Output: uaj_factor, sz_split
sz_split = 0
// Compute spltting factor
while n_inner_arrays > n_regs do
    n_inner_arrays = n_inner_arrays / 2
    sz_split = sz_split + 1
end while
n_inner_regs = n_regs / 2
if n_inner_arrays > n_inner_regs or sz_split > 0 then
    // Rely on autovectorization, as there might be not
    enough registers to improve performance by tiling
    return <sz_split, 0>
end if
// Compute unroll-and-jam factor for vector-register
tiling
n_regs_avail = n_regs - n_inner_arrays
uaj_factor = ⌈n_reg_avail / n_inner_arrays⌉
return <sz_split, uaj_factor>
```

Algorithm 7: Procedure to estimate the most suitable unroll-and-jam factor and/or split size.

V. Performance Evaluation

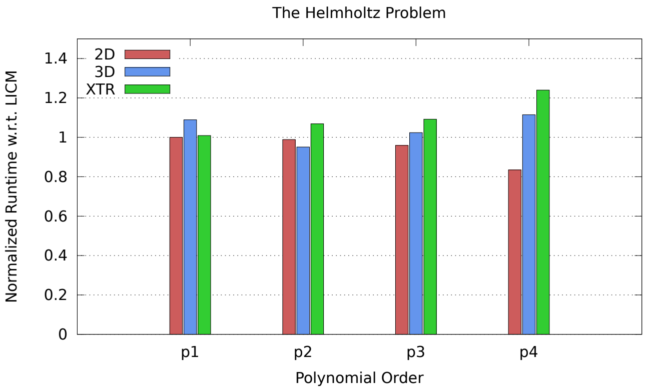

Fig. 3. Helmholtz

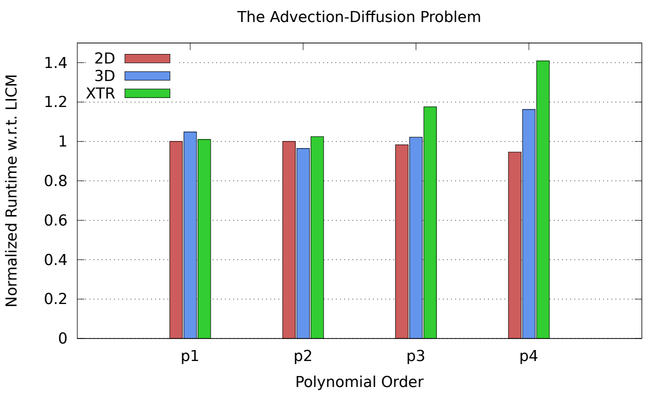

Fig. 4. Advection-Diffusion

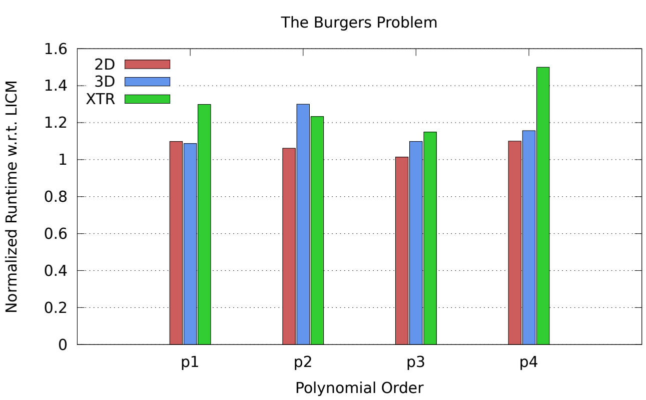

Fig. 5. Burgers

VI. Related Work

VII. Conclusions

Acknowledgment

The authors would like to thank...

References

[1] The firedrake project, 2014.

[2] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. Liszt: A domain specific language for building portable mesh-based pde solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 9:1–9:12, New York, NY, USA, 2011. ACM.

[3] G. R. Markall, F. Rathgeber, L. Mitchell, N. Loriant, C. Bertolli, D. A. Ham, and P. H. J. Kelly. Performance portable finite element assembly using PyOP2 and FEniCS. In *Proceedings of the International Supercomputing Conference (ISC) '13*, volume 7905 of *Lecture Notes in Computer Science*, June 2013. In press.

[4] Kristian B. Olgaard and Garth N. Wells. Optimizations for quadrature representations of finite element tensors through automated code generation. *ACM Trans. Math. Softw.*, 37(1):8:1–8:23, January 2010.

[5] Jaewook Shin, Mary W. Hall, Jacqueline Chame, Chun Chen, Paul F. Fischer, and Paul D. Hovland. Speeding up nek5000 with autotuning and specialization. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 253–262, New York, NY, USA, 2010. ACM.