COFFEE: AN OPTIMIZING COMPILER FOR FINITE ELEMENT LOCAL ASSEMBLY

Fabio Luporini

Florian Rathgeber, Lawrence Mitchell, Gheorghe-Teodor Bercea, Andrew T. T. McRae, J. Ramanujam, Ana Lucia Varbanescu, David A. Ham, and Paul H. J. Kelly

Imperial College London

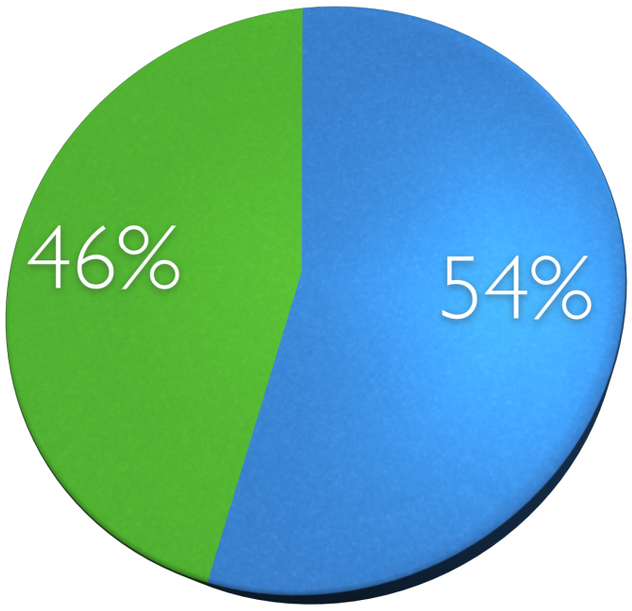
16/06/2014

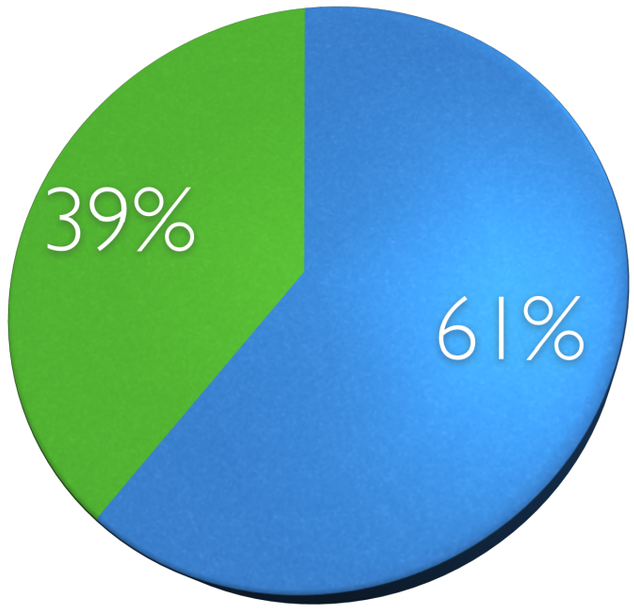
FEniCS '14

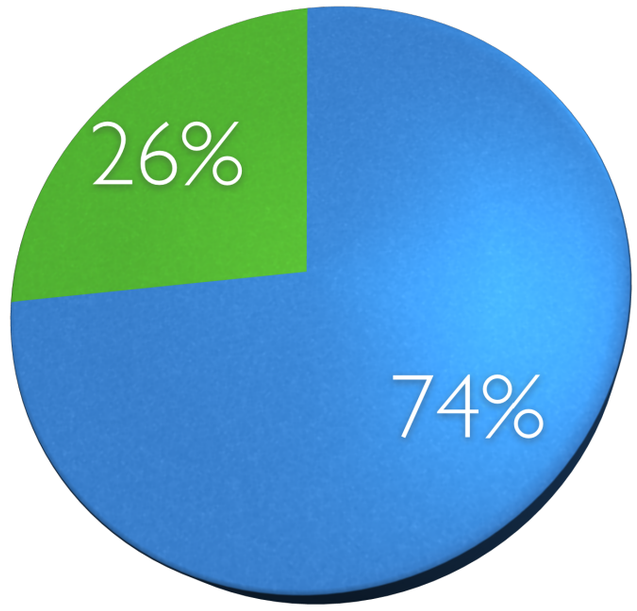
COFFEE: AN OPTIMIZING COMPILER FOR FINITE ELEMENT LOCAL ASSEMBLY

INDICATIVE COST OF LOCAL ASSEMBLY IN AN FEM

Execution time ≈ Assembly + Solver







COFFEE: AN OPTIMIZING COMPILER FOR FINITE ELEMENT LOCAL ASSEMBLY

+

COFFEE: AN OPTIMIZING COMPILER FOR FINITE ELEMENT LOCAL ASSEMBLY

THE PHILOSOPHY: SEPARATING MATH FROM CODE OPTIMIZATION

- FEniCS Form Compiler (FFC): Quadrature, Tensor Contraction, UFLACS (Martin, in progress)

- **Firedrake** also based on FFC; it currently supports **Quadrature,** optimized through **COFFEE** (rather than using FFC's built-in optimizations)

- **FFC takes UFL ("the math") and produces an abstract representation of assembly, while COFFEE handles platform-specific code optimization exploiting domain knowledge**

COFFEE'S OBJECTIVES:

Given a **"tabulate_tensor()"** function (LHS/RHS of a form):

- **Minimization of floating-point operations**
  - **Expression Rewriter**
- **Optimization of Instruction Level Parallelism**
  - **SIMD (Single Instruction, Multiple Data) Vectorization**
  - **…**

EXPRESSION REWRITER

```
for i
 for j
  for k
   A[j][k] += B[i][j] * C[i][k] +
              (B[i][j] * D[i][k])*alpha
```

Illustrated through a fictitious, simplified example

- Evaluation of a LHS integral
- A ≈ element matrix
- B, C ≈ basis functions
- D ≈ derivative of C
- alpha ≈ problem-specific constant

EXPRESSION REWRITER

```
for i
 for j
  for k
   A[j][k] += B[i][j] * C[i][k] +
              (B[i][j] * D[i][k])*alpha
```

EXPRESSION REWRITER

```
for i
 for j
  for k
   A[j][k] += B[i][j] * C[i][k] +
              (B[i][j] * D[i][k])*alpha
```

```
for i
 for j
  for k
   A[j][k] += B[i][j] * C[i][k] +
              B[i][j] * (D[i][k] * alpha)
```

EXPRESSION REWRITER

```
for i
 for j
  for k
   A[j][k] += B[i][j] * (C[i][k] + D[i][k]*alpha)
```

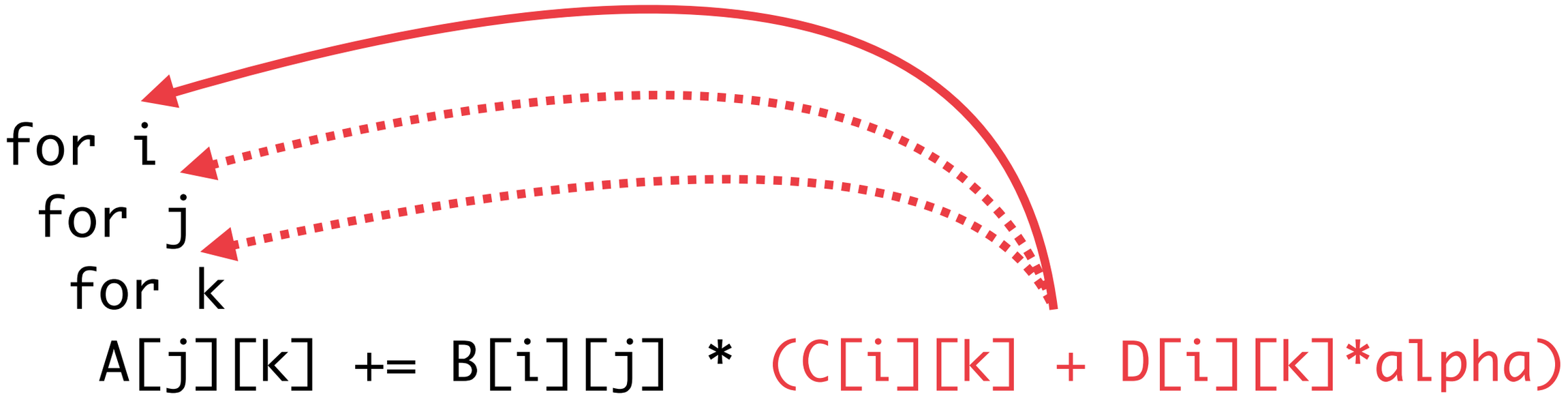
Detect presence of same array => Factorization of terms

EXPRESSION REWRITER

```
for i
 for j
  for k
   A[j][k] += B[i][j] * (C[i][k] + D[i][k]*alpha)
```

Detect invariant sub-expressions => hoist them out

EXPRESSION REWRITER



Detect invariant sub-expressions =>
hoist them out

WHERE?

A REAL COMPILER PROBLEM

```
for i
  for k
   TMP[i][k] = (C[i][k] + D[i][k]*alpha)
```

```
for i
 for j
  for k
   A[j][k] += B[i][j] * TMP[i][k]
```

A REAL COMPILER PROBLEM

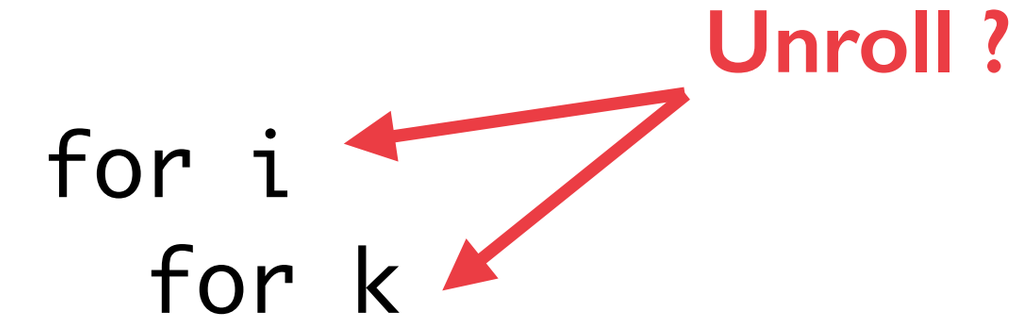
Unroll ?

```
for i
  for k
   TMP[i][k] = (C[i][k] + D[i][k]*alpha)
```

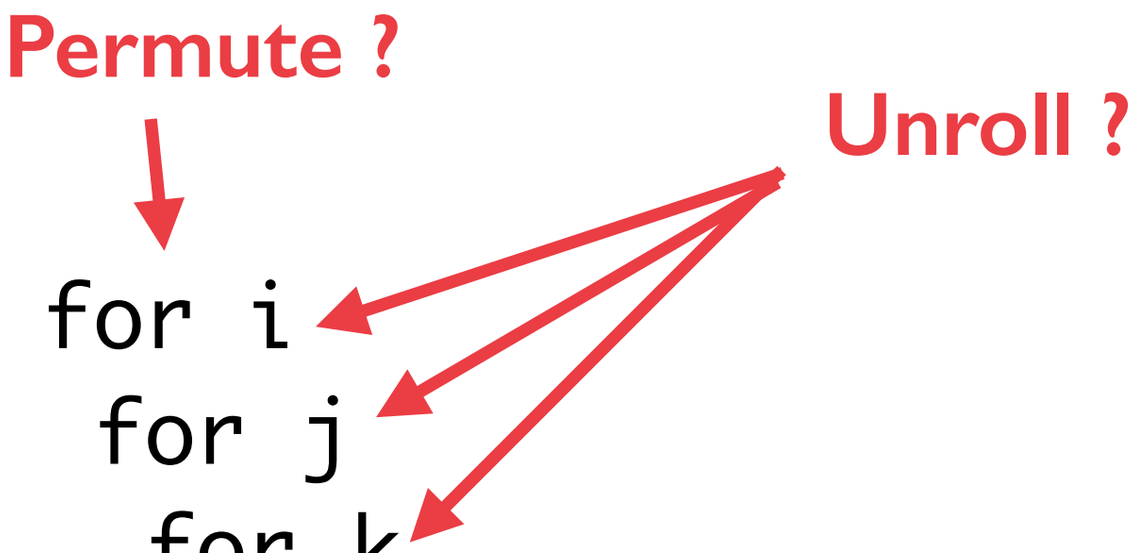
Unroll ?

```
for i
 for j
  for k
   A[j][k] += B[i][j] * TMP[i][k]
```

A REAL COMPILER PROBLEM




```
for i
  for k
   TMP[i][k] = (C[i][k] + D[i][k]*alpha)
```




```
for i
 for j
  for k
   A[j][k] += B[i][j] * TMP[i][k]
```

A REAL COMPILER PROBLEM

Unroll ?

```
for i
  for k
   TMP[i][k] = (C[i][k] + D[i][k]*alpha)
```

Permute ?

Unroll ?

```
for i
 for j
  for k
   A[j][k] += B[i][j] * TMP[i][k]
```

MMM ?

A REAL COMPILER PROBLEM

Unroll ?

Vectorize ?

```
for i
  for k
   TMP[i][k] = (C[i][k] + D[i][k]*alpha)
```

Permute ?

Unroll ?

Vectorize ?

MMM ?

```
for i
 for j
  for k
   A[j][k] += B[i][j] * TMP[i][k]
```

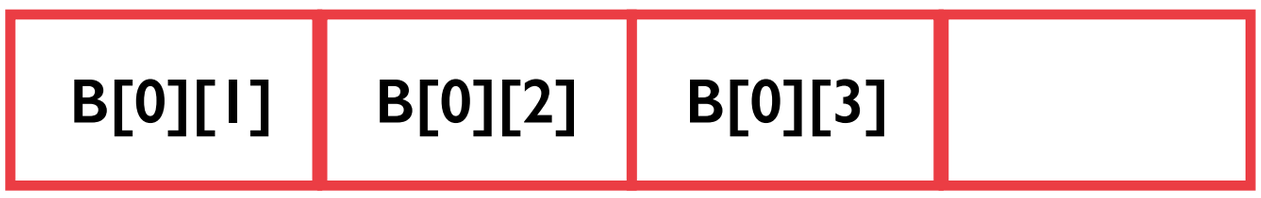
SIMD VECTORIZATION

```
double B[3][3] = {…}
double C[3][3] = {…}

for i = 0 < 3
   for k = 0 < 3
    TMP[i][k] = (C[i][k] + D[i][k]*alpha)
```

```
for i = 0 < 3
   for j = 0 < 3
      for k = 0 < 3
        A[j][k] += B[i][j] * TMP[i][k]
```

Double precision & memory stride **=>**



SIMD VECTORIZATION

```
double B[3][4] __attribute__((aligned(32))) = {…}
double C[3][4] __attribute__((aligned(32))) = {…}

for i = 0 < 3
  for k = 0 < 4
   TMP[i][k] = (C[i][k] + D[i][k]*alpha)
```

```
for i = 0 < 3
  for j = 0 < 3
    #pragma vector aligned
    #pragma simd
    for k = 0 < 4
      A[j][k] += B[i][j] * TMP[i][k]
```

Padding + Data alignment + Trip counts adjustment

SIMD VECTORIZATION

```
…
__m256 r0, r1, r2, r3 …;
for i = 0 < 3
   for j = 0 < 3
      #pragma vector aligned
      for k = 0 < 4
      r_1 = _mm256_load_pd (&B[i][j]);
      r_2 = _mm256_load_pd (&TMP[i][k]);
      r_3 = _mm256_mul_pd (r_1, r_2);
      …
```

COFFEE generates intrinsics code implementing an outer-product vectorization along the j and k loops

A REAL COMPILER PROBLEM

Unroll ?

Vectorize ?

```
for i
  for k
   TMP[i][k] = (C[i][k] + D[i][k]*alpha)
```

Permute ?

Unroll ?

Vectorize ?

MMM ?

```
for i
 for j
  for k
   A[j][k] += B[i][j] * TMP[i][k]
```

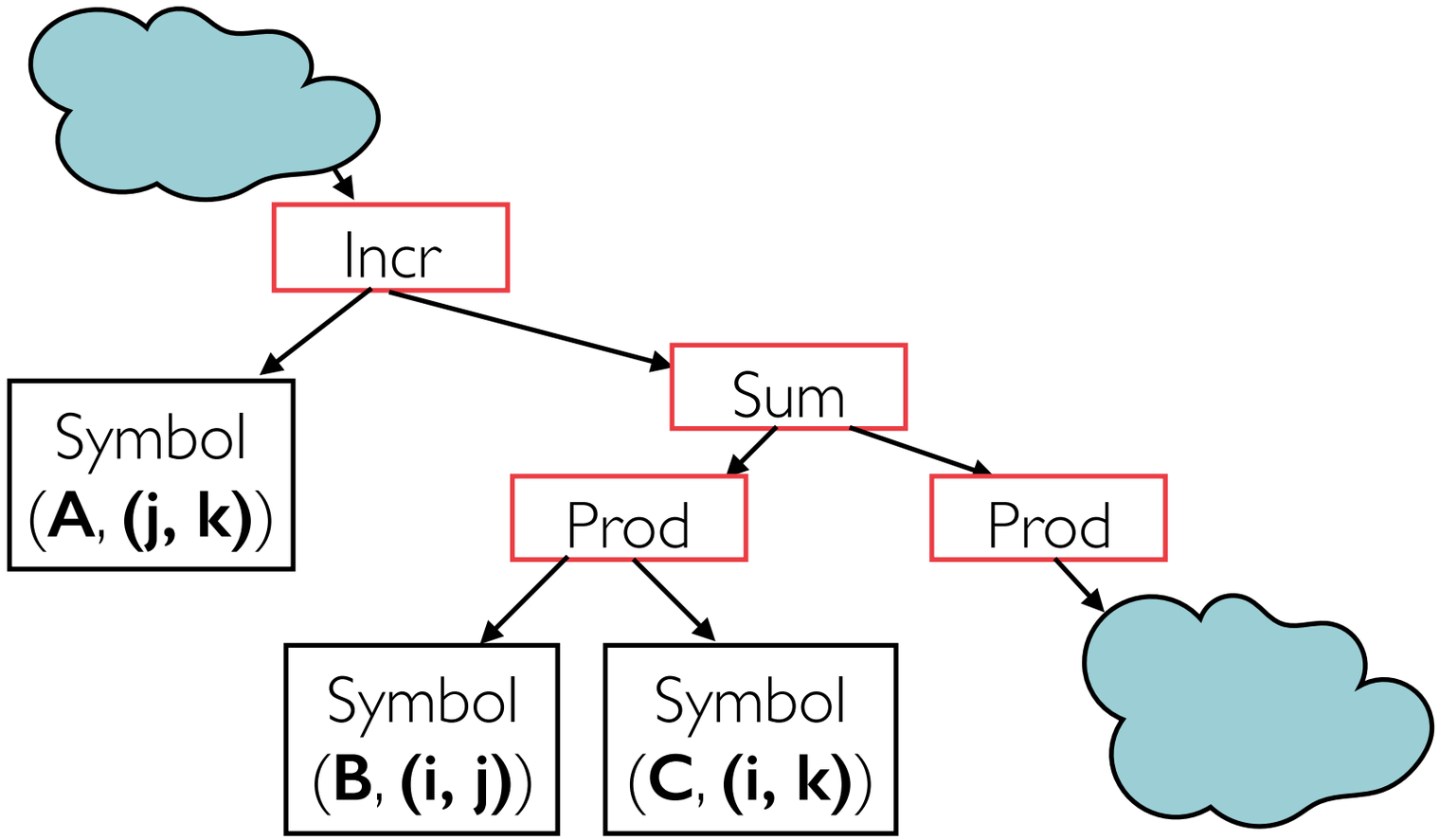
PUTTING EVERYTHING TOGETHER: **AUTOTUNING**

- **PROBLEM: MANY POSSIBLE CODE VARIANTS**
- **Test multiple code variants on a fake mesh, each variant up to X ms**
- **Just a linear search, but driven by heuristics, to keep the implementation space to explore reasonably small**

A GLIMPSE OF THE IMPLEMENTATION

```
for i
 for j
  for k
   A[j][k] += B[i][j] * C[i][k] +
              (B[i][j] * D[i][k])*alpha
```

FFC now produces ABSTRACT SYNTAX TREES!



EXPERIMENTAL RESULTS

- Single core results
- Intel Sandy Bridge (I7-2600 CPU @ 3.40GHz)
- Compiler: Intel (version 13.1)
- Elements belong to CG family
- Problems: LHS assembly of Helmholtz, Diffusion, and Burgers

EXPERIMENTAL RESULTS

		Expr Rewriter	Expr Rewriter + Vector-friendly code
		p1	p1
Helmholtz	triangle	1.05	1.32
Helmholtz	tetrahedron	1.36	1.35
Helmholtz	prism	2.16	2.63
Diffusion	triangle	1.09	1.38
Diffusion	tetrahedron	1.30	1.41
Diffusion	prism	2.15	2.55
Burgers	triangle	1.53	1.56
Burgers	tetrahedron	1.61	1.61
Burgers	prism	2.11	2.19

Speed-up over original code (no FFC optimizations used here)

EXPERIMENTAL RESULTS

		Expr Rewriter		Expr Rewriter + Vector-friendly code	
		p1	p2	p1	p2
Helmholtz	triangle	1.05	1.46	1.32	1.88
Helmholtz	tetrahedron	1.36	2.10	1.35	3.32
Helmholtz	prism	2.16	2.28	2.63	2.74
Diffusion	triangle	1.09	1.68	1.38	1.99
Diffusion	tetrahedron	1.30	2.20	1.41	3.70
Diffusion	prism	2.15	1.82	2.55	3.13
Burgers	triangle	1.53	1.81	1.56	2.28
Burgers	tetrahedron	1.61	2.24	1.61	2.10
Burgers	prism	2.11	2.20	2.19	2.32

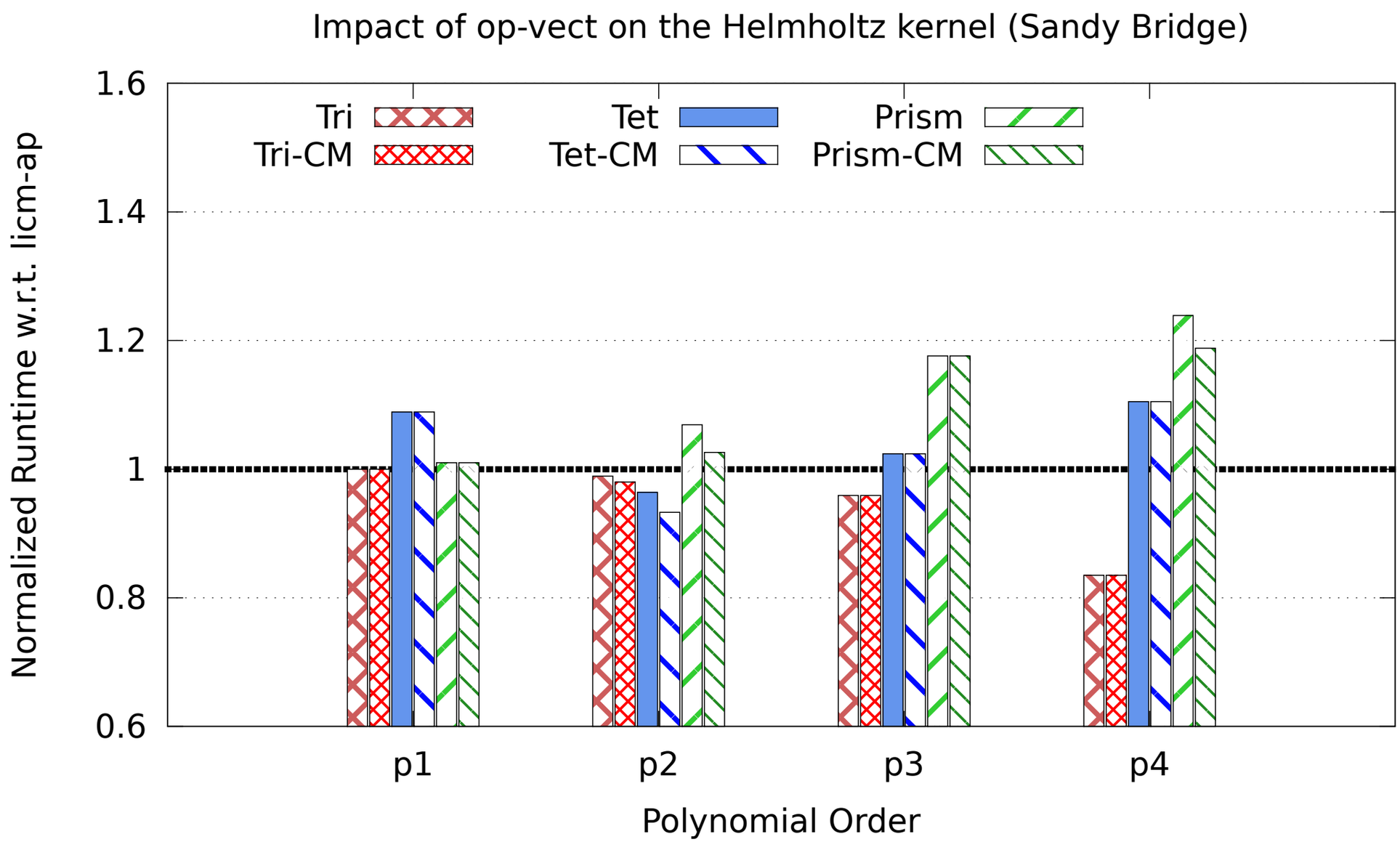
Speed-up over original code (no FFC optimizations used here)

EXPERIMENTAL RESULTS

		Expr Rewriter				Expr Rewriter + Vector-friendly code			
		p1	p2	p3	p4	p1	p2	p3	p4
Helmholtz	triangle	1.05	1.46	1.68	1.67	1.32	1.88	2.87	4.13
Helmholtz	tetrahedron	1.36	2.10	2.64	2.27	1.35	3.32	2.66	3.27
Helmholtz	prism	2.16	2.28	2.45	2.06	2.63	2.74	2.43	2.75
Diffusion	triangle	1.09	1.68	1.97	1.64	1.38	1.99	3.07	4.28
Diffusion	tetrahedron	1.30	2.20	3.12	2.60	1.41	3.70	3.18	3.82
Diffusion	prism	2.15	1.82	2.71	2.32	2.55	3.13	2.73	2.69
Burgers	triangle	1.53	1.81	2.68	2.46	1.56	2.28	2.61	2.77
Burgers	tetrahedron	1.61	2.24	1.69	1.59	1.61	2.10	1.60	1.78
Burgers	prism	2.11	2.20	1.66	1.32	2.19	2.32	1.64	1.42

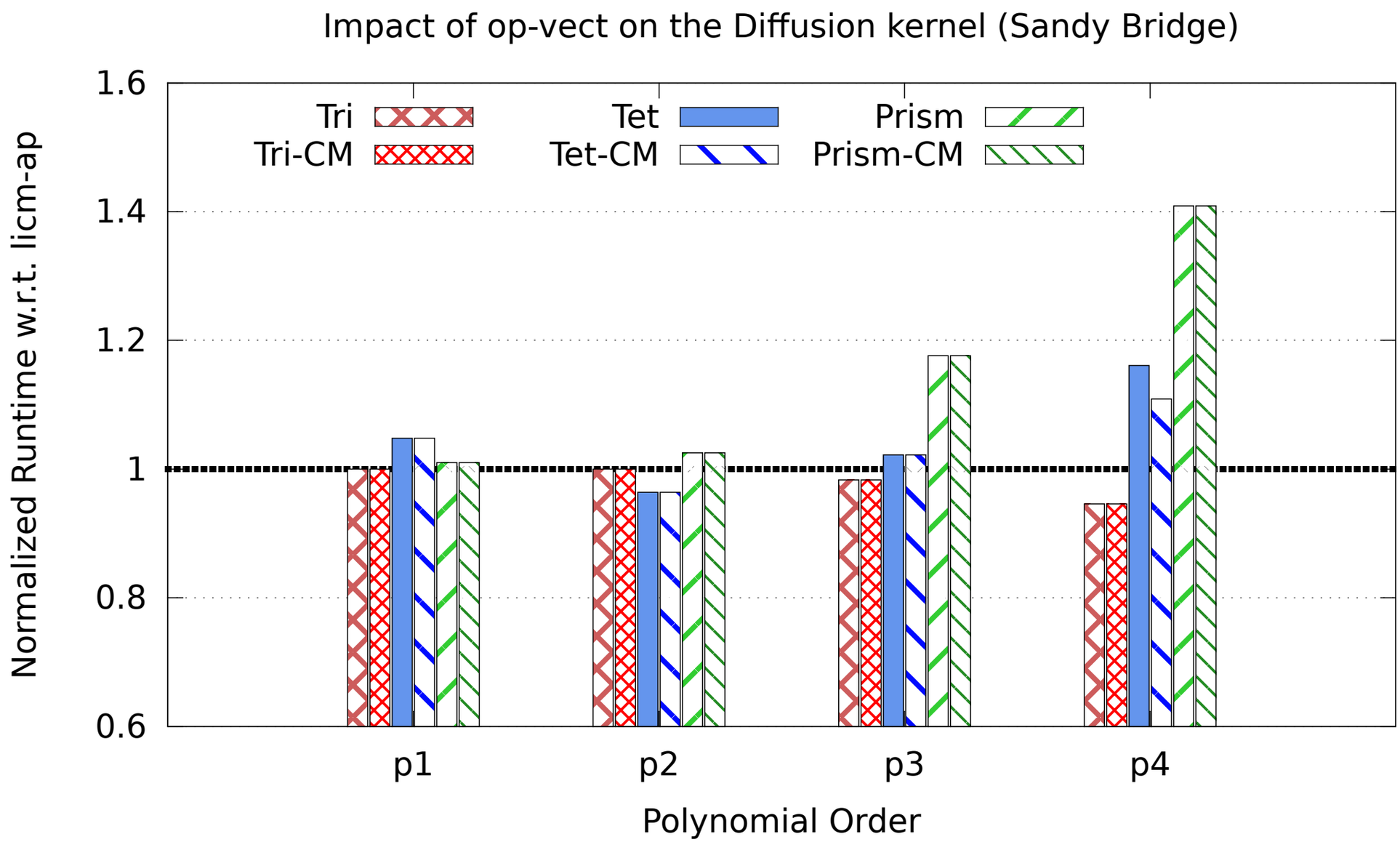
Speed-up over original code (no FFC optimizations used here)

COFFEE'S VECTORIZATION - HELMHOLTZ



COFFEE's outer-product vectorization over Expression Rewriter + vector-friendly code

COFFEE'S VECTORIZATION - **DIFFUSION**



COFFEE's outer-product vectorization over Expression Rewriter + vector-friendly code

COMPARISON WITH "BEST" DOLFIN

"Best" means the fastest run-time obtained by DOLFIN when using QUADRATURE with or without optimizations

Helmholtz: max speed-up: 4.14x
max slow-down: -

Diffusion: max speed-up: 4.28x
max slow-down: -

Burgers: max speed-up: 2.24x
max slow-down: 1.61x (p1, zeros)

Many more results: more forms, with/without pre-multiplying functions, gcc compiler, absolute run-times, …: OFFLINE!

SUMMARY

- **COFFEE is a real compiler, fully integrated with Firedrake, capable of optimizing FFC code**

- **Already evaluated for a set of forms, architectures, and compilers. But we are all keen to try it with more complex forms (e.g. hyperelasticity)**

- **Hopefully, source of inspiration for FEniCS/FFC development**

- **Try COFFEE at the developers session tomorrow**

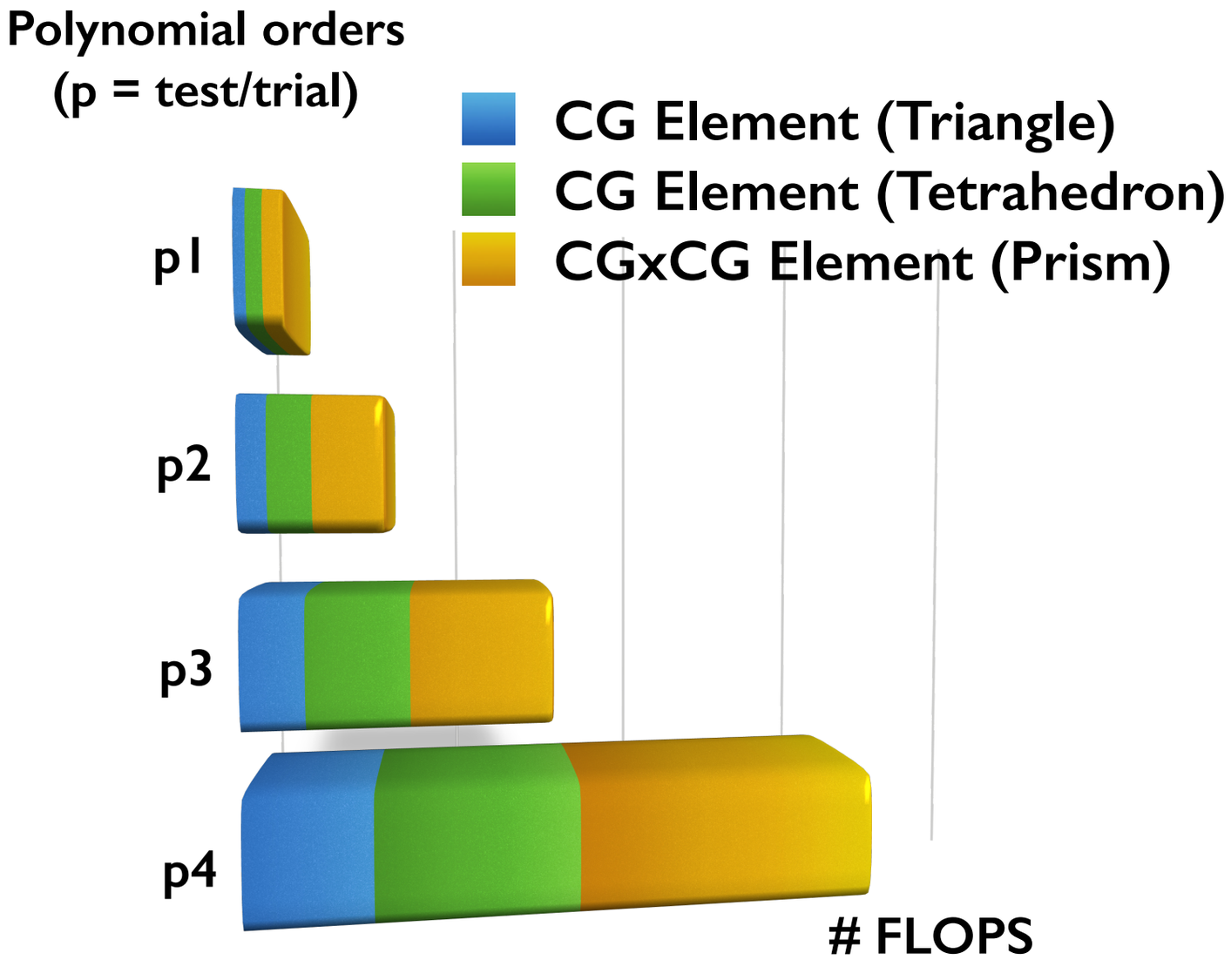
THANKS!

(and COFFEE break now!)

SPARE SLIDES

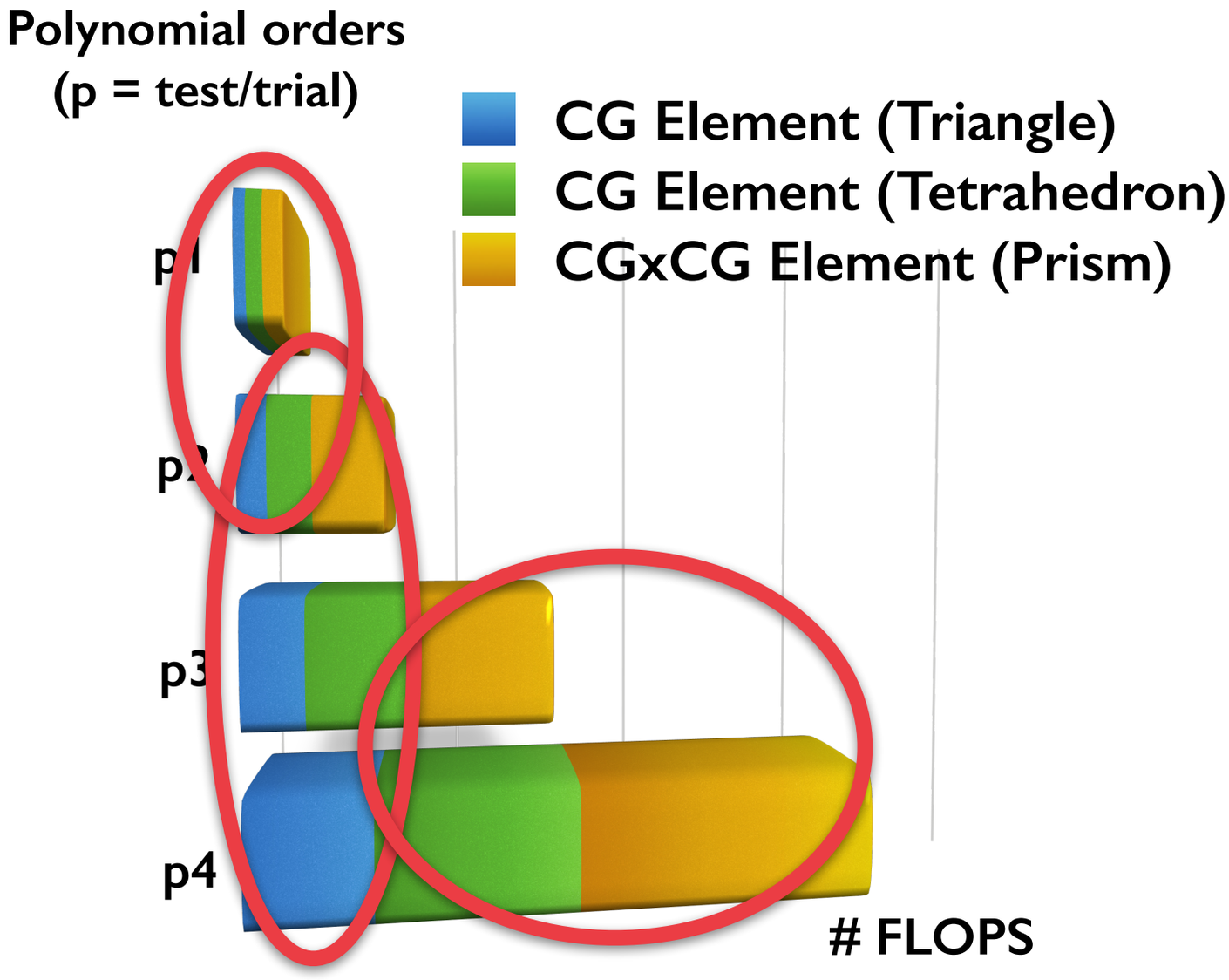
CHALLENGE WHEN OPTIMIZING **QUADRATURE-BASED** LOCAL ASSEMBLY

Differential operators and function spaces impact the computational cost of local assembly kernels



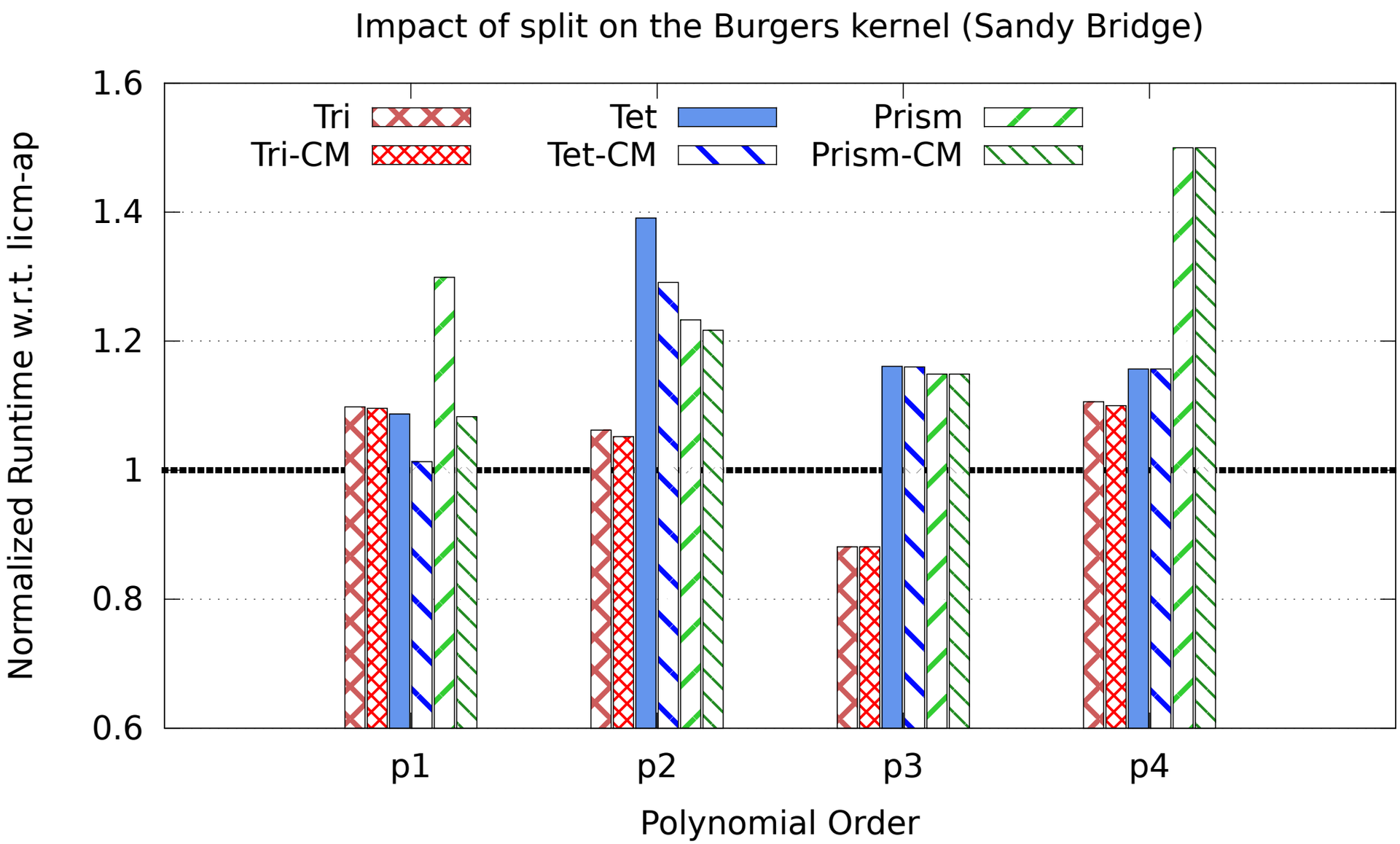
CHALLENGE WHEN OPTIMIZING **QUADRATURE-BASED** LOCAL ASSEMBLY

Differential operators and function spaces impact the computational cost of local assembly kernels



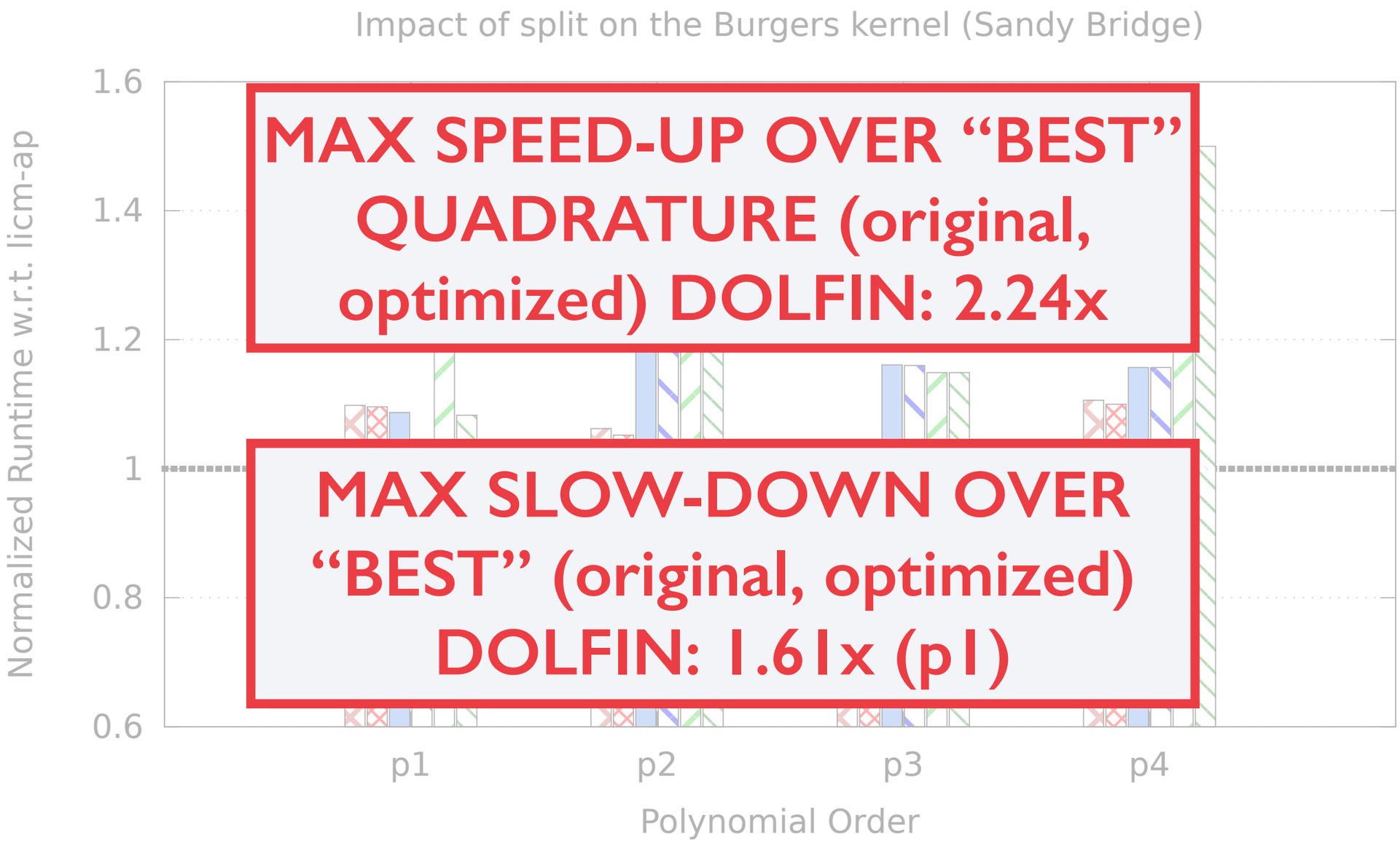
Depending on the "complexity" of a kernel, different sets of code transformations are needed to maximize performance

EXPERIMENTAL RESULTS - BURGERS



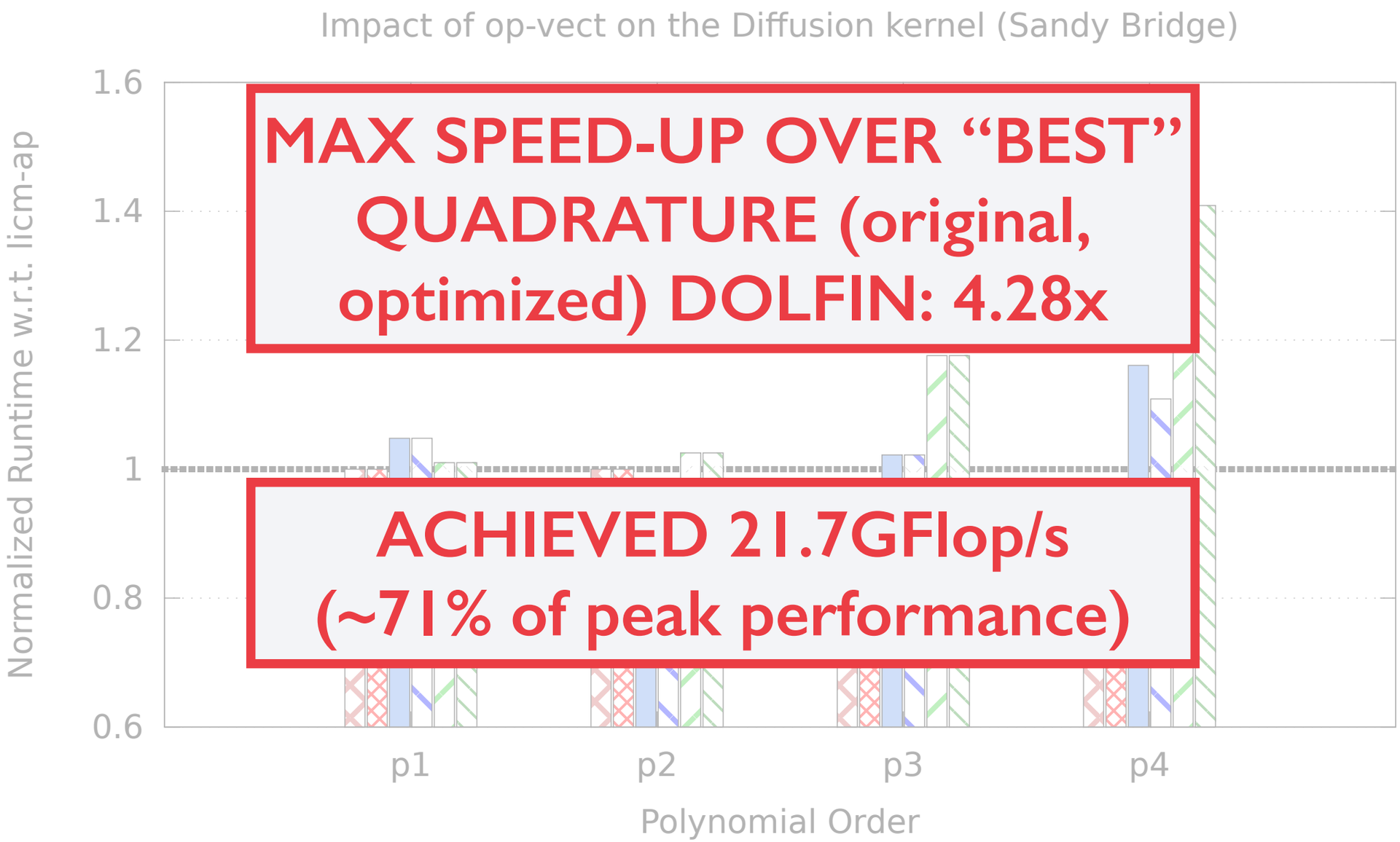
COFFEE's "Loop fission" (based on +'s associativity) over Expression Rewriter + vector-friendly code

EXPERIMENTAL RESULTS - BURGERS



COFFEE's "Loop fission" (based on +'s associativity) over Expression Rewriter + vector-friendly code

EXPERIMENTAL RESULTS - DIFFUSION



COFFEE's outer-product vectorization over Expression Rewriter + vector-friendly code