Data Layout Transformation for Stencil Computations on Short-Vector SIMD Architectures

Tom Henretty[1] Kevin Stock[1] **Louis-Noël Pouchet**[1] Franz Franchetti[2]
J. Ramanujam[3] P. Sadayappan[1]

[1] **The Ohio State University**
[2] **Carnegie Mellon University**
[3] **Louisiana State University**

March 29, 2011
ETAPS CC'11
Saarbrucken, Germany

Outline

Short-Vector SIMD

- Perform identical computation on small chunks of data
  - Operations are independent
  - Vector size: from 2 to 64
  - Packing operations to form a vector (shuffle, extract, ...)
- Low latency, multiple SIMD units per CPU
  - Maximal Speedup equals the vector size
- Ubiquitous feature on modern processors
  - x86 – SSE, AVX
  - Power – VMX / VSX
  - ARM – NEON
  - Cell SPU

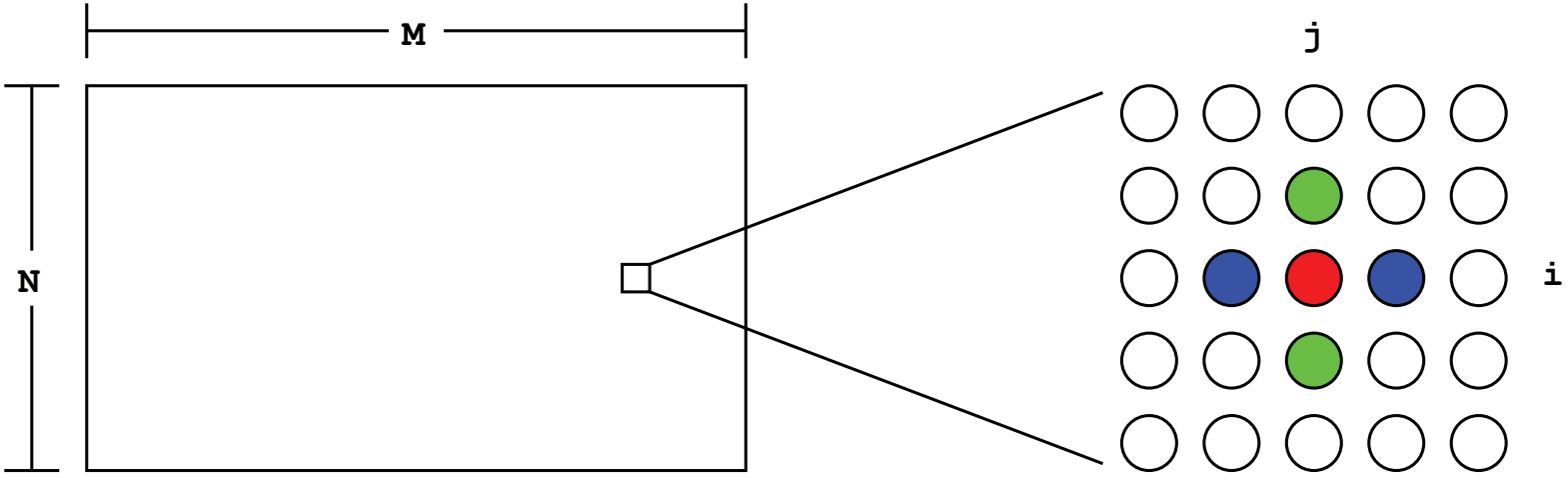
A Brief on Stencil Computations

- Typically: iterative update of a structured (fixed) grid
- Compute a point from neighbor points values
  - Same grid / multiple grids
- Numerous application domains use stencils
  - Finite difference methods for solving PDEs
  - Image processing
  - Computational electromagnetics, CFD, numerical relativity, ...
- Domain-Specific Languages for Stencils (Fenics, RNPL, ...)

Stencil Example

(a) 5 point stencil C code

```
for (t = 0; t < TMAX; ++t)
  for (i = 1; i < N-1; ++i)
    for (j = 1; j < M-1; ++j)
      a[i][j] =           b[i+1][j] +
                b[i][j-1] + b[i  ][j] + b[i][j+1] +
                            b[i-1][j];
```



(b) Arrays a, b, and stencil detail

Vectorization of Stencil Computation

- Two "main" types of stencils
  - Jacobi-like: the output does not depend on the input
  - Seidel-like: in-place update
- Loop transformations expose tiling possibilities, and at least one inner-most parallel loop

- Auto-vectorization successful (ICC, GCC)...
- ...But SIMD speedup is far from optimal!

Performance Consideration

```
for (t = 0; t < T; ++t) {
  for (i = 0; i < N; ++i)
    for (j = 1; j < N+1; ++j)
S1:   C[i][j] = A[i][j] +  A[i][j-1];
  for (i = 0; i < N; ++i)
    for (j = 1; j < N+1; ++j)
S2:   A[i][j] =  C[i][j] +  C[i][j-1];
}
```

Performance:	AMD Phenom	1.2 GFlop/s
	Core2	3.5 GFlop/s
	Core i7	4.1 GFlop/s

(a) Stencil code

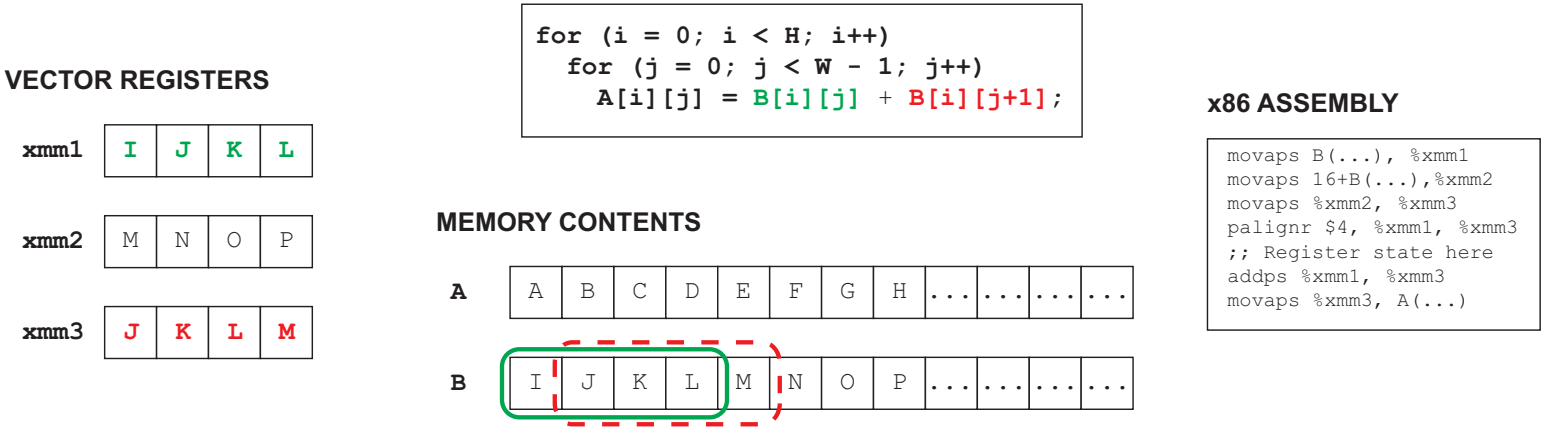
```
for (t = 0; t < T; ++t) {
  for (i = 0; i < N; ++i)
    for (j = 0; j < N; ++j)
S3:   C[i][j] =  A[i][j] +  B[i][j];
  for (i = 0; i < N; ++i)
    for (j = 0; j < N; ++j)
S4:   A[i][j] =  C[i][j] +  B[i][j];
}
```

Performance:	AMD Phenom	**1.9** GFlop/s
	Core2	**6.0** GFlop/s
	Core i7	**6.7** GFlop/s

(b) Non-Stencil code

Stencil code (a) has much lower performance than the non-stencil code (b) despite accessing 50% fewer data elements

Stream Alignment Conflict

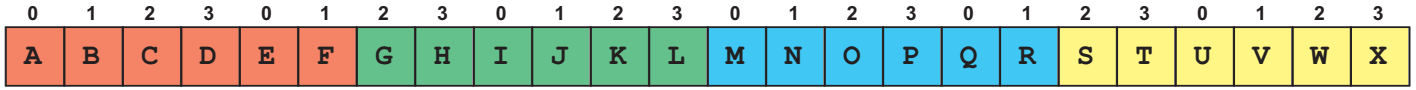


- Load and shuffle:
  - Load `[I,J,K,L]` and `[M,N,O,P]`
  - Shuffle to create `[J,K,L,M]`
- Multiple unaligned loads
  - Load `[I,J,K,L]` and `[J,K,L,M]`
  - Not possible on architectures with alignment constraints

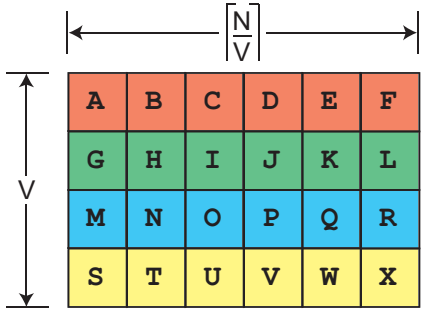
Overview of the Solution

- Stream Alignment Conflict: adjacent elements in memory maps to adjacent vector slots
- Key idea: break this property, to have both operands in identical vector slot
- Achieved through Data Layout Transformation
  - No shuffle needed
  - No extra unaligned load
  - But not trivial to achieve!

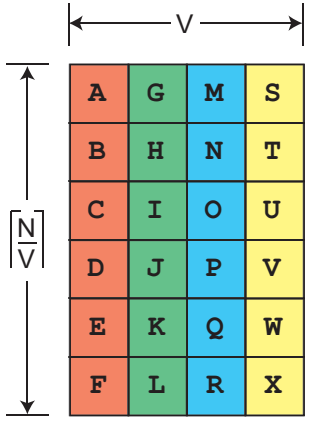
Data Layout Transformation Example



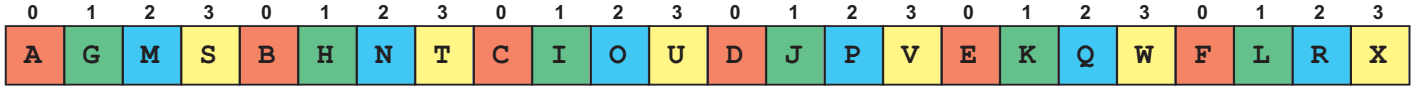
(a) Original Layout



(b) Dimension Lifted



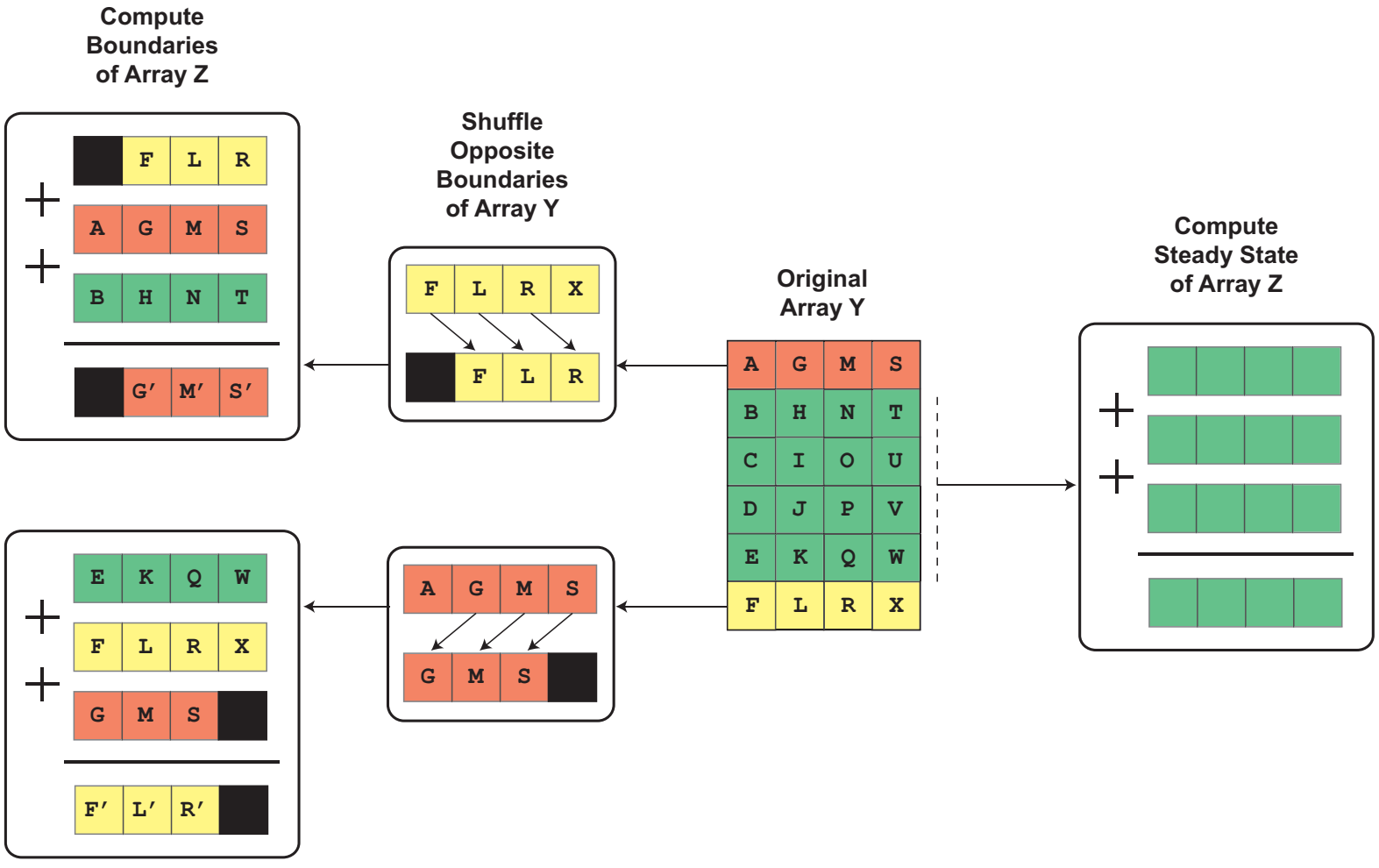
(c) Transposed



(d) Transformed Layout

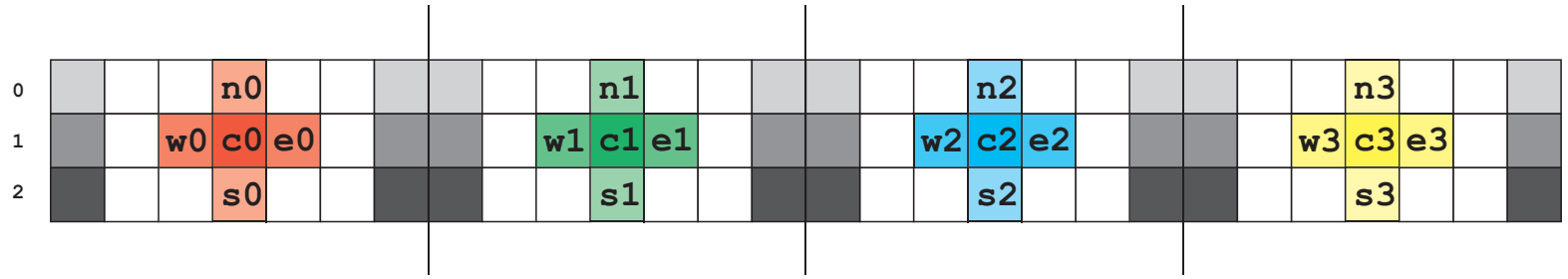
```
for (i = 1; i < 24; ++i)
  B[i] = (A[i-1] + A[i] + A[i+1]) / 3;
```

Handling Boundaries

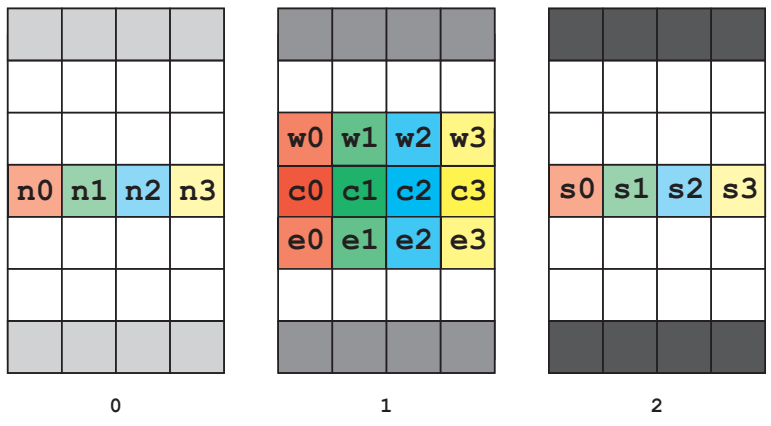


Higher-dimensional Stencils

(a) Original Layout



(b) Transformed Layout



Overview of Code Generation Algorithm

1. Detect arrays/statements that suffers from SAC
2. Perform Dimension-Lift-and-Transpose of those arrays
3. Generate Vector code for the inner-loop
   - Ghost cell copy-in and copy-out code
   - Boundary code
   - Steady state code

Detection of Stream Alignment Conflict

- Standard compiler framework operating on array subscript functions
- Main idea: detect cross-iteration reuse
- Robust to stream offset via iteration shifting
  - Minimize the reuse distance
  - Some alignment conflicts are artificial and fixed with stream realignment
- Requires the window of the stencil to be constant
  - The window size is used to compute the amount of ghost cells

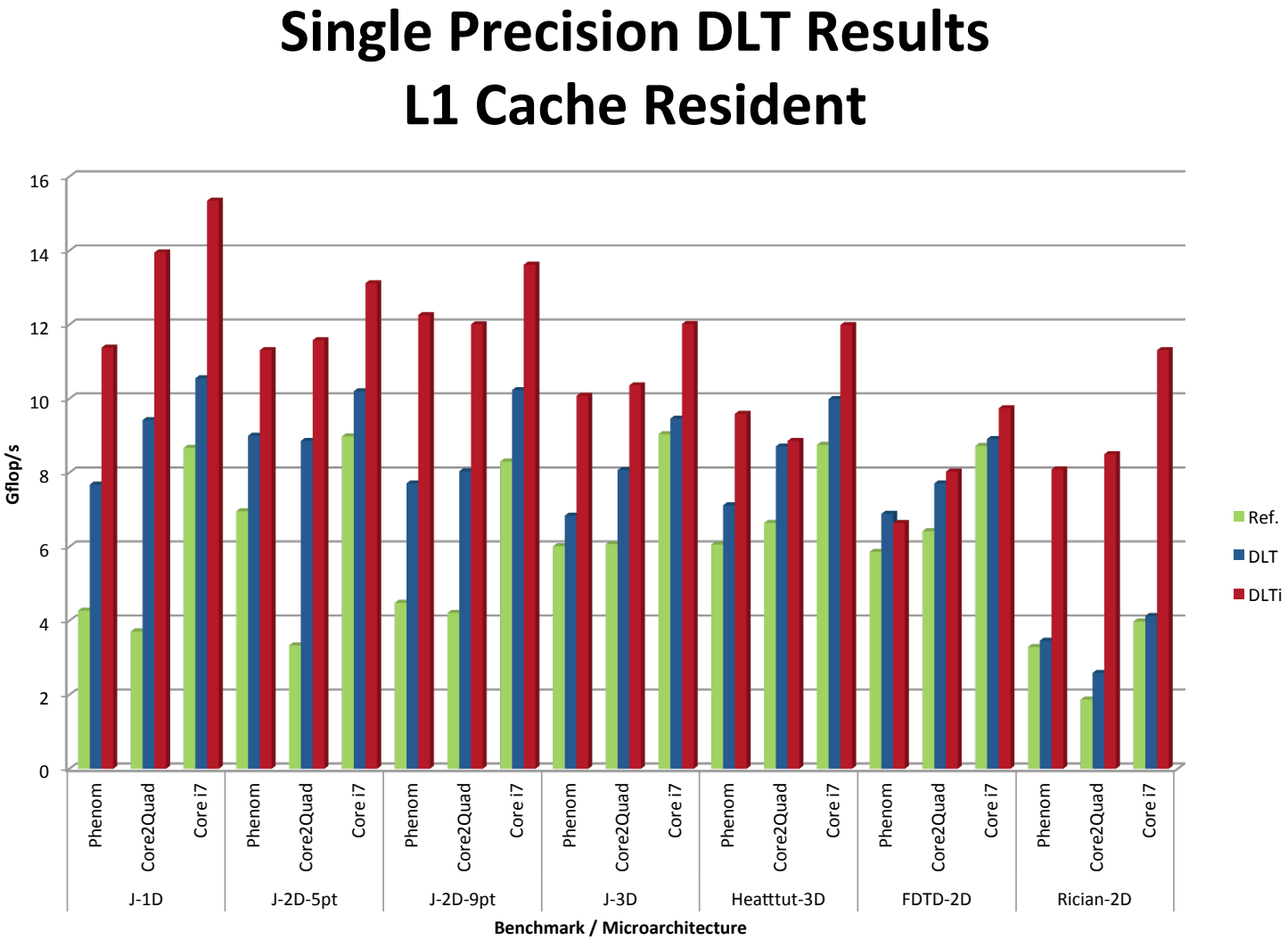
Experimental Setup

- Experiments run on 3 architectures (x86):
  - Intel Core2 Quad (Kentsfield): SAC resolved with low-performance shuffles
  - AMD Phenom (K10): SAC resolved with average-performance shuffles
  - Intel Core i7 (Nehalem): SAC resolved with fast redundant loads
- Data is L1-resident
  - assume tiling was performed beforehand if necessary
- Tested compiler: Intel ICC 11.1

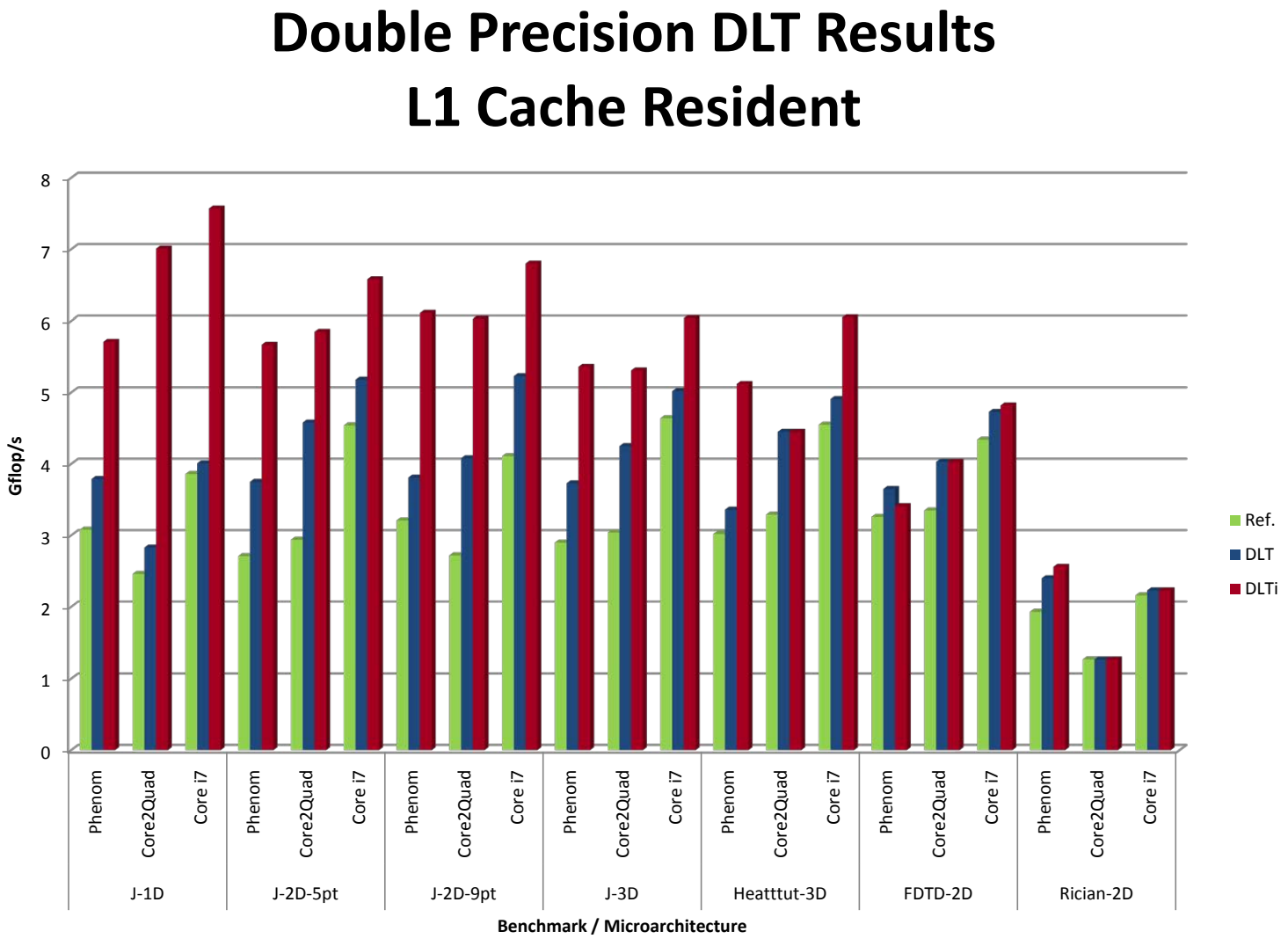
Three Code Variants Evaluated

1. **Ref**: reference code
   - Straightforward C implementation
   - Always auto-vectorized by the compiler
2. **DLT**: basic layout transformed
   - Straightforward C implementation with DLT arrays
   - Always auto-vectorized by the compiler
3. **DLTi**: intrinsics + layout transformed
   - C implementation with DLT arrays and SSE vector intrinsics

Single Precision Results



Double Precision Results



Summary of Experiments

- Performance improvement matches the shuffle/unaligned load costs
- Tested higher-dimensional stencils show less improvement:
  - more intra-stencil dependences
  - higher cache pressure
- Manual check of the ASM showed no shuffle, no redundant load instructions

Conclusion

- **Stream Alignment Conflict is the performance bottleneck for auto-vectorized stencils**
- Impact varies with micro-architecture characteristics, but is always significant
- A data layout transformation can solve this problem
- Strong performance improvement observed
  - Manual vectorization still beats automatic vectorization