TECHNIQUES FOR EFFICIENT MATLAB-TO-C COMPILATION

JOÃO BISPO, LUÍS REIS, JOÃO CARDOSO

SPeCS
Special-Purpose Computing Systems, languages and tools

ACM SIGPLAN 2nd International Workshop on Libraries, Languages and Compilers for Array Programming

Portland OR, USA - June 13th, 2015

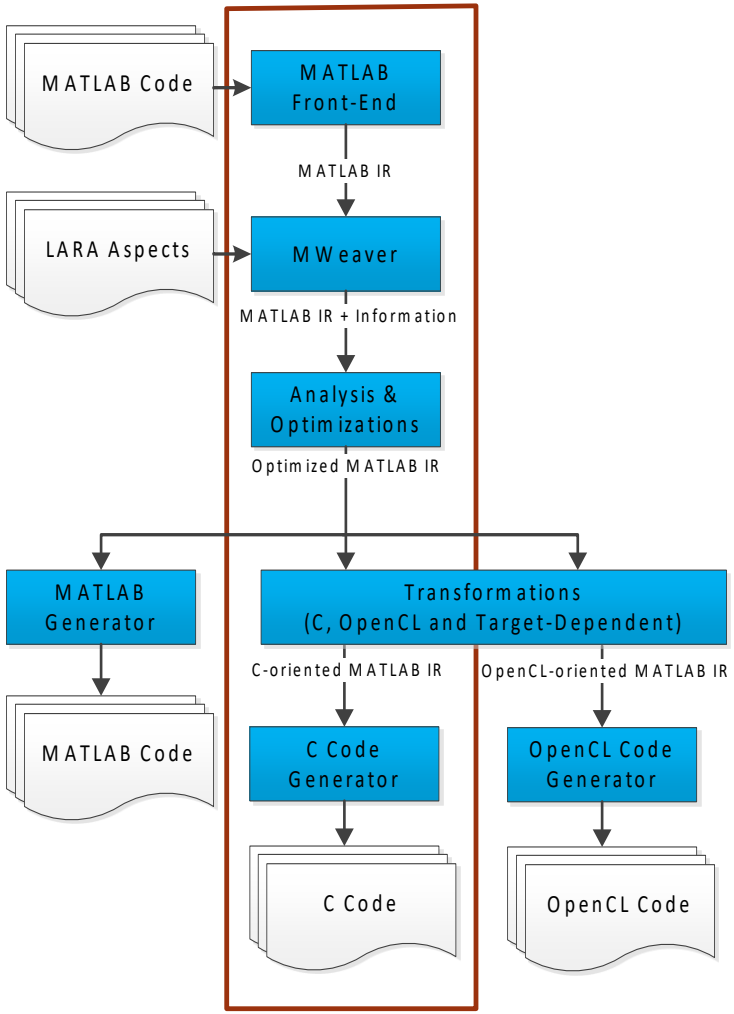
Universidade do Porto
FEUP Faculdade de Engenharia

OUTLINE

- Introduction
  - MATISSE C
  - Current Status
- Techniques
  - Element-wise Mapping Operations
  - MATISSE Primitives
  - Third Party Libraries
  - Algebraic Solver
- Results
- Conclusion and Future Work

INTRODUCTION - MATISSE C

- To maximize performance of generated C code we need to be target-aware
  - CPU architecture (memory hierarchy, cache sizes)
  - No floating-point HW units / units that perform poorly
  - HW synthesis (compliance to different tools)
- Our approach
  - Take advantage of high-level abstraction
  - Single reference implementation (MATLAB)
  - Additional information (LARA aspects)



INTRODUCTION - CURRENT STATUS

- MATLAB Parser
  - Tested with ~28.000 M-files (SourceForge, GitHub, literature)
  - About 2 minutes in Core2Quad a 2.66GHz (~14.000 lines/s per core)
- MATLAB Weaver
  - Supports MATLAB-to-MATLAB transformations
  - Annotates code with additional information
- MATLAB to C Engine
  - Currently supports 70 built-in MATLAB functions
  - Attention to performance (cast minimization, pointer vs copy, static vs dynamic, third party libraries) and code readability
  - Tested on several platforms (Desktop, Odroid, BeagleBoard)

TECHNIQUES – MAPPING OPERATIONS

- Hypotenuse exemple
  - "Best-case" scenario

```
R = sqrt((A .* A) + (B .* B));
```

TECHNIQUES – MAPPING OPERATIONS

- Hypotenuse exemple
  - "Best-case" scenario

```
R = sqrt((A .* A) + (B .* B));
```

- No element-wise mapping optimization

```
sqrt_e((add_e(((mult_e(A, A, temp_m0))),
((mult_e(B, B, temp_m1))), temp_m2)), R);
```

TECHNIQUES – MAPPING OPERATIONS

- Hypotenuse exemple
  - "Best-case" scenario

```
R = sqrt((A .* A) + (B .* B));
```

- No mapping

```
sqrt_e((add_e(((mult_e(A, A, temp_m0))),
((mult_e(B, B, temp_m1))), temp_m2)), R);
```

- With mapping

```
for(i = 0; i < numel_A; i++){
   R[i] = sqrt(A[i]*A[i] + B[i]*B[i]);
}
```

TECHNIQUES – MATLAB TEMPLATES

- MATISSE uses a small core to bootstrap translation of further MATLAB code
  - Core functionality implemented with C templates
  - Extended functionality implemented with MATLAB templates
    - E.g.: mean, min, max, linspace…

- + Speeds up development of additional functionality
- - Sometimes difficult to write efficient code directly in MATLAB

TECHNIQUES - MATISSE PRIMITIVES

MATLAB:
```
C = zeros(size(A))
```

C:
```
zeros_double(size_d(A, &temp_m0), C)
```

- Initialize a matrix with the same shape as another matrix
  - Always sets values to zero (not needed in certain situations, e.g., mapping)
  - Allocates a new array with the size of A

TECHNIQUES - MATISSE PRIMITIVES

MATLAB:
```
C = zeros(size(A))
```

C:
```
zeros_double(size_d(A, &temp_m0), C)
```

- Initialize a matrix with the same shape as another matrix
  - Always sets values to zero (not needed in certain situations, e.g., mapping)
  - Allocates a new array with the size of A

MATLAB:
```
C = matisse_new_array_from_matrix(A)
```

C:
```
new_array(A->shape, A->dims, C)
```

- Primitives: MATLAB functions with efficient C implementations
  - Used in MATLAB templates

TECHNIQUES - THIRD-PARTY LIBRARIES

- Competitive performant code needs third party libraries
- Wide-range of complex functionality available as C/C++ libraries
  - Linear Algebra (e.g., BLAS, LAPACK)
  - FFT (e.g., FFTW)
  - Computer Vision (e.g., OpenCV)
  - etc...
- Integration is not always trivial
  - Custom data-structures
  - Target-dependent performance, availability…

TECHNIQUES - ALGEBRAIC SIMPLIFIER/SOLVER

```
B = A(N×M×(i-1)+1:N×M×i)
```

- Algebraic simplifier/solver can extract compile-time information
- #elements of range == #elements of B
  - Useful for code with statically allocated memory

TECHNIQUES - ALGEBRAIC SIMPLIFIER/SOLVER

```
B = A(N×M×(i-1)+1:N×M×i)

#elements = N×M×i-(N×M×(i-1)+1)+1
```

TECHNIQUES - ALGEBRAIC SIMPLIFIER/SOLVER

```
B = A(N×M×(i-1)+1:N×M×i)

#elements = N×M×i-(N×M×(i-1)+1)+1

#elements = N×M
```

- N and M are statically known
  - The size of B is statically known
  - B can be implemented as a statically allocated matrix

EXPERIMENTAL SETUP

- PC – desktop PC with a 2.66 GHz Core 2 Quad processor, Windows 7 64-bit, 12 GB of RAM
- ODROID – ODROIDXU+E board, with an Exynos 5410 SoC with an 1.6 GHz ARM's big.LITTLE configuration, 1 GB of RAM
- BeagleBoard – BeagleBoard-XM revB running Ubuntu 12.10 32-bit, with a 1 GHz ARM Cortex-A8 and 512 MB of RAM

- Gcc 4.8, MATLAB 2014a, OpenBLAS v0.2.12

RESULTS – THIRD PARTY LIBRARIES

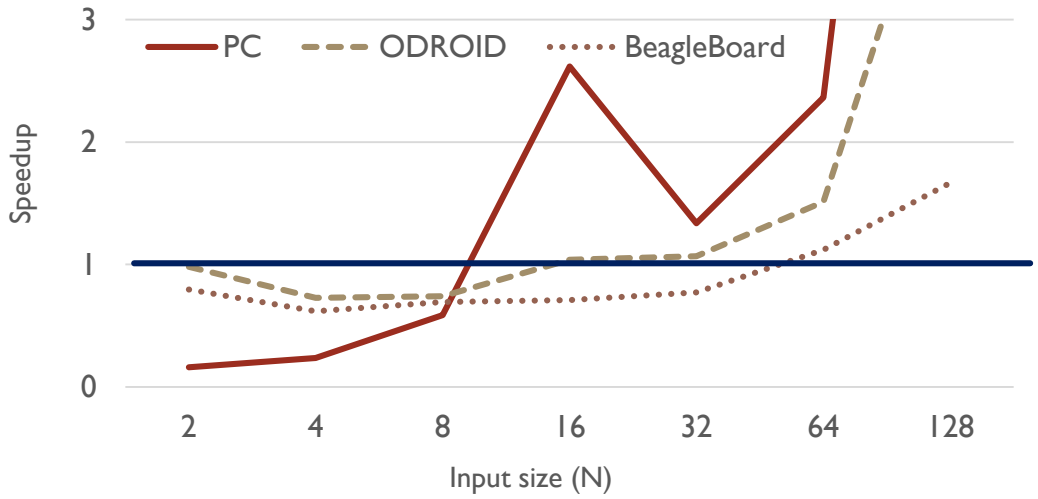
- Benchmark *closure*
  - Bottleneck is matrix multiplication
  - 94× speedup on PC (BLAS vs simple implementation, N = 1024)

- However, BLAS can be slower when N is small
  - Threshold is dependent on the target platform
  - Aspects allow flexible approach

RESULTS – THIRD PARTY LIBRARIES

- Benchmark *closure*
  - Bottleneck is matrix multiplication
  - 94× speedup on PC (BLAS vs simple implementation, N = 1024)

- However, BLAS can be slower when N is small
  - Threshold is dependent on the target platform
  - Aspects allow flexible approach



RESULTS – MAPPING OPERATIONS

Benchmark	PC	ODROID	BeagleBoard
capacitor	1.3×	1.0×	1.0×
hypotenuse	3.7×	3.2×	1.6×
nbody1d	1.8×	1.1×	1.2×

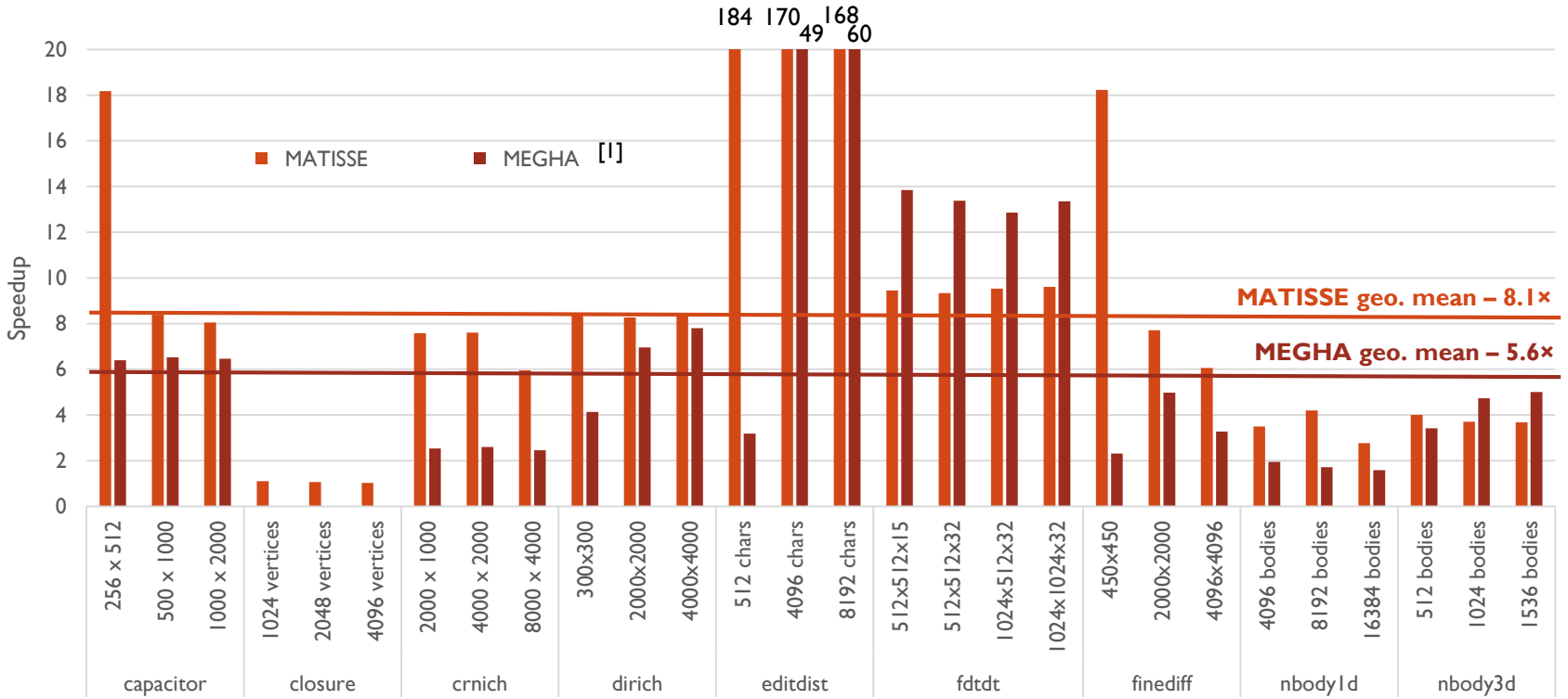
- Speedup relative to C code without mapping
- PC and ODROID code was vectorized by gcc (SSE and NEON, respectively)
- BeagleBoard processor does not have vector units
  - Less allocations, less memory operations, better locality

RESULTS – MATISSE PRIMITIVES

Benchmark	PC	ODROID	BeagleBoard
capacitor	1.5×	1.0×	1.0×
crnich	1.3×	1.0×	1.0×
hypotenuse	1.3×	1.1×	1.0×
nbody1d	1.3×	1.1×	1.0×

- Speedup relative to MATLAB templates without primitives
- Some improvements on PC
  - Avoiding memory initialization, less memory allocations

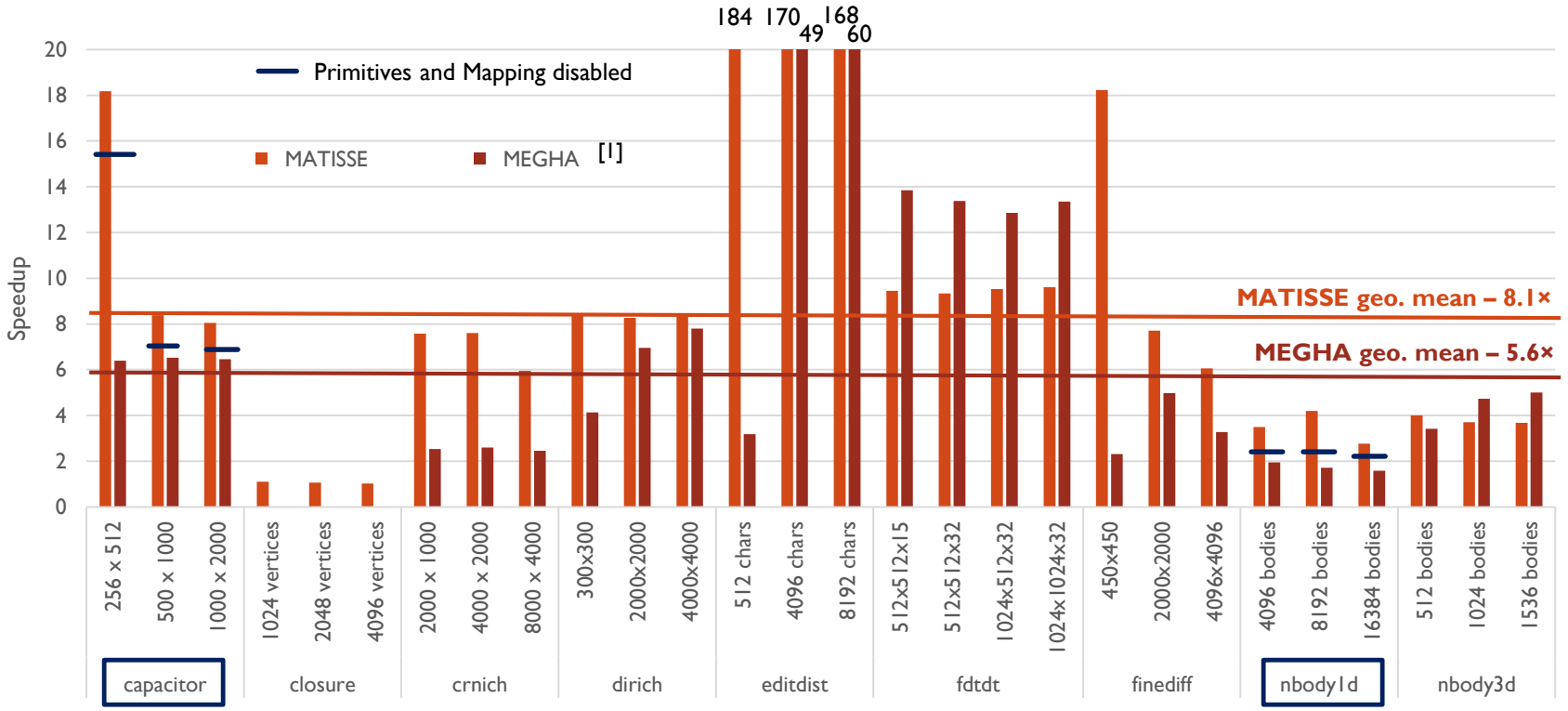
RESULTS – MATISSE C & MEGHA VS MATLAB



- MATISSE: 8.0× geo. mean on AMD A10-7850K 4.10GHz, from 2014

[1] A. Prasad, J. Anantpur, and R. Govindarajan, "Automatic compilation of MATLAB programs for synergistic execution on heterogeneous processors," in *ACM Sigplan Notices*, 2011, vol. 46, pp. 152–163.

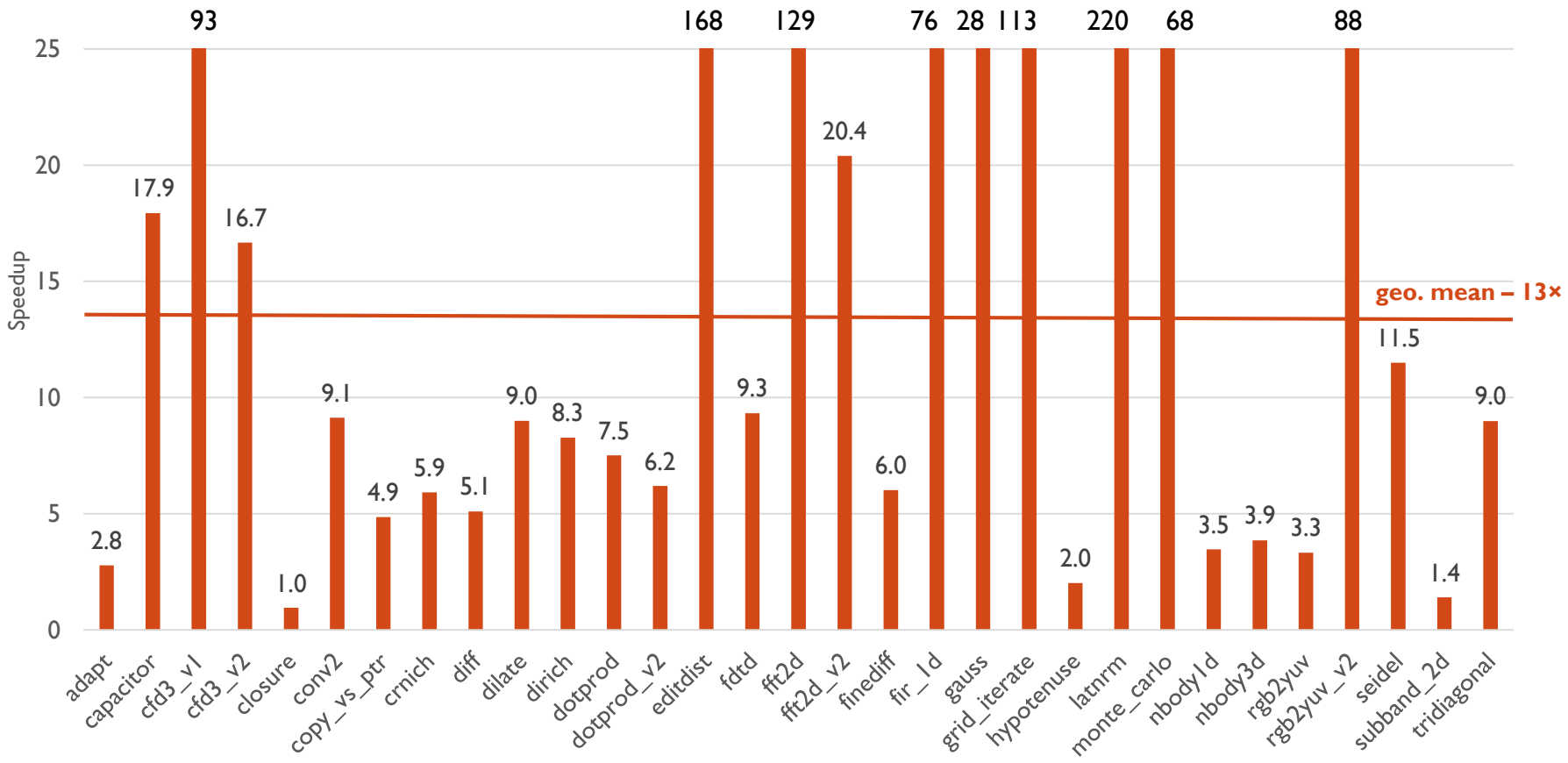
RESULTS – MATISSE C & MEGHA VS MATLAB



- MATISSE: 8.0× geo. mean on AMD A10-7850K 4.10GHz, from 2014

[1] A. Prasad, J. Anantpur, and R. Govindarajan, "Automatic compilation of MATLAB programs for synergistic execution on heterogeneous processors," in *ACM Sigplan Notices*, 2011, vol. 46, pp. 152–163.

RESULTS – MATISSE C VS MATLAB ON PC



- Results in C within 0.1% of the MATLAB value

RELATED WORK

Work	Description
MATLAB Coder	Mathworks MATLAB-to-C solution
FALCON	MATLAB to FORTRAN90
Joisha, Banerjee et al.	Focus on type and shape inference techniques
McLab	AOP, JIT, MATLAB semantics, back-ends (FORTRAN95, X10)
MEGHA	MATLAB to C/CUDA

CONCLUSIONS

- MATLAB abstraction level enables certain analysis
  - E.g., element-wise mapping
- High abstraction level is a double-edged sword
  - Use of primitives to break abstraction when needed
- Performance on par or better than state-of-art compiler
- Importance of being target-aware
  - Tailor code to target architecture (e.g., BLAS threshold)

ONGOING AND FUTURE WORK

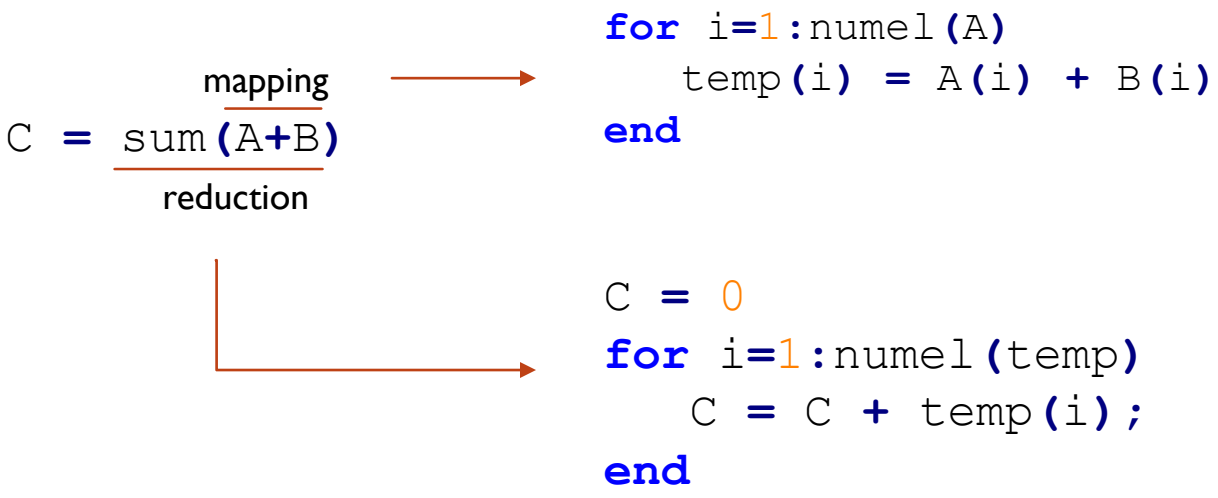
- Improve type/shape-inference
  - E.g., split type-inference into two phases, MATLAB types and back-end types
- Further research on multi-target C/OpenCL
  - E.g., high-level synthesis for hardware
- Add support to other MATLAB features
  - Cell Arrays, Function Handles, Structures, Plots

THANK YOU!
QUESTIONS?

MATISSE DEMO:
http://specs.fe.up.pt/tools/matisse

TECHNIQUES – MAPPING OPERATIONS

- Possible to infer map/reduce patterns for certain operations
- Consider that A and B are vectors



- Currently only supports 'mapping' concept
- Future work: add 'reduction' to be able to generate a single loop

ASPECT – IDIV

- closure.m

```
% B is a square matrix whose side size is always a power of two
function [B_out] = closure(B)

N = size(B, 1);


ii = N/2; % This division will always be an integer division
while ii >= 1,
      B = B*B;
      ii = ii/2; % This division also
end;

B_out = B > 0;
```

ASPECT – IDIV

- closure.m

```
% B is a square matrix whose side size is always a power of two
function [B_out] = closure(B)

N = size(B, 1);


 ii = N/2; % This division will always be an integer division
while ii >= 1,
      B = B*B;
      ii = ii/2; % This division also
end;

B_out = B > 0;
```

ASPECT – IDIV

- closure.m

```
% B is a square matrix whose side size is always a power of two
function [B_out] = closure(B)

N = size(B, 1);


ii = N/2; % This division will always be an integer division
while ii >= 1,
      B = B*B;
      ii = ii/2; % This division also
end;

B_out = B > 0;
```

- closure_idivide.lara

```
aspectdef closure_idivide
    select function{"closure"} end
    apply
        call replace_operator("/", "N", "2", "matisse_idivide");
        call replace_operator("/", "ii", "2", "matisse_idivide");
    end
end
```