Topic V26

Floating-Point

Matrix Multiplication

Reading: (Section 3.5)

Floating Point (example)

Matrix multiplication of C = A * B (Double Precision Matrix Multiply)

Assume C, A, and B are all square matrices with 32 elements in each dimension

This is an abuse of notation in the textbook; in C you cannot skip the size of previous dimensions

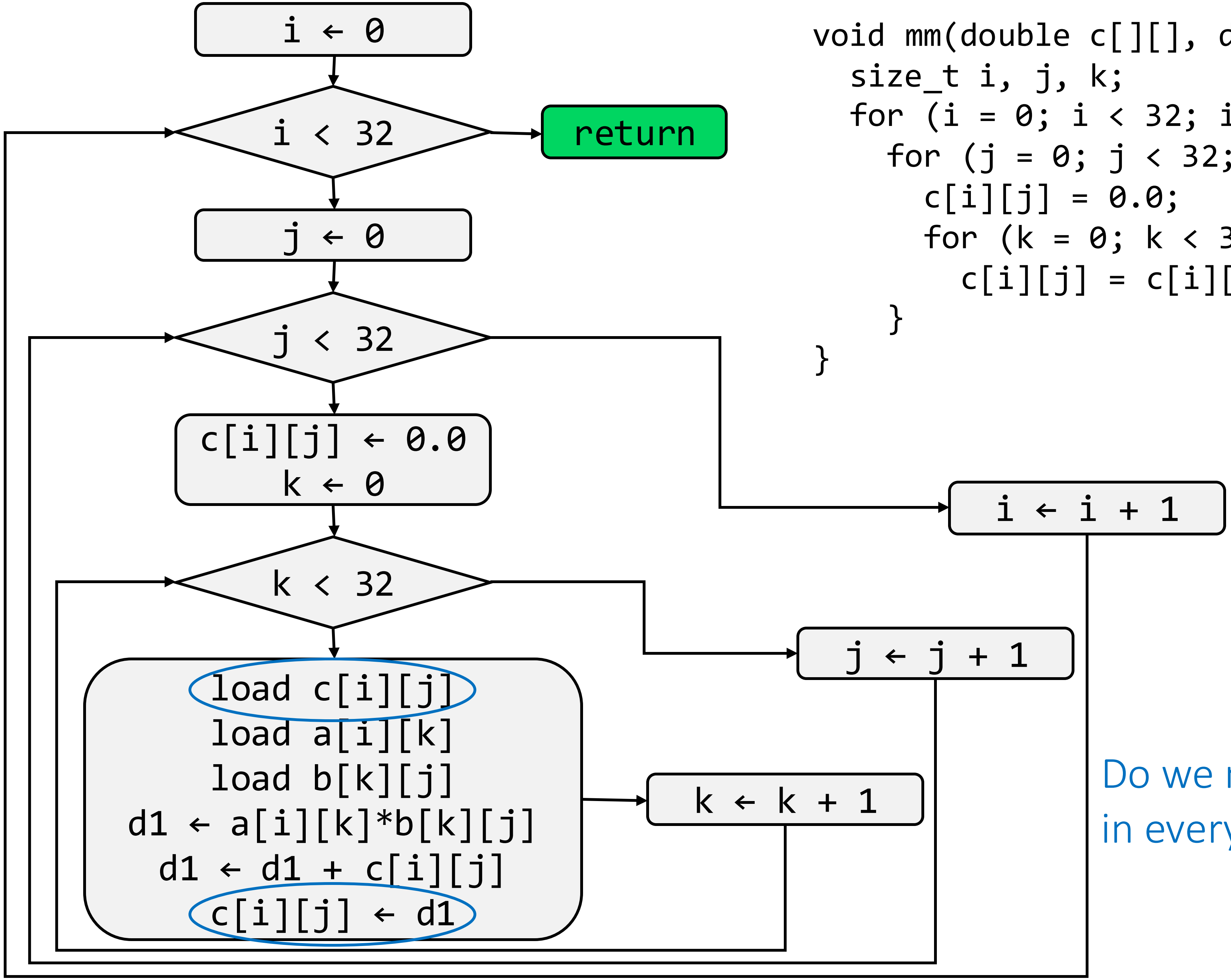
```
void mm(double c[][], double a[][], double b[][]){
  size_t i, j, k;
  for (i = 0; i < 32; i = i + 1)
    for (j = 0; j < 32; j = j + 1){
      c[i][j] = 0.0;
      for (k = 0; k < 32; k = k + 1)
        c[i][j] = c[i][j] + a[i][k] * b[k][j];
    }
}
```

Parameter Passing

```
base of c[] ↔ a0
base of a[] ↔ a1
base of b[] ↔ a2
```

Assumption

```
i ↔ s0
j ↔ s1
k ↔ s2
```



i ← 0

i < 32

return

j ← 0

j < 32

c[i][j] ← 0.0
k ← 0

k < 32

load c[i][j]
load a[i][k]
load b[k][j]
d1 ← a[i][k]*b[k][j]
d1 ← d1 + c[i][j]
c[i][j] ← d1

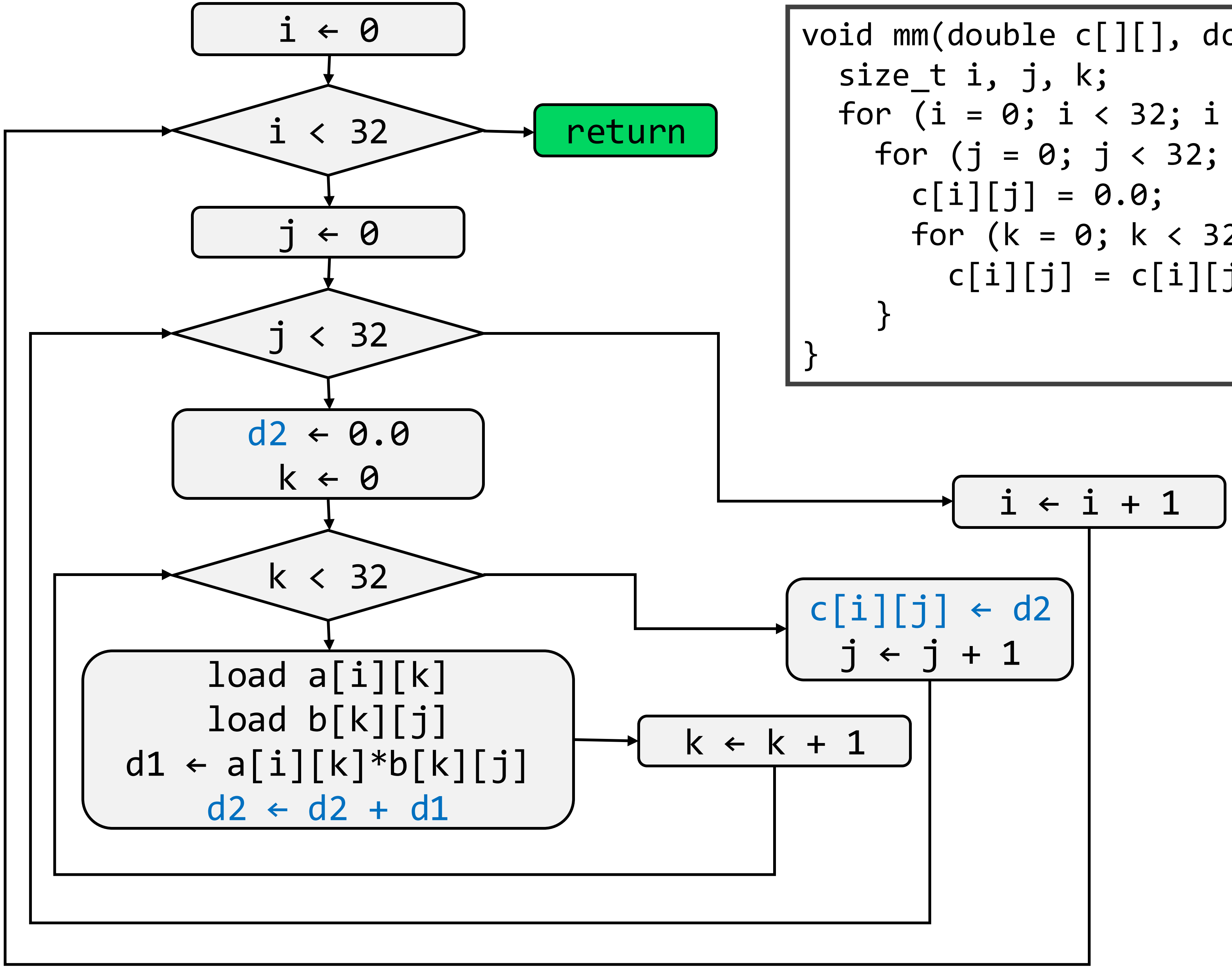
k ← k + 1

j ← j + 1

i ← i + 1

```
void mm(double c[][], double a[][], double b[][]){
  size_t i, j, k;
  for (i = 0; i < 32; i = i + 1)
    for (j = 0; j < 32; j = j + 1){
      c[i][j] = 0.0;
      for (k = 0; k < 32; k = k + 1)
        c[i][j] = c[i][j] + a[i][k] * b[k][j];
    }
}
```

Do we need to load and store c[i][j] in every iteration of loop k?



```
void mm(double c[][], double a[][], double b[][]){
  size_t i, j, k;
  for (i = 0; i < 32; i = i + 1)
    for (j = 0; j < 32; j = j + 1){
      c[i][j] = 0.0;
      for (k = 0; k < 32; k = k + 1)
        c[i][j] = c[i][j] + a[i][k] * b[k][j];
    }
}
```

Parameter Passing

base of c[] ↔ a0

base of a[] ↔ a1

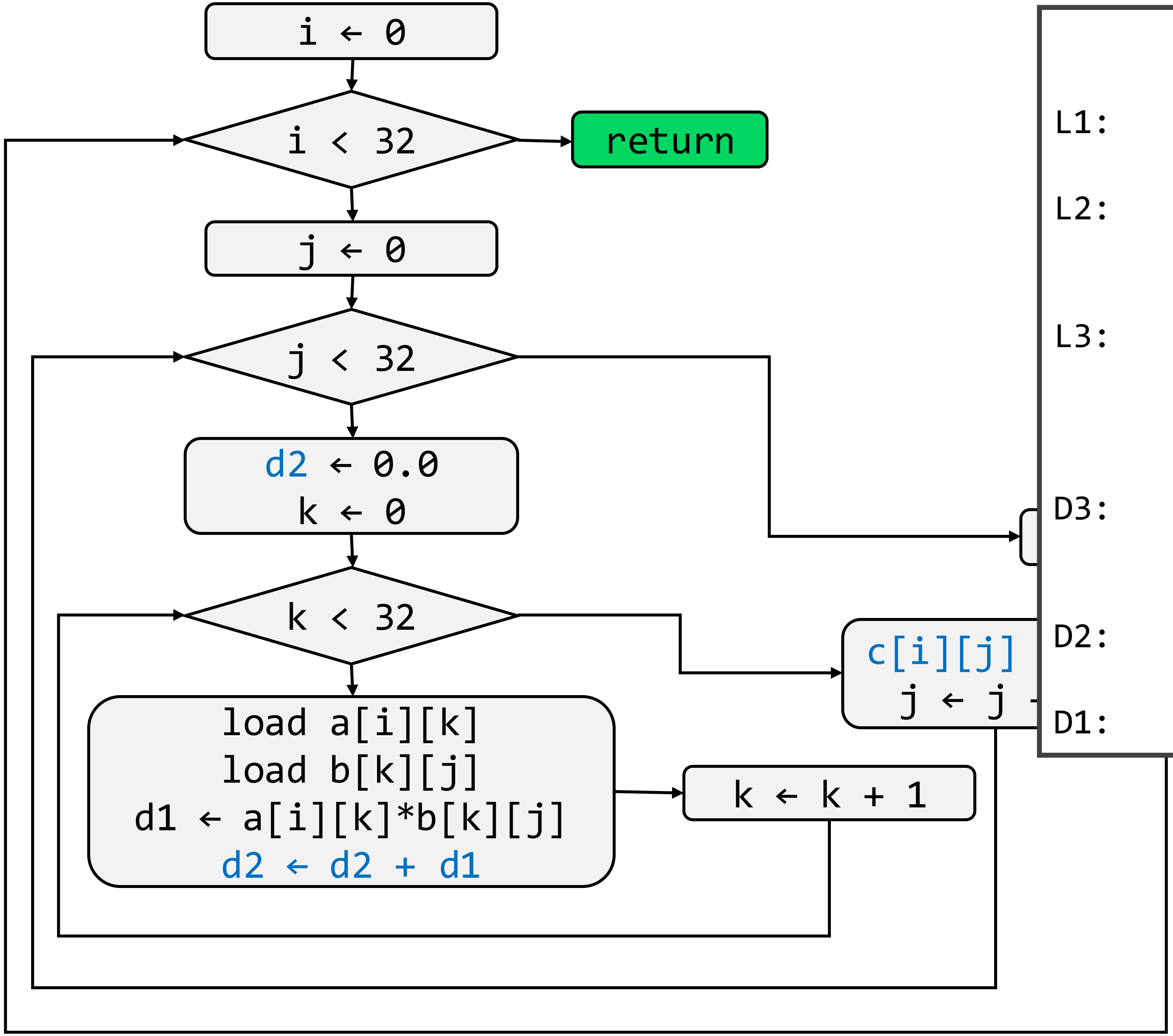
base of b[] ↔ a2

Assumption

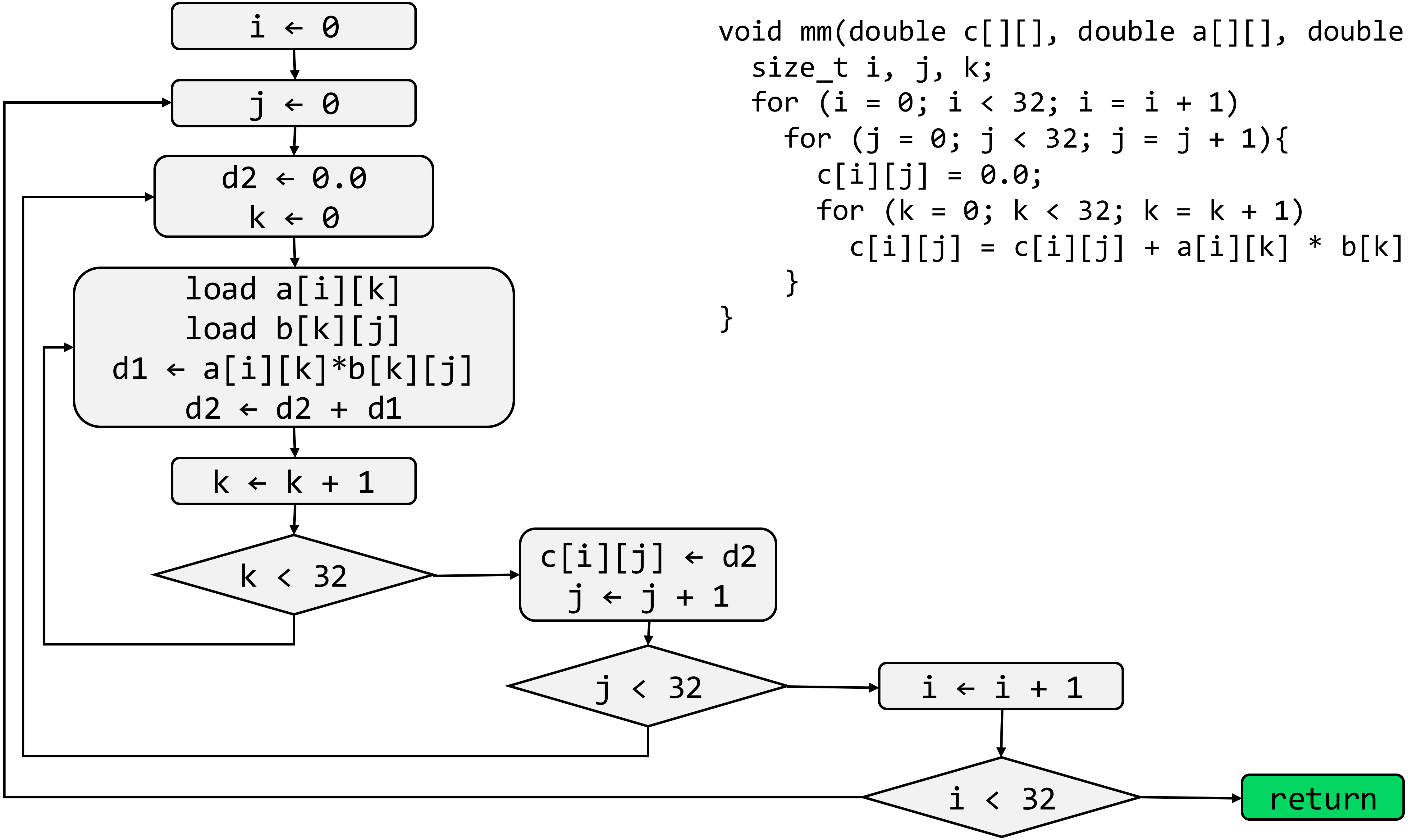
i ↔ s0

j ↔ s1

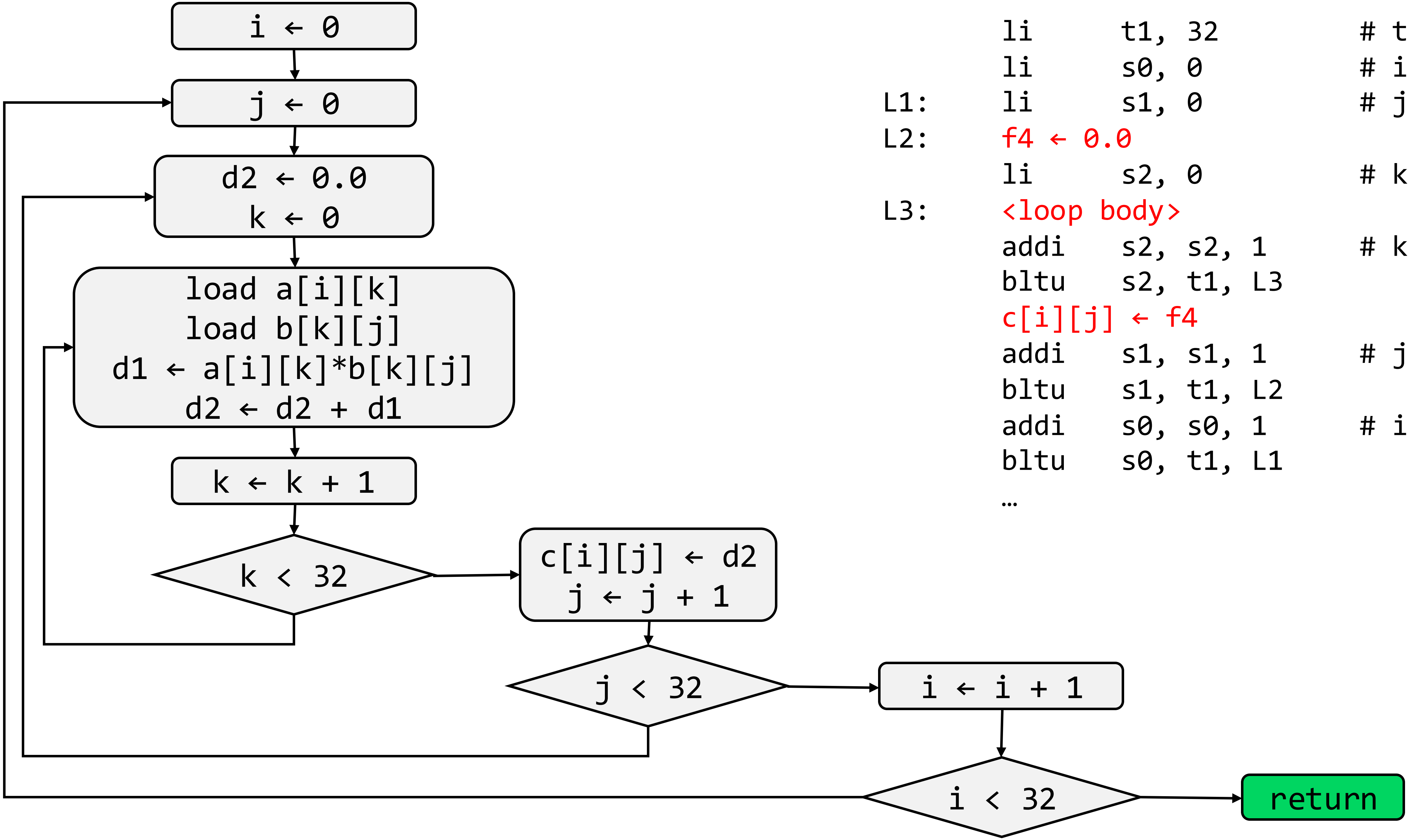
k ↔ s2



```
        li      t1, 32          # t1 ← 32
        li      s0, 0           # i ← 0
L1:     bgeu    s0, t1, D1
        li      s1, 0           # j ← 0
L2:     bgeu    s1, t1, D2
        f4 ← 0.0
        li      s2, 0           # k ← 0
L3:     bgeu    s2, t1, D3
        <loop body>
        addi    s2, s2, 1       # k ← k + 1
        jal     zero, L3
D3:     c[i][j] ← f4
        addi    s1, s1, 1       # j ← j + 1
        jal     zero, L2
D2:     addi    s0, s0, 1       # i ← i + 1
        jal     zero, L1
D1:     …
```



```
void mm(double c[][], double a[][], double b[][]){
  size_t i, j, k;
  for (i = 0; i < 32; i = i + 1)
    for (j = 0; j < 32; j = j + 1){
      c[i][j] = 0.0;
      for (k = 0; k < 32; k = k + 1)
        c[i][j] = c[i][j] + a[i][k] * b[k][j];
    }
}
```



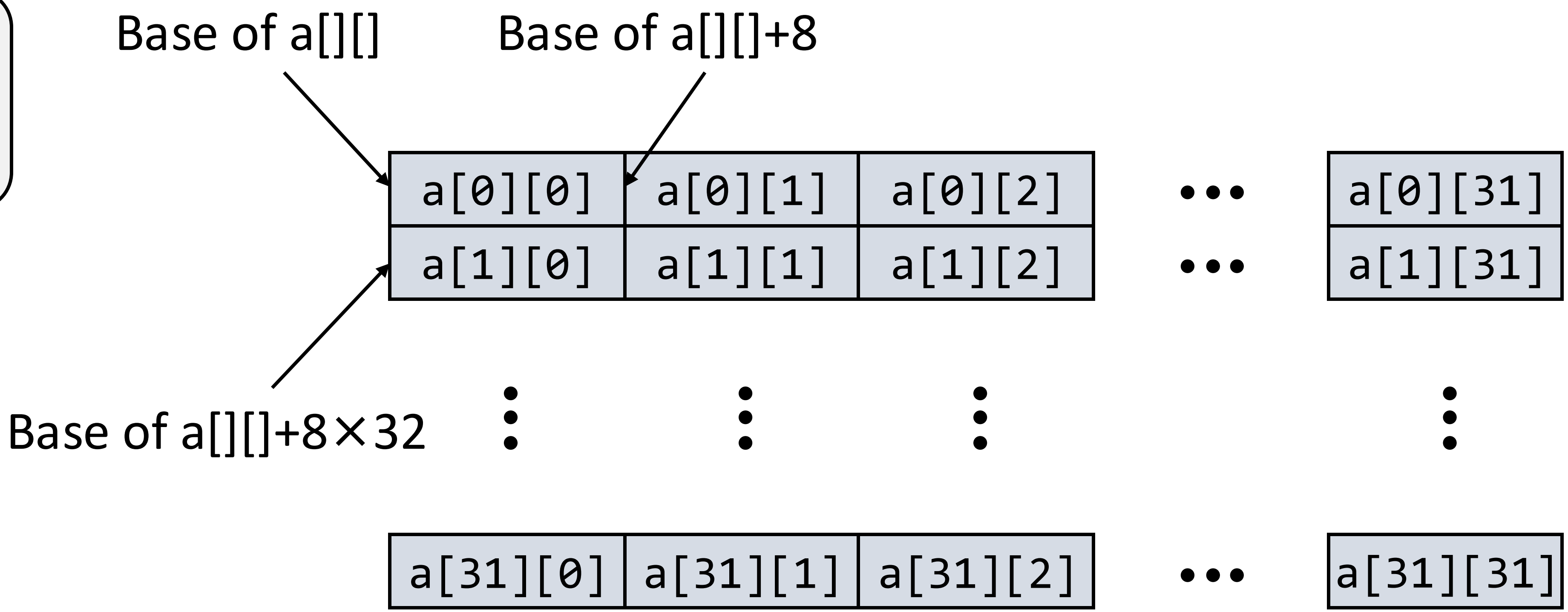
```
        li      t1, 32          # t1 ← 32
        li      s0, 0           # i ← 0
L1:     li      s1, 0           # j ← 0
L2:     f4 ← 0.0
        li      s2, 0           # k ← 0
L3:     <loop body>
        addi    s2, s2, 1       # k ← k + 1
        bltu    s2, t1, L3
        c[i][j] ← f4
        addi    s1, s1, 1       # j ← j + 1
        bltu    s1, t1, L2
        addi    s0, s0, 1       # i ← i + 1
        bltu    s0, t1, L1
        …
```

The Loop Body

How do we load a[i][k] into a floating point register?

First we have to consider how a 2-dimensional matrix of doubles is stored in memory

```
      load a[i][k]
      load b[k][j]
d1 ← a[i][k]*b[k][j]
      d2 ← d2 + d1
```



Base of a[][]

Base of a[][]+8

Base of a[][]+8×32

a[0][0]	a[0][1]	a[0][2]	•••	a[0][31]
a[1][0]	a[1][1]	a[1][2]	•••	a[1][31]
⋮	⋮	⋮		⋮
a[31][0]	a[31][1]	a[31][2]	•••	a[31][31]

In general, the address of a[i][k] is given by:

address(a[i][k]) = base of a[][] + (i × 32 + k) × 8

The Loop Body (Cont.)

In general, the address of a[i][k] is given by:

address(a[i][k]) = base of a[][] + (i × 32 + k) × 8

```
     load a[i][k]
     load b[k][j]
d1 ← a[i][k]*b[k][j]
     d2 ← d2 + d1
```

RISC-V assembly for load a[i][k]

```
L3:     slli    t2, s0, 5       # t2 ← 32 * i
        add     t2, t2, s2      # t2 ← 32 * i + k
        slli    t2, t2, 3       # t2 ← (32 * i + k) * 8
        add     t2, a1, t2      # t2 ← Addr(a[i][k])
        fld     f1, 0(t2)       # f1 ← a[i][k]
```

Write the code to load b[k][j] in f2

RISC-V assembly for load b[k][j]

```
        slli    t2, s2, 5       # t2 ← 32 * k
        add     t2, t2, s1      # t2 ← 32 * k + j
        slli    t2, t2, 3       # t2 ← (32 * k + j) * 8
        add     t2, a2, t2      # t2 ← Addr(b[k][j])
        fld     f2, 0(t2)       # f2 ← b[k][j]
```

The Loop Body (Cont.)

Once we have loaded a[i][k] into f1 and b[k][j] into f2 we can proceed to perform the multiply and the add

```
    load a[i][k]
    load b[k][j]
d1 ← a[i][k]*b[k][j]
    d2 ← d2 + d1
```

RISC-V assembly for FP multiply and add

```
fmul.d        f1, f1, f2   # f1 ← a[i][k] × b[k][j]
fadd.d        f0, f0, f1   # c[i][j] + a[i][k] × b[k][j]
```

Initializing and Storing f4

How can we initialize f4?

RISC-V assembly to initialize f4

```
fcvt.d.w     f4, zero
```

Converts the integer zero into a floating-point zero

How can we store f4 in a[i][j]?

RISC-V assembly to store f4 in c[i][j]

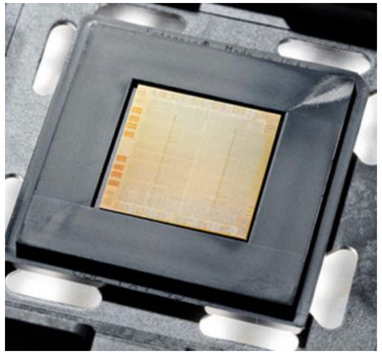
```
slli   t2, s0, 5     # t2 ← 32 × i
add    t2, t2, s1    # t2 ← 32 × i + j
slli   t2, t2, 3     # ← (32 × i + j) × 8
add    t2, a0, t2    # t2 ← Addr(c[i][j])
fsd    f4, 0(t2)     # c[i][j] ← f4
```

```
       li      t1, 32           # t1 ← 32
       li      s0, 0            # i ← 0
L1:    li      s1, 0            # j ← 0
L2:    f4 ← 0.0
       li      s2, 0            # k ← 0
L3:    <loop body>
       addi    s2, s2, 1        # k ← k + 1
       bltu    s2, t1, L3
       c[i][j] ← f4
       addi    s1, s1, 1        # j ← j + 1
       bltu    s1, t1, L2
       addi    s0, s0, 1        # i ← i + 1
       bltu    s0, t1, L1
       …
```

```
                        mm:     …
                                li          t1, 32          # t1 ← 32
                                li          s0, 0           # i ← 0
                        L1:     li          s1, 0           # j ← 0
                        L2:     fcvt.d.w    f4, zero
                                li          s2, 0           # k ← 0
                        L3:     slli        t2, s0, 5       # t2 ← 32 * i
                                add         t2, t2, s2      # t2 ← 32 * i + k
load a[i][k] in f1              slli        t2, t2, 3       # t2 ← (32 * i + k) * 8
                                add         t2, a1, t2      # t2 ← Addr(a[i][k])
                                fld         f1, 0(t2)       # f1 ← a[i][k]
                                slli        t2, s2, 5       # t2 ← 32 * k
                                add         t2, t2, s1      # t2 ← 32 * k + j
load b[k][j] in f2              slli        t2, t2, 3       # t2 ← (32 * k + j) * 8
                                add         t2, a2, t2      # t2 ← Addr(b[k][j])
                                fld         f2, 0(t2)       # f2 ← b[k][j]
                                fmul.d      f1, f1, f2      # f1 ← a[i][k] × b[k][j]
                                fadd.d      f0, f0, f1      # c[i][j] + a[i][k] × b[k][j]
                                addi        s2, s2, 1       # k ← k + 1
                                bltu        s2, t1, L3
                                slli        t2, s0, 5       # t2 ← 32 × i
                                add         t2, t2, s1      # t2 ← 32 × i + j
store f4 in c[i][j]             slli        t2, t2, 3       # ← (32 × i + j) × 8
                                add         t2, a0, t2      # t2 ← Addr(c[i][j])
                                fsd         f4, 0(t2)       # c[i][j] ← f4
                                addi        s1, s1, 1       # j ← j + 1
                                bltu        s1, t1, L2
                                addi        s0, s0, 1       # i ← i + 1
                                bltu        s0, t1, L1
                                …
```

```
                          mm:      …
                                   li         t1, 32        # t1 ← 32
                                   li         s0, 0         # i ← 0
                          L1:      li         s1, 0         # j ← 0
                          L2:      fcvt.d.w   f4, zero
                                   li         s2, 0         # k ← 0
                          L3:      slli       t2, s0, 5     # t2 ← 32 * i
                                   add        t2, t2, s2    # t2 ← 32 * i + k
load a[i][k] in f1                 slli       t2, t2, 3     # t2 ← (32 * i + k) * 8
                                   add        t2, a1, t2    # t2 ← Addr(a[i][k])
                                   fld        f1, 0(t2)     # f1 ← a[i][k]
                                   slli       t2, s2, 5     # t2 ← 32 * k
                                   add        t2, t2, s1    # t2 ← 32 * k + j
load b[k][j] in f2                 slli       t2, t2, 3     # t2 ← (32 * k + j) * 8
                                   add        t2, a2, t2    # t2 ← Addr(b[k][j])
                                   fld        f2, 0(t2)     # f2 ← b[k][j]
                                   fmul.d     f1, f1, f2    # f1 ← a[i][k] × b[k][j]
                                   fadd.d     f0, f0, f1    # c[i][j] + a[i][k] × b[k][j]
                                   addi       s2, s2, 1     # k ← k + 1
                                   bltu       s2, t1, L3
                                   slli       t2, s0, 5     # t2 ← 32 × i
                                   add        t2, t2, s1    # t2 ← 32 × i + j
store f4 in c[i][j]                slli       t2, t2, 3     # ← (32 × i + j) × 8
                                   add        t2, a0, t2    # t2 ← Addr(c[i][j])
                                   fsd        f4, 0(t2)     # c[i][j] ← f4
                                   addi       s1, s1, 1     # j ← j + 1
                                   bltu       s1, t1, L2
                                   addi       s0, s0, 1     # i ← i + 1
                                   bltu       s0, t1, L1
                                   …
```

Matrix Multiplication Accelerators



Hot Chips 32 Server Processors

IBM Releases Power ISA v3.1; To Present POWER10 At Hot Chips 32

May 23, 2020 David Schor 0 Comments bfloat16, Hot Chips, Hot Chips 32, IBM, OpenPOWER,

IBM Reveals Next-Generation IBM POWER10 Processor

New CPU co-optimized for Red Hat OpenShift for enterprise hybrid cloud

IBM BRINGS AN ARCHITECTURE GUN TO A CHIP KNIFE FIGHT

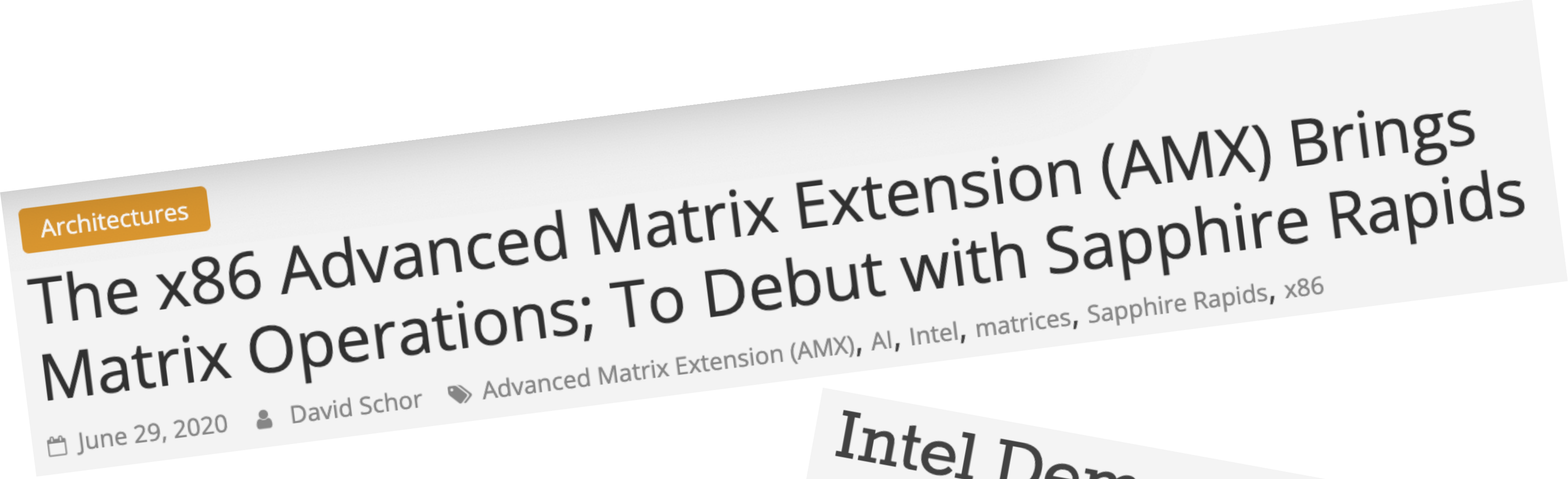
August 18, 2020 Timothy Prickett Morgan

THE MEMORY AREA NETWORK AT THE HEART OF IBM'S POWER10

September 3, 2020 Timothy Prickett Morgan

BLOG – IBM Announces POWER10 Processor Technology

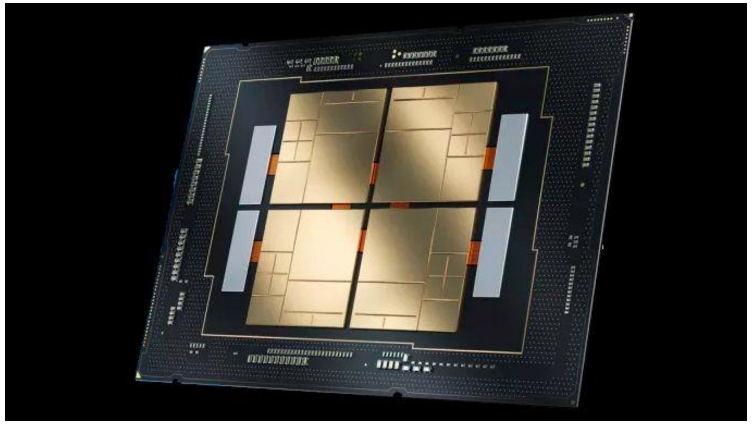
August 21st, 2020

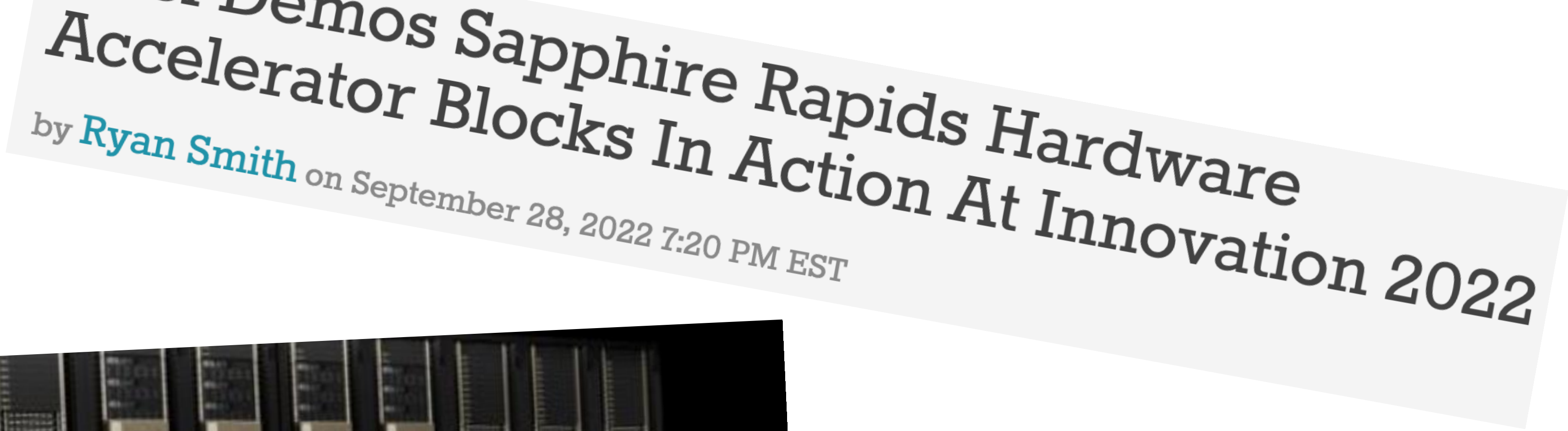


Architectures

The x86 Advanced Matrix Extension (AMX) Brings Matrix Operations; To Debut with Sapphire Rapids

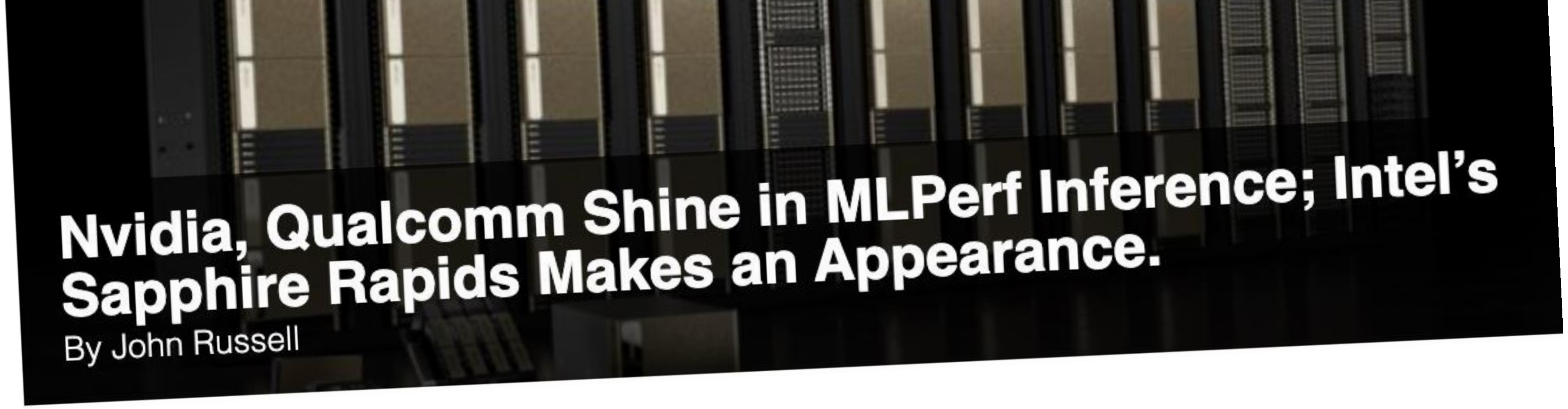
June 29, 2020 David Schor Advanced Matrix Extension (AMX), AI, Intel, matrices, Sapphire Rapids, x86





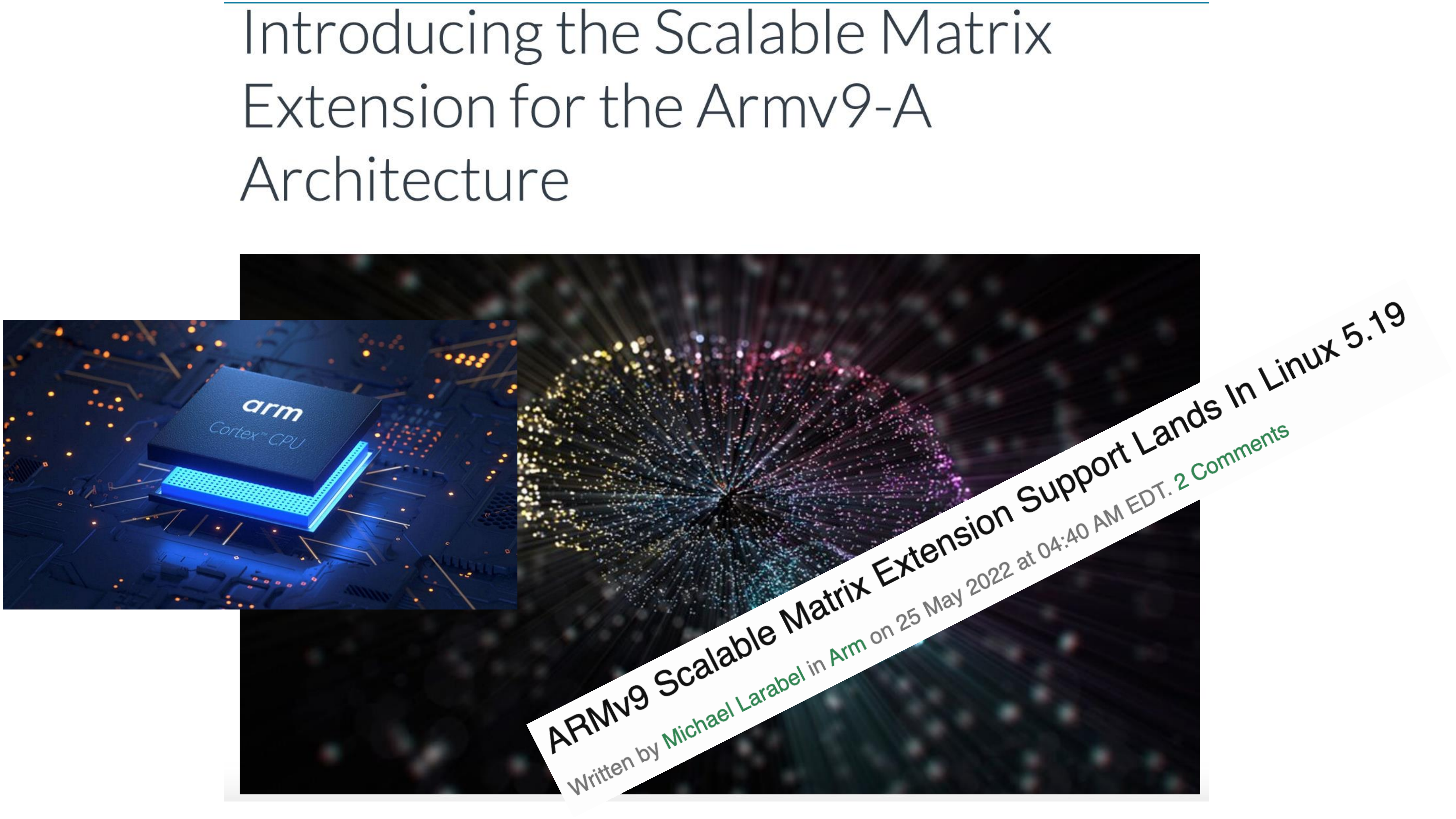
Intel Demos Sapphire Rapids Hardware Accelerator Blocks In Action At Innovation 2022

by Ryan Smith on September 28, 2022 7:20 PM EST



Nvidia, Qualcomm Shine in MLPerf Inference; Intel's Sapphire Rapids Makes an Appearance.

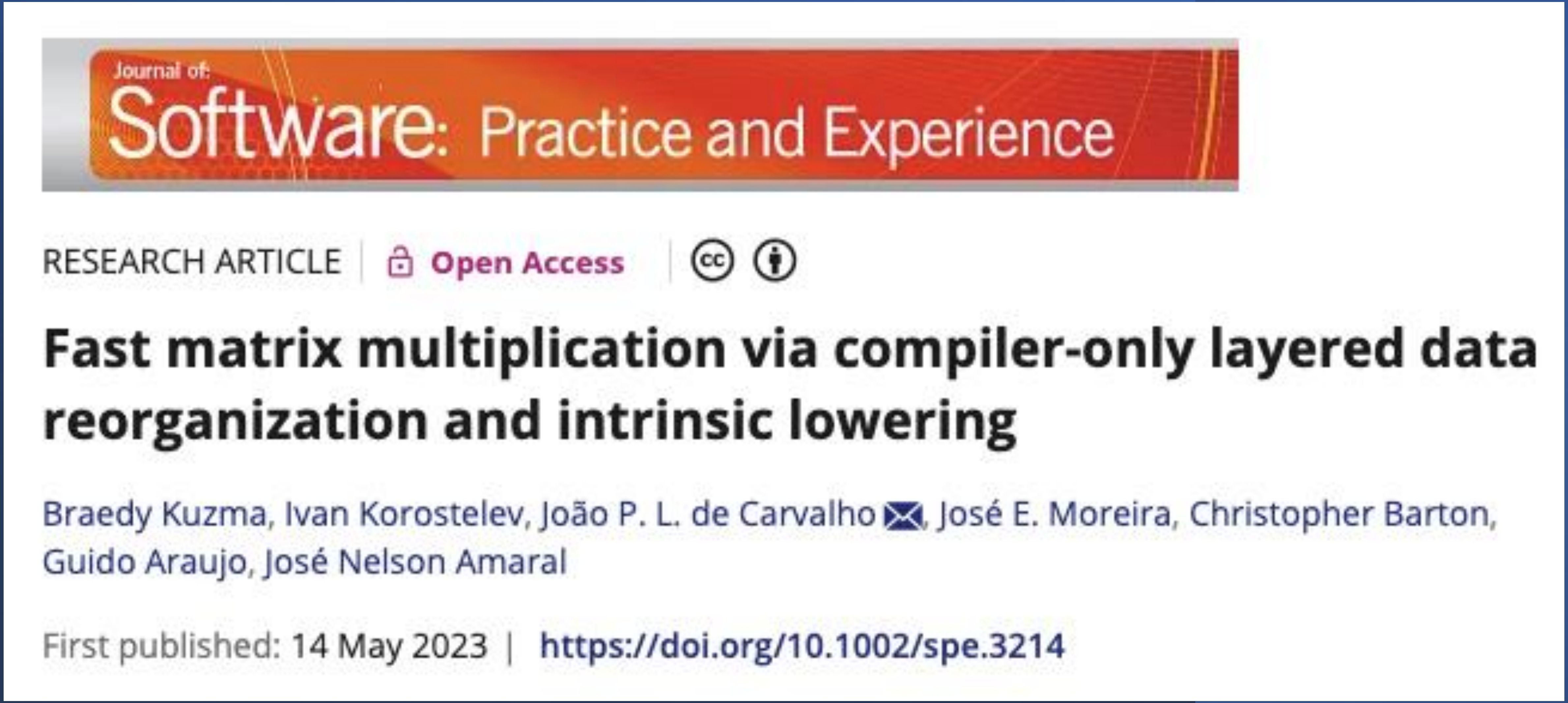
By John Russell



Introducing the Scalable Matrix Extension for the Armv9-A Architecture

ARMv9 Scalable Matrix Extension Support Lands In Linux 5.19

Written by Michael Larabel in Arm on 25 May 2022 at 04:40 AM EDT. 2 Comments



Journal of: Software: Practice and Experience

RESEARCH ARTICLE | Open Access

Fast matrix multiplication via compiler-only layered data reorganization and intrinsic lowering

Braedy Kuzma, Ivan Korostelev, João P. L. de Carvalho, José E. Moreira, Christopher Barton, Guido Araujo, José Nelson Amaral

First published: 14 May 2023 | https://doi.org/10.1002/spe.3214

IEEE micro

The magazine for chip and silicon systems designers

VOLUME 42, NUMBER 5

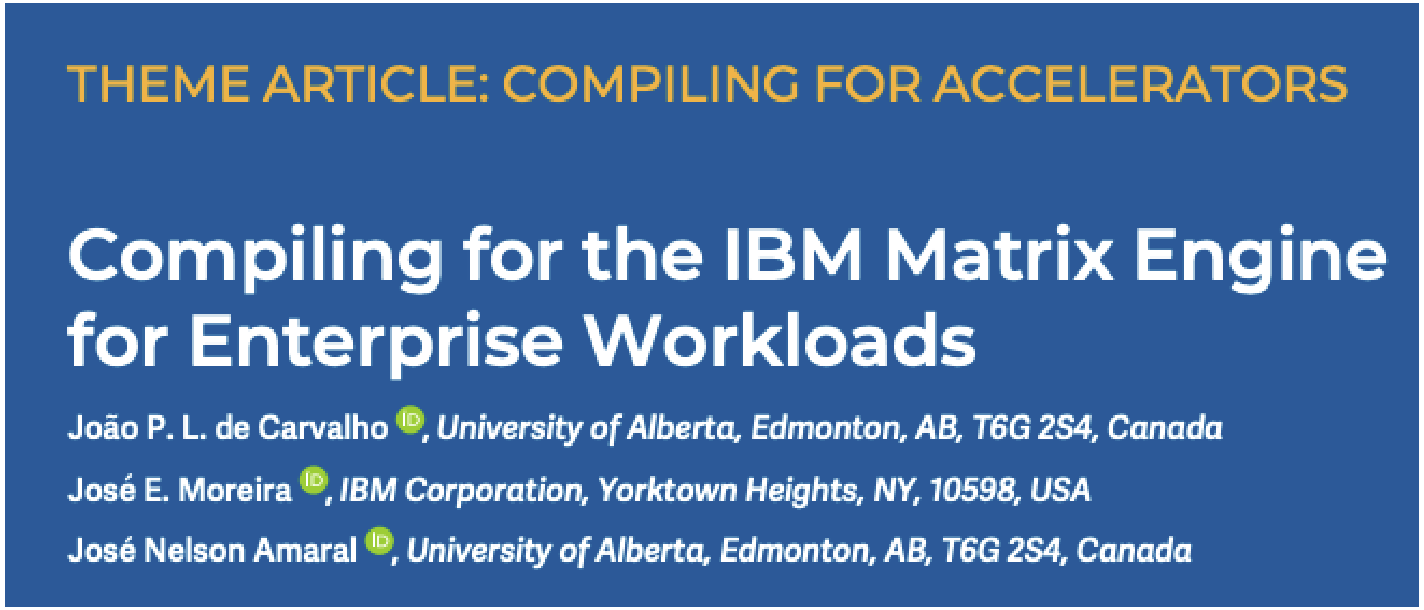
SEPTEMBER/OCTOBER 2022

Compiling for Accelerators

IEEE

IEEE COMPUTER SOCIETY

www.computer.org/micro



THEME ARTICLE: COMPILING FOR ACCELERATORS

Compiling for the IBM Matrix Engine for Enterprise Workloads

João P. L. de Carvalho, *University of Alberta, Edmonton, AB, T6G 2S4, Canada*

José E. Moreira, *IBM Corporation, Yorktown Heights, NY, 10598, USA*

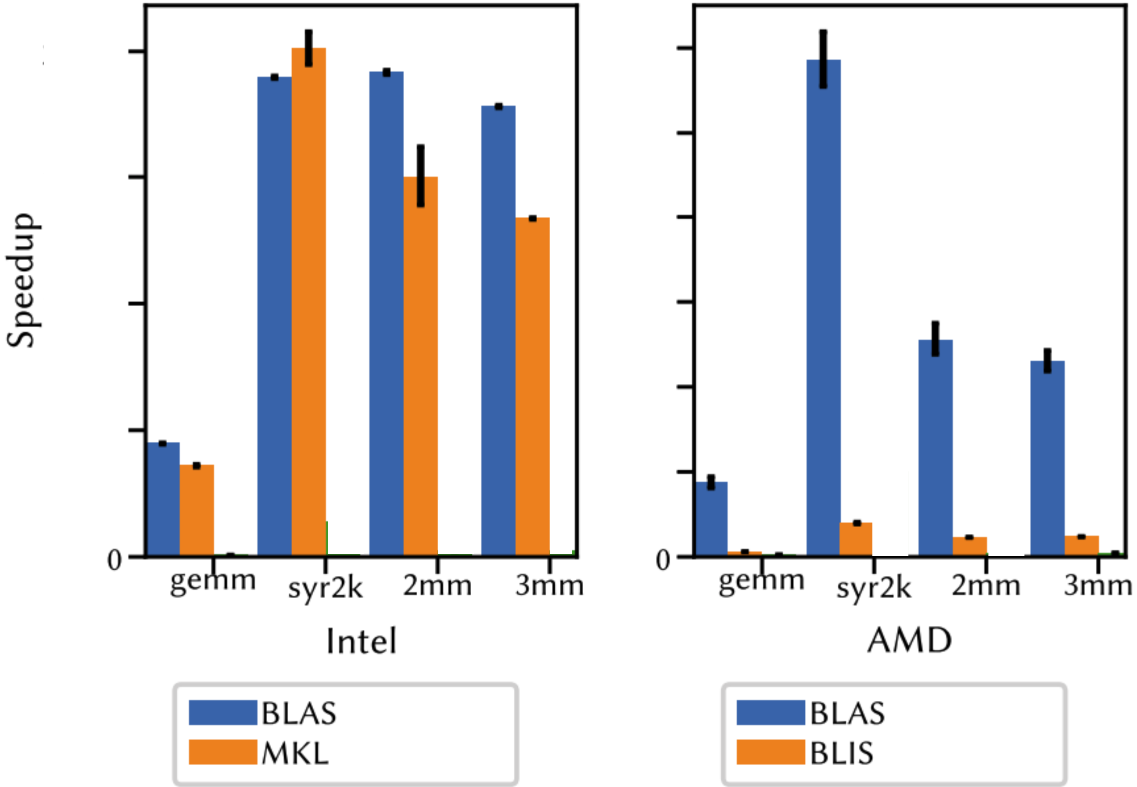
José Nelson Amaral, *University of Alberta, Edmonton, AB, T6G 2S4, Canada*

2020
Volume 1, Number 3

ACM Transactions on
Architecture and
Code Optimization

KernelFaRer: Replacing Native-Code Idioms with High-Performance Library Calls

JOÃO P. L. DE CARVALHO, University of Campinas (UNICAMP), Brazil
BRAEDY KUZMA, IVAN KOROSTELEV, and JOSÉ NELSON AMARAL,
University of Alberta, Canada
CHRISTOPHER BARTON, IBM Corporation, Canada
JOSÉ MOREIRA, IBM Corporation, USA
GUIDO ARAUJO, University of Campinas (UNICAMP), Brazil


Speedup
gemm
syr2k
2mm
3mm
Intel
BLAS
MKL
gemm
syr2k
2mm
3mm
AMD
BLAS
BLIS