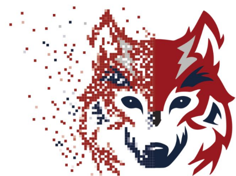
Performance engineering on A64FX with SVE intrinsics

(Early experience on Ookami)

Robert J. Harrison

NSF OAC 1942140

Contact: robert.harrison@stonybrook.edu



OOKAMI

iACS INSTITUTE FOR ADVANCED COMPUTATIONAL SCIENCE

What is Ookami?

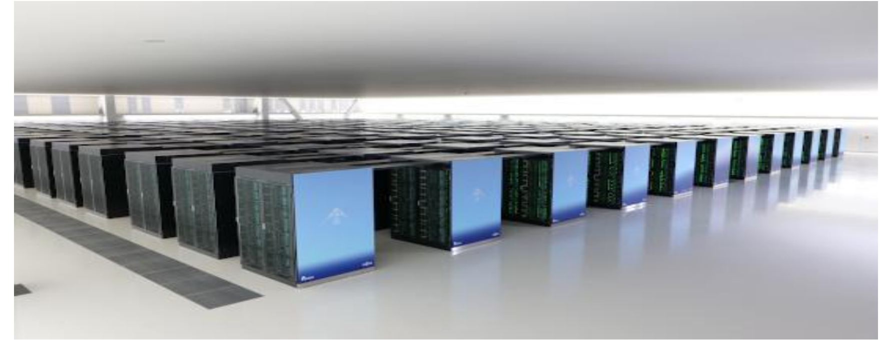
- Test bed for NSF and US researchers
- First deployment of the Fujitsu A64FX Post-K processor outside of Japan
    - ARM64 + SVE (scalable vector extensions)
- Possibly revolutionary new path to exascale emphasizing scientific productivity, performance, and energy efficiency
- New processor & very high-bandwidth memory promise performance of GPUs with programmability of CPUs

Ookami (狼) means wolf in Japanese --- an homage both to the origin of the processor and the Stony Brook seawolf mascot.

Fugaku #1 Fastest computer in the world

First machine to be fastest in all 5 major benchmarks
https://bit.ly/33RLmBK

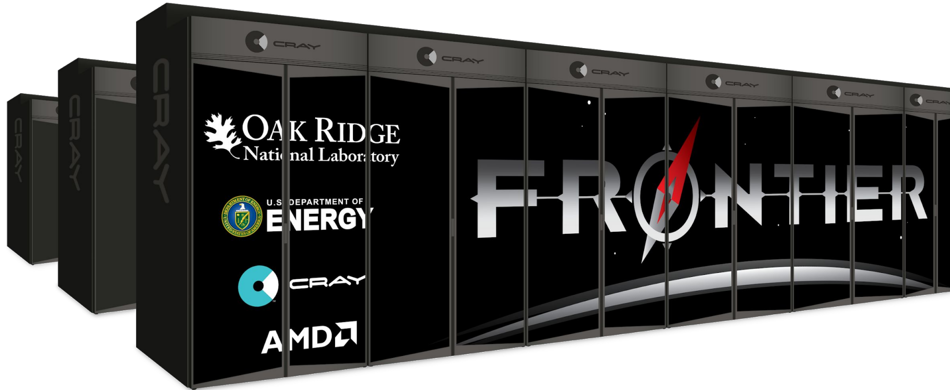
- Green-500 benchmark (11/19)
  https://bit.ly/382Ls9Y
- Top-500 benchmark (6/20)
  https://bit.ly/2RWivXo
  - 415 PFLOP/s in double precision – nearly 3x Summit!!
- HPCG benchmark (6/20)
  https://bit.ly/2RVwDQX
- HPL-AI benchmark (6/20)
  https://bit.ly/308DbzZ
- Graph-500 benchmark (6/20)
  https://bit.ly/3mUoJVY



- 432 racks
- 158,976 nodes
- 7,630,848 cores
- 440 PF/s dp (880 sp; 1,760 hp)
- 32 Gbyte memory per node
- 1 Tbyte/s memory bandwidth/node
- Tofu-2 interconnect

https://aurora.alcf.anl.gov/

- 1.0+ EF in double precision; Intel Xeon + Intel Xe GPU + Intel Optane



https://www.olcf.ornl.gov/frontier/

- 1.5+ EF in double precision; AMD EPYC CPU + AMD Radeon Instinct GPU

Europe and Japan are on a different <u>path</u>

- ARM + SVE
  - https://www.montblanc-project.eu
  - https://www.r-ccs.riken.jp/en/postk/project/outline
  - ARM – 21B units/year sold vs. ~400M for x86
  - Scalable vector extensions – SIMD designed to increase ease of obtaining high performance for HPC and data apps
- A64FX – successful co-design by RIKEN-Kobe+Fujitsu
  - A technology <u>path</u> – not a one-off

Mitsuhisa Sato, "Overview of the Post-K processor,"
http://www.jicfus.jp/jp/wp-content/uploads/2018/11/msato-190109.pdf
"Fujistu high-performance CPU for the Post-K computer"
https://www.fujitsu.com/global/documents/solutions/business-technology/tc/catalog/20180821hotchips30.pdf

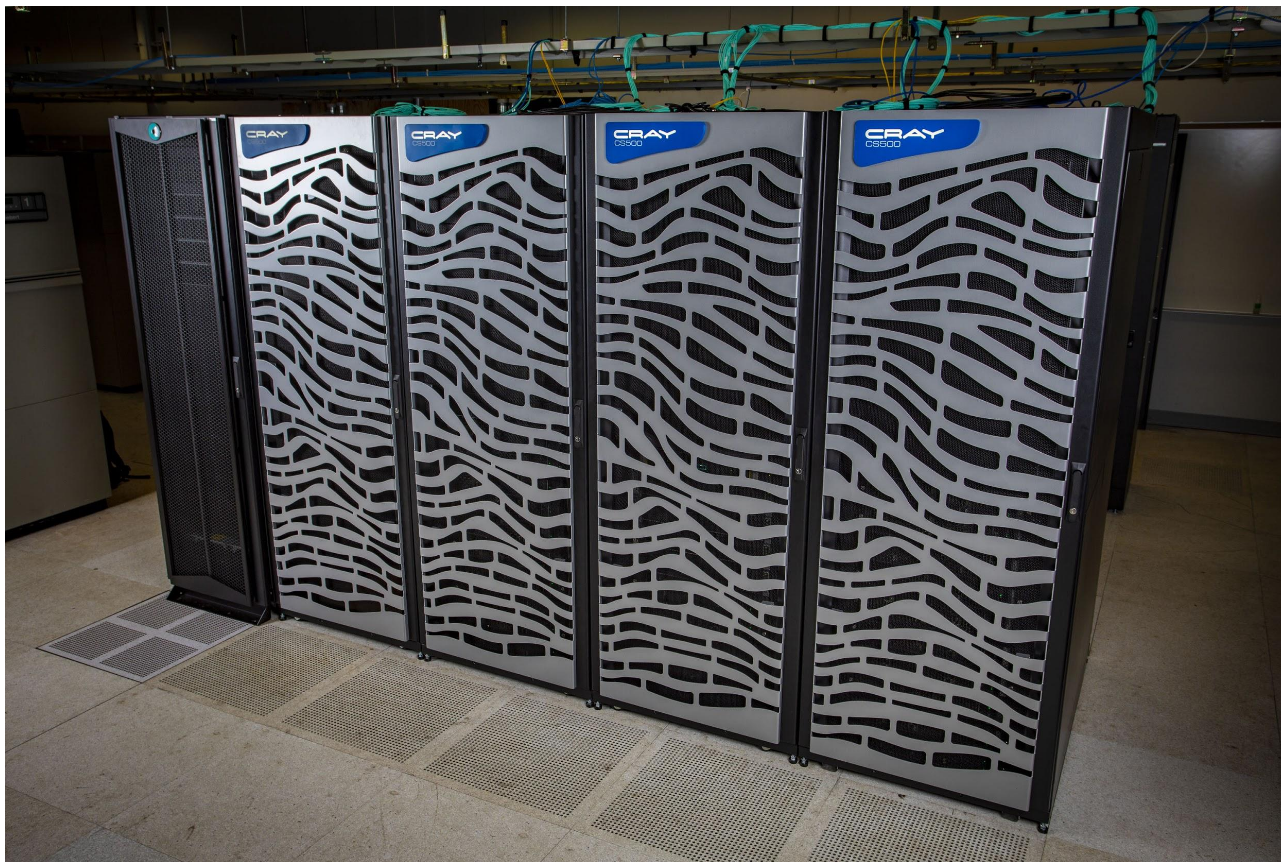
tl;dr

"Programmability of a CPU, performance of a GPU"

Satoshi Matsuoka

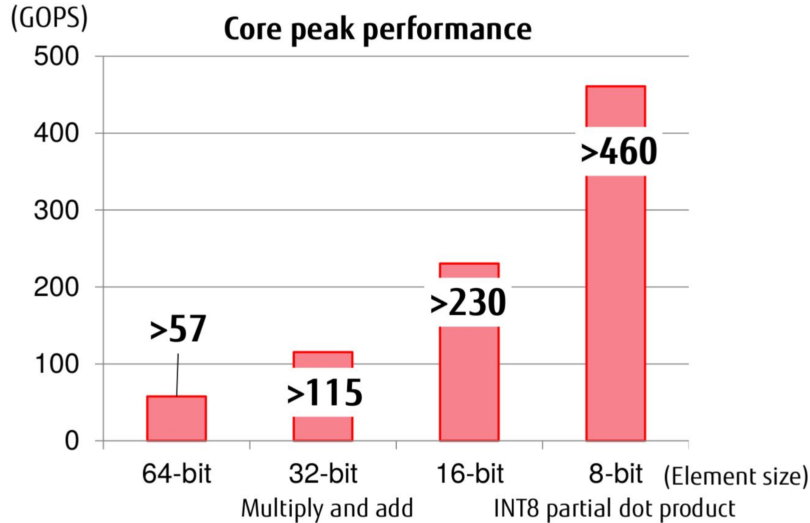
- Blazing fast memory
- Easily accessed performance
- New technology path to exascale

Ookami configuration



Node	
Processor	A64FX
#Cores	48
Peak DP	2.76 TOP/s
Peak INT8	22.08 TOP/s
Memory	32GB@**1TB/s**
System	
#Nodes	176
Peak DP	486 TOP/s
Peak INT8	3886 TOP/s
Memory	5.6 TB
Disk	0.8 PB Lustre
Comms	IB HDR-100

A64FX at a glance

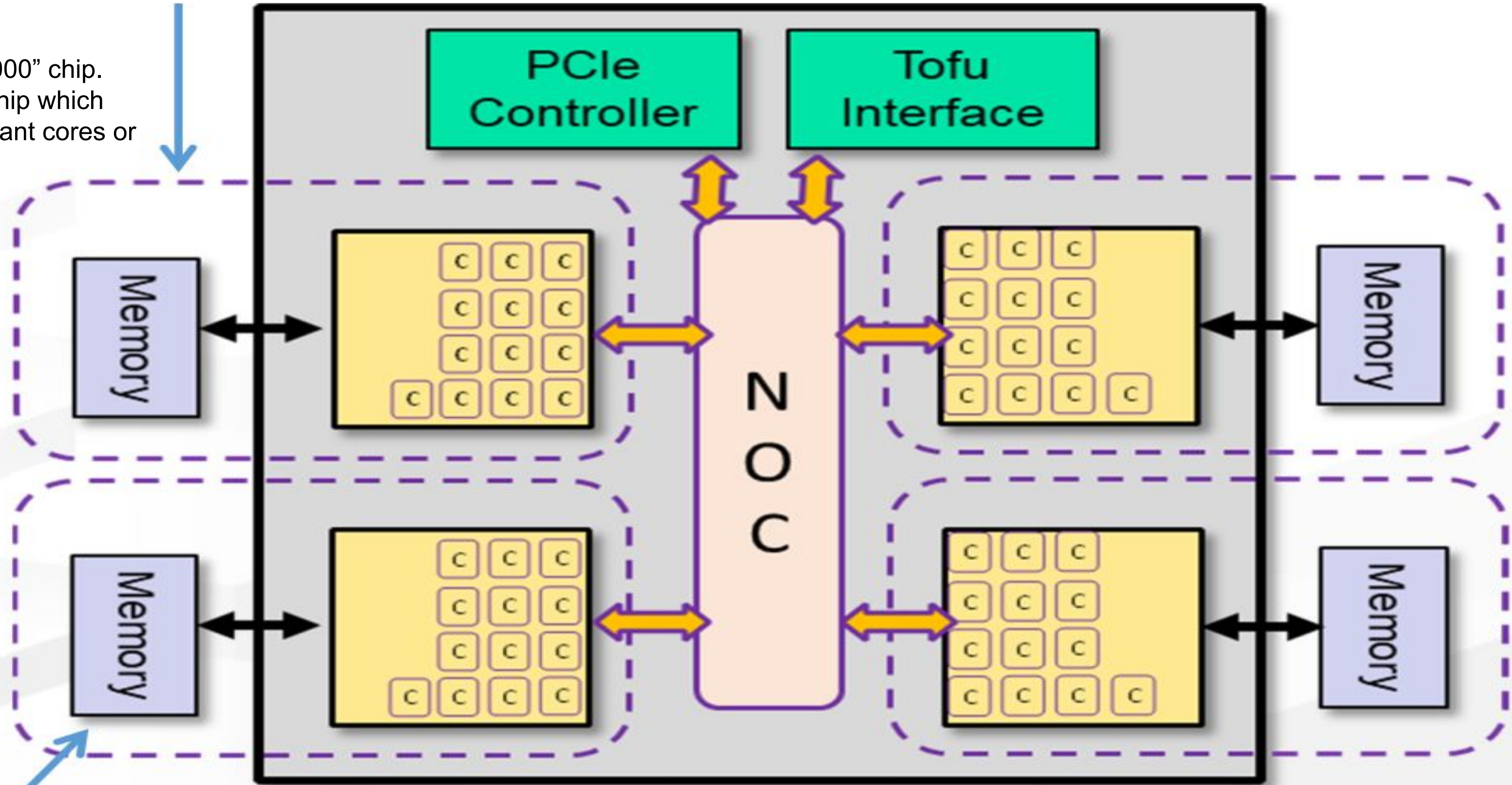


Taken by RJH at SC19

- ARM V8 64-bit
- 512-bit SVE
- 48 compute cores
- 4 NUMA regions
- 32 (4x8) GB HBM @ 1 TB/s
- PCIe 3 (+ Tofu-3) network

A64FX NUMA node architecture

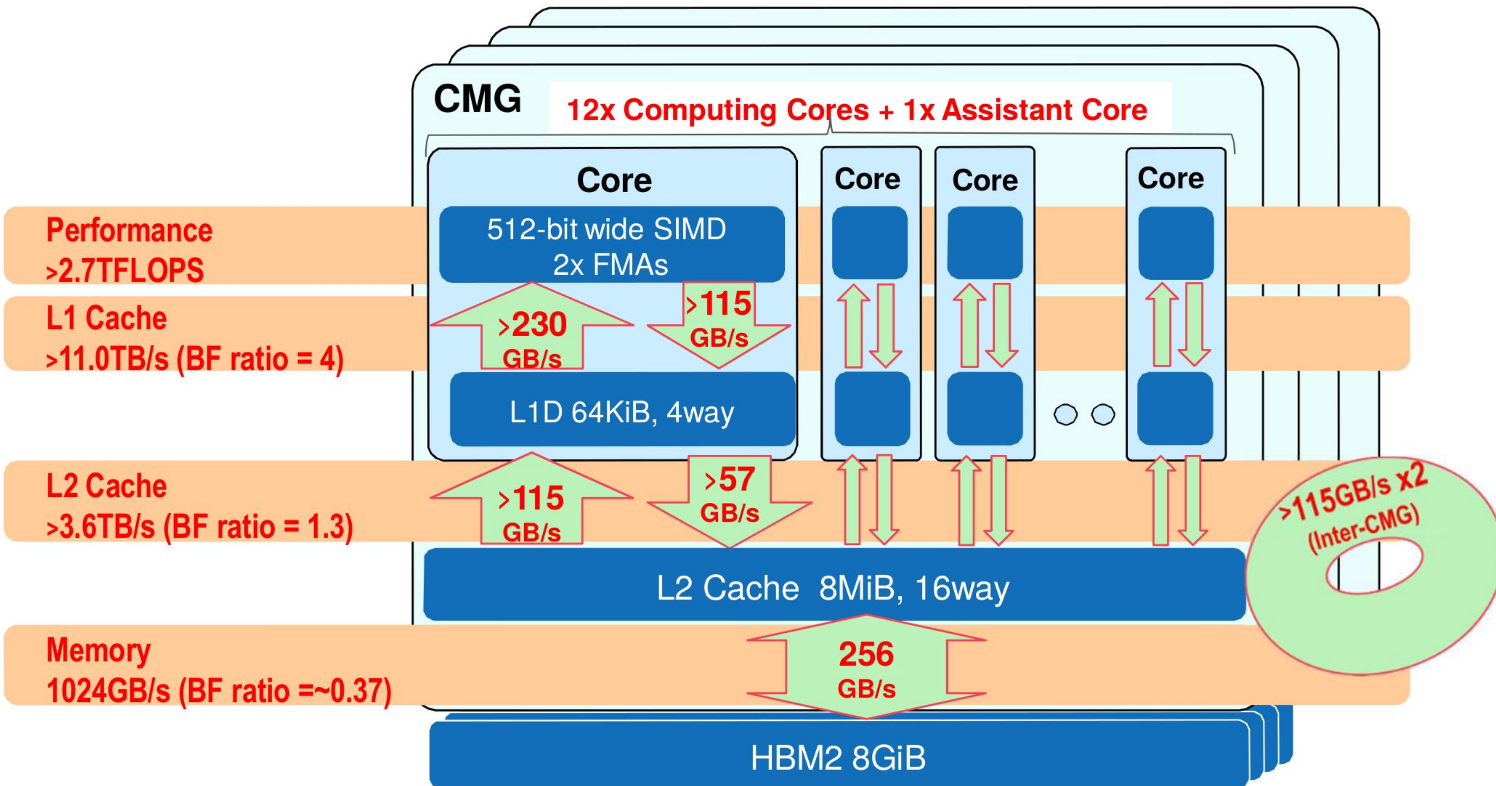
Diagram is of the “1000” chip.
We have the “700” chip which does not have assistant cores or the Tofu interface.



CMG – core memory group

http://www.jicfus.jp/jp/wp-content/uploads/2018/11/msato-190109.pdf

A64FX Core memory group



http://www.jicfus.jp/jp/wp-content/uploads/2018/11/msato-190109.pdf

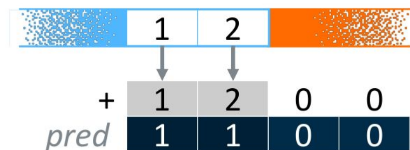
Scalable Vector Extensions (SVE)

- SVE enables Vector Length Agnostic (VLA) programming
  - VLA enables portability, scalability, and optimization
  - The actual vector length is set by the CPU architect
    - Any multiple of 128 bits up to 2048 bits
    - May be dynamically reduced by the OS or hypervisor
- Predicate-centric architecture
  - Predicates are central, not an afterthought
  - Support complex nested conditions and loops.
  - Predicate generation also sets condition flags.
  - Reduces vector loop management overhead.
- SVE was designed for HPC and can vectorize complex structures
  - Gather-load and scatter-store; horizontal reductions
  - SVE begins to tackle traditional barriers to auto-vectorization
    - Software-managed speculative vectorization allows uncounted loops to be vectorized.
    - In-vector serialised inner loop permits outer loop vectorization in spite of dependencies.
- Support from open source and commercial tools

	1	2	3	4
+	5	5	5	5
pred	1	0	1	0
=	6	2	8	4

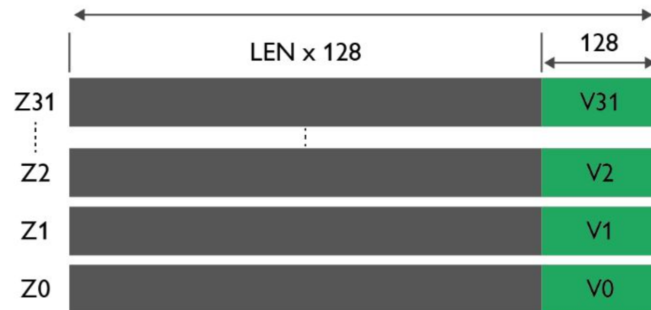
```
for (i = 0; i < n; ++i)
```

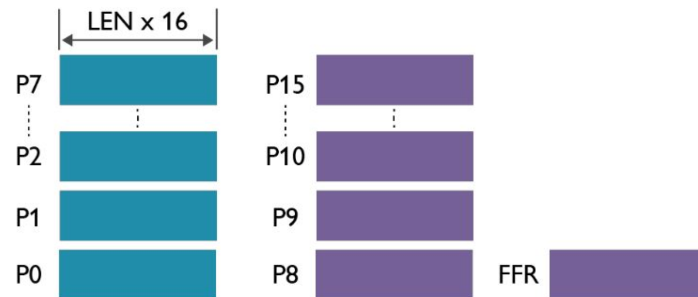
INDEX i	n-2	n-1	n	n+1
CMPLT n	1	1	0	0



SVE Registers

- **Scalable vector registers**
  - `Z0-Z31` extending NEON's 128-bit `V0-V31`.
  - Packed DP, SP & HP floating-point elements.
  - Packed 64, 32, 16 & 8-bit integer elements.
- **Scalable predicate registers**
  - `P0-P15` predicates for loop / arithmetic control.
  - 1/8th size of SVE registers (1 bit / byte).
  - `FFR` first fault register for software speculation.





© 2019 Arm Limited
arm

Naive SVE intrinsics version of unit stride DAXPY

- Per-lane predication
  - Operations work on individual lanes under control of a predicate register.
- Predicate-driven loop control and management
  - Eliminate scalar loop heads and tails by processing partial vectors.
- Vector partitioning & software-managed speculation
  - First Faulting Load instructions allow memory accesses to cross into invalid pages.

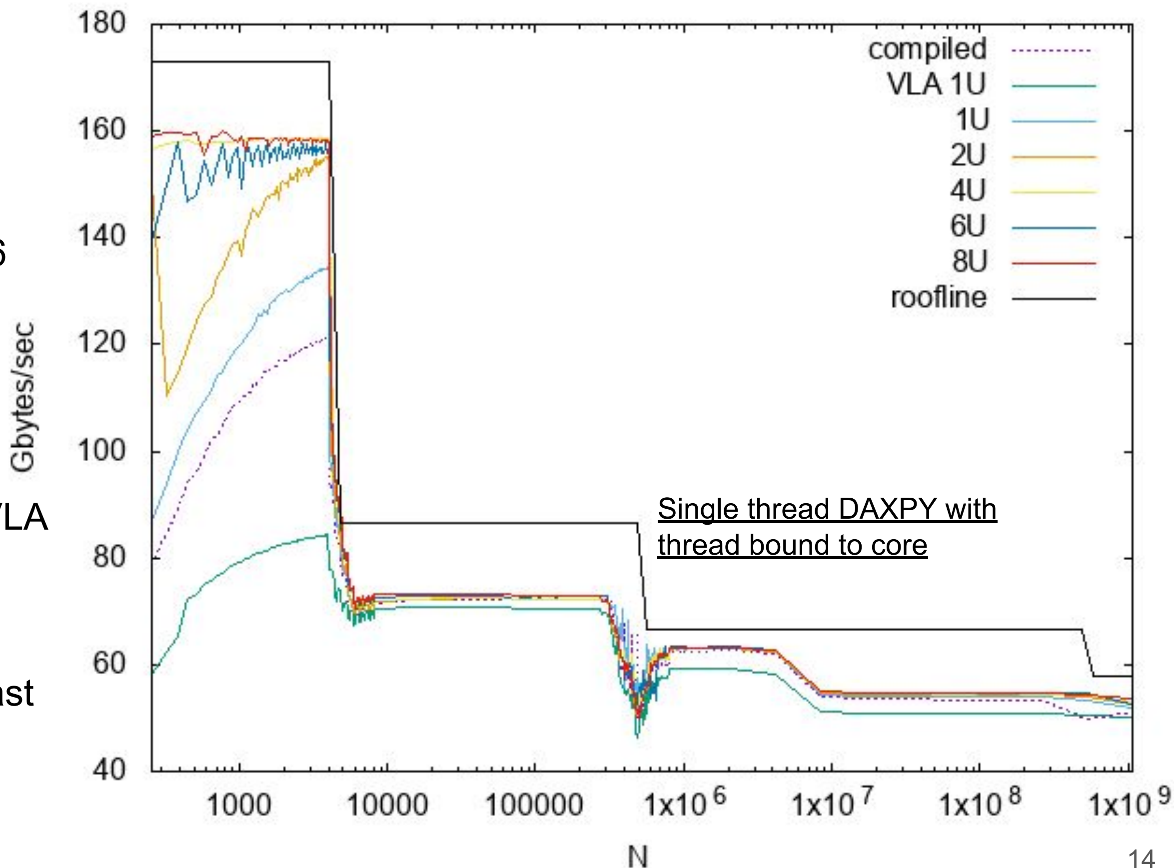
```
void daxpy_1_1(int64_t n, double da,
               double *dx, double *dy) {
    int64_t i = 0;
    svbool_t pg = svwhilelt_b64(i, n);  // [1]
    do {
        svfloat64_t dx_vec = dx_vec svld1(pg, &dx[i]);  // [2]
        svfloat64_t dy_vec = svld1(pg, &dy[i]);  // [2]
        svst1(pg, &dy[i], svmla_x(pg, dy_vec, dx_vec, da));  //[3]
        i += svcntd();  // [4]
        pg = svwhilelt_b64(i, n);  // [1]
    } while (svptest_any(svptrue_b64(), pg));  // [5]
}
```

[1] - Initialize a predicate register to control the loop
[2] - Load some values into an SVE vector, guarded by the loop predicate.
[3] - Perform a floating-point multiply-add operation, and store result.
[4] - Increment i by the number of double-precision lanes in the vector.
[5] - ptest returns true if any lane of the (updated) predicate is active.

https://developer.arm.com/documentation/100891/0612/coding-considerations/using-sve-intrinsics-directly-in-your-c-code

DAXPY single thread

- Multiple implementations
  - Compiled naive loop
  - VLA no unrolling (1U)
  - 512-bit fixed width with 1,2,4,6 and 8-way unrolling
- Compiled naive kernel attains near full main memory B/W
- SVE intrinsics+unroll crucial for in-cache performance
  - Fixed-width SVE faster than VLA
- gcc 11 for SVE intrinsics
- armclang 21
  - Older ARMPL seemed to be compiled but new version is fast
  - armclang generates SVE, but inferior to GCC for intrinsics



SVE DAXPY fixed 512 bit width (no unrolling)

(Illustrative code does not do loop tail since this just replicates the body with an updated predicate).

1. Define predicate true for all lanes.
2. Replicate a across SIMD vector
3. Load x[i], y[i]
4. y[i] += a*x[i]
5. Store y[i]

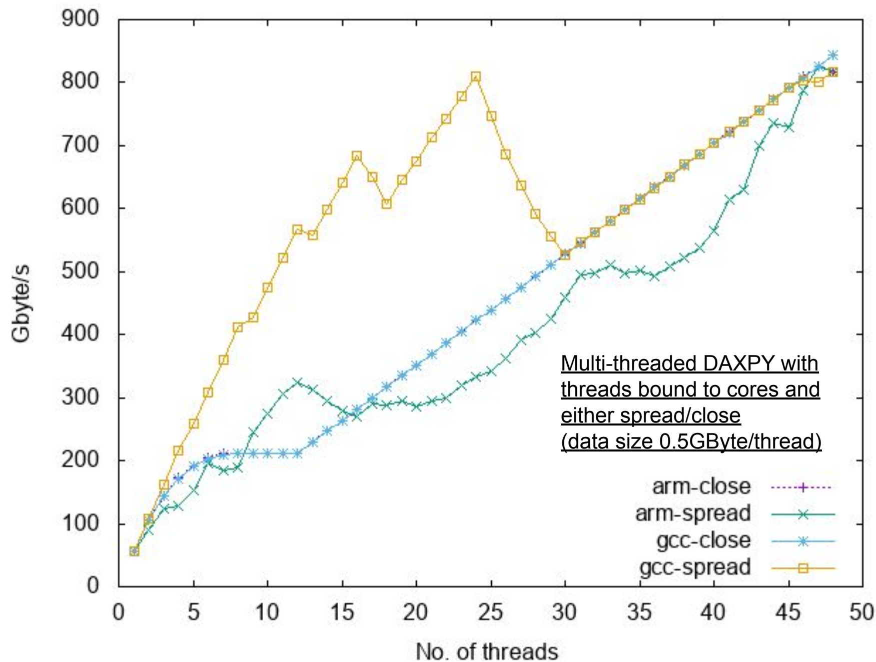
Almost 1:1 correspondence with AVX512

```
void daxpy_1_1_512(int64_t n, double a,
   const double * __restrict__ x,
   double * __restrict__ y)
{
  typedef svfloat64_t vec
       __attribute__((arm_sve_vector_bits(512)));
  typedef svbool_t pred
       __attribute__((arm_sve_vector_bits(512)));

  pred everything=svptrue_b64();        // [1]
  vec avec = svdup_n_f64(a);            // [2]
  for (int i=0; i<n; i+=8,x+=8,y+=8) {
    vec xvec = svld1_f64(everything,x); // [3]
    vec yvec = svld1_f64(everything,y); // [3]
    yvec = svmad_f64_x(everything,avec,xvec,yvec); //[4]
    svstnt1_f64(everything,y,yvec);     // [5]
  }
}
```

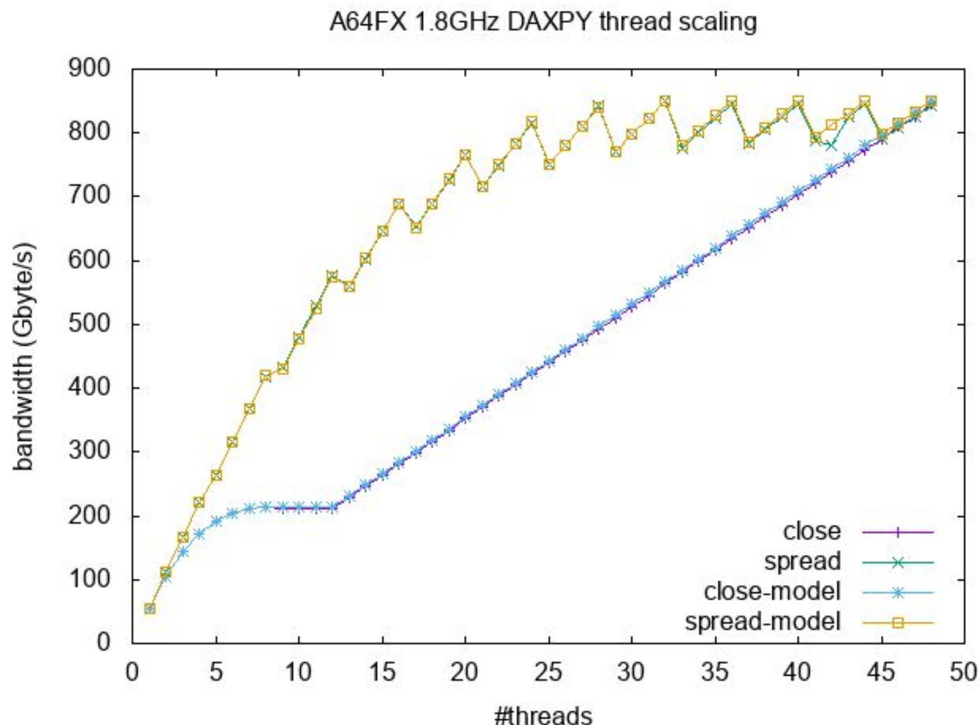
DAXPY multi-thread using OpenMP

- Compiled kernel per thread
- ~0.5 Gbyte data per thread allocated on thread's stack
- Threads bound to cores
- Spread/Close allocation
- ARM 21 and GCC 11
- Close binding working as expected with both compilers
- Spread binding seems to be broken
  - GCC beyond 24 threads seems to revert to close binding
  - ARM compiler seems to not bind at all

- Each CMG delivers about 210GB/s --- 840GB/s total
  - Only attainable with strict attention to data locality
  - About 6 threads/CMG can saturate bandwidth



Ditto using pthreads



Performance model gives perfect agreement

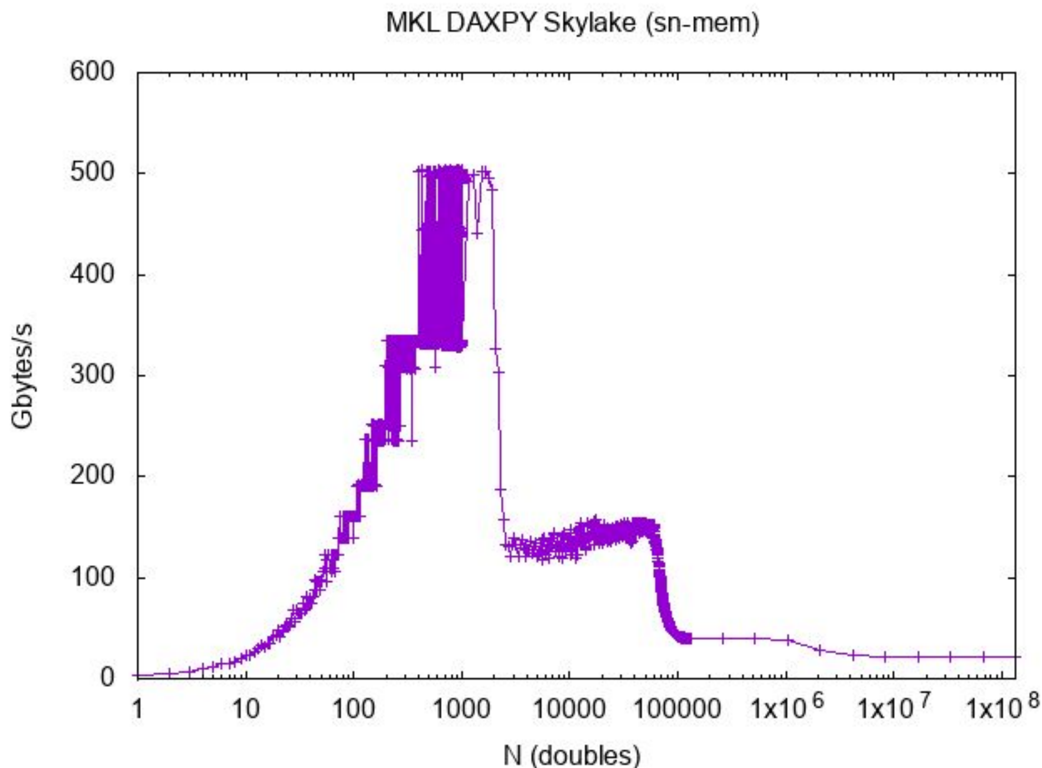
- Each thread is moving the same amount of data
- So the most highly-occupied CMG limits performance
- Take expt. data from close binding with threads1-12.

Can see that

- CMG memory bandwidth is perfectly scalable *with respect for locality*

DAXPY --- comparison with Intel Skylake single thread

- In-cache bandwidth differences
  - Clockspeed - 3.7GHz vs 1.8 GHz
  - Instruction issue
    - Skylake: 2 loads+1 store
    - A64FX: 2 loads or 1 store
- Additional cache levels visible
- Asymptotic 1 thread bandwidth
  - Skylake: 21.1 Gbyte/s
  - A64FX: 53.0 Gbyte/s (2.5x)
- Asymptotic 1 thread bytes/flop
  - Skylake: 0.17
  - A64FX: 0.92 (5.4x)
- Asymptotic 1 socket bandwidth
  - Skylake: 145* Gbytes/s
  - A64FX: 840 Gbytes/s (5.8x)
- Asymptotic 1 socket bytes/flop
  - Skylake: ~0.06 (24 cores@3GHz)
  - A64FX: 0.30 (~5x)



*https://community.intel.com/t5/Software-Tuning-Performance/Stream-benchmark-on-Skylake-SP-Xeon-results-too-low/td-p/1171787

Matrix-transpose times matrix kernel motivation from discontinuous spectral element code (MADNESS)

$$r^{nl+m}_{pqr\ldots} = r^{nl+m}_{pqr\ldots} + \sum_{\mu=1}^{M} \sum_{p'q'r'\ldots=1}^{2k} X^{\mu m_x}_{p'p} Y^{\mu m_y}_{q'q} Z^{\mu m_z}_{r'r} \ldots a^{nl}_{p'q'r'\ldots} \Longrightarrow r = r + \sum_{\mu} \left(\left(a^T X^{\mu} \right)^T Y^{\mu} \right)^T Z^{\mu} \ldots$$

$$r^{nl}_{pqr\ldots} = \sum_{\mu=1}^{M} \sum_{p'q'r'\ldots=1}^{2k} a^{nl}_{p'q'r'\ldots} C_{p'p} C_{q'q} C_{r'r} \ldots \Longrightarrow r = \left(\left(a^T C \right)^T C \right)^T C \ldots$$

- Transformation of all indices in a tensor is efficiently mapped to mTxm kernel
  - Implicit index fusion automatically handles cyclic permutation of indices
- k is order of the polynomial (circa 6 to 10); 1 to 6 dimensions
- In 3D, resultant matrix operations either $(k,k^2)^T*(k,k)$ or $(2k,4k^2)^T*(2k,2k)$
  - Most BLAS libraries are not optimized for these small, highly-rectangular matrices
  - On Intel, *recent* MKL and small-mxm libraries are fast

MADNESS matrix transpose times matrix kernel

- Optimized for small, non-square matrices on single core
  - Code generation using intrinsics (SVE, Neon, AVX2, AVX512) plus auto-tuning
  - On SVE uses fixed 512-bit SIMD based upon feedback from RIKEN team
- Best A64FX performance
  - 53.24 GFLOP/s = 92.4% of single core peak (57.6 GFLOP/s)
  - ni=15, nj=40, nk=124 --- all 3 matrices fit in L1
- Some optimizations still missing
  - Full unrolling of small matrix operations with modular arithmetic for register allocation

Algorithm

```
Tile i loop into cache
    Tile j loop into registers (multiple of SIMD width)
        Tile i into registers
            Zero Cij registers
            For all k
                Load bkj for j in tile
                For i in tile fully unrolled
                    Load Aki and duplicate across register
                    Cij += Aki * Bkj
            Store Cji
```

Config file input to mTxm code generator

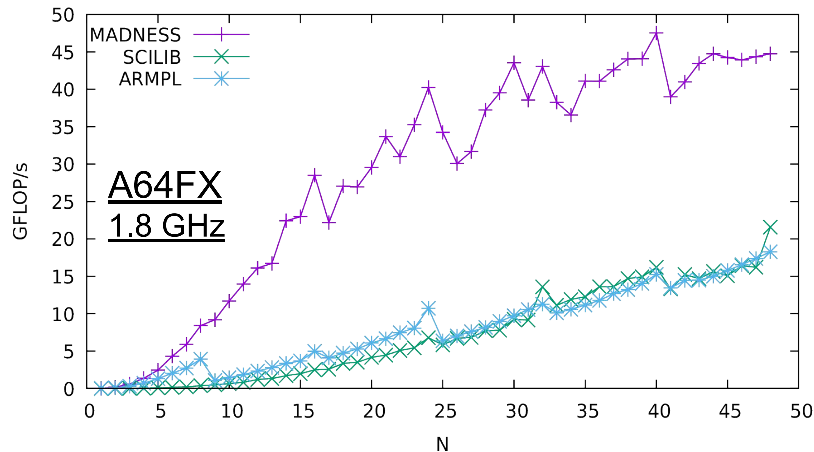
SVE

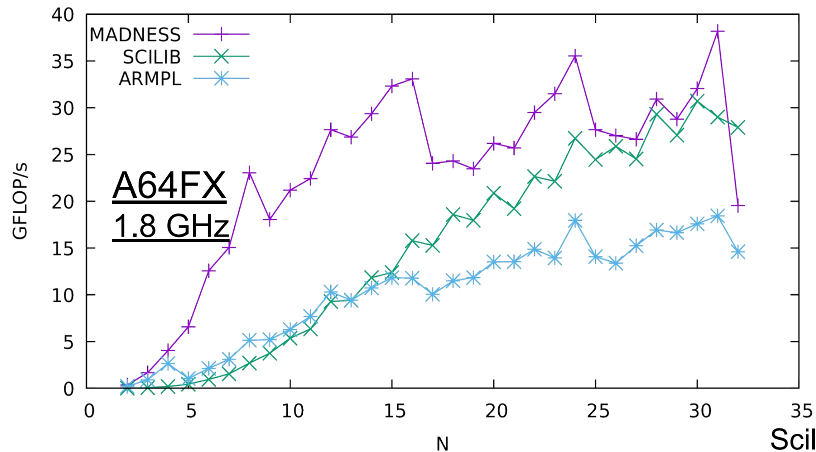
```
REGISTER_TYPE="vec"
REGISTER_WIDTH = 8
NUMBER_OF_REGISTERS = 32
MAX_JTILE = 56
MAX_ITILE = 30

TARGET_ITILE = 6
TARGET_JTILE = 32

CACHE_SIZE = 8192

def zero(register):
   print("%s=svdup_n_f64(0.0); " % register,end="")

def fma(a,b,c):
   print("%s=svmad_f64_x(everything,%s,%s,%s); "%(c,a,b,c),end="")

def load(register,ptr,is_incomplete):
   if is_incomplete:
      print("%s=svld1_f64(mask,%s); " % (register,ptr),end="")
   else:
      print("%s=svld1_f64(everything,%s); " % (register,ptr),end="")

def store(register,ptr,is_incomplete):
   etc.
```

AVX2

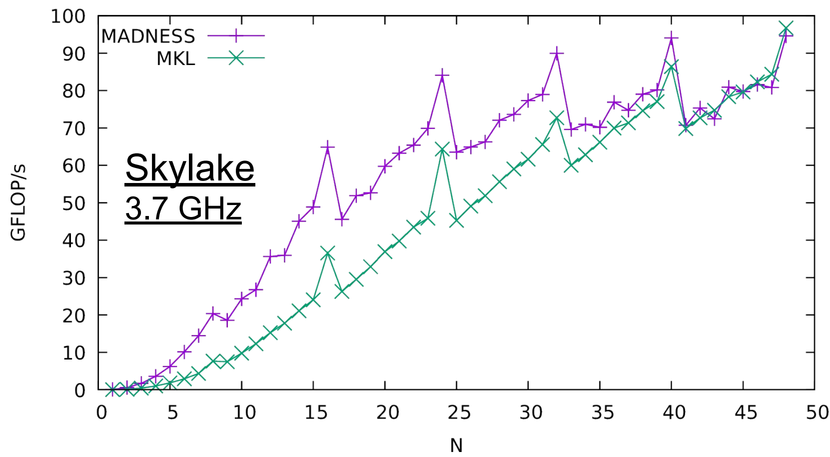
```
REGISTER_TYPE="__m256d"
REGISTER_WIDTH = 4
NUMBER_OF_REGISTERS = 16
MAX_JTILE = 20
MAX_ITILE = 16

TARGET_ITILE = 3
TARGET_JTILE = 16

CACHE_SIZE = 8192 # empirical optimization

def zero(register):
   print("%s=_mm256_setzero_pd(); " % register,end="")

def fma(a,b,c):
   print("%s=_mm256_fmadd_pd(%s,%s,%s); "%(c,a,b,c),end="")

def load(register,ptr,is_incomplete):
   if is_incomplete:
      print("%s=_mm256_maskload_pd(%s,mask); " % (register,ptr),end="")
   else:
      print("%s=_mm256_loadu_pd(%s); " % (register,ptr),end="")

def store(register,ptr,is_incomplete):
   etc.
```

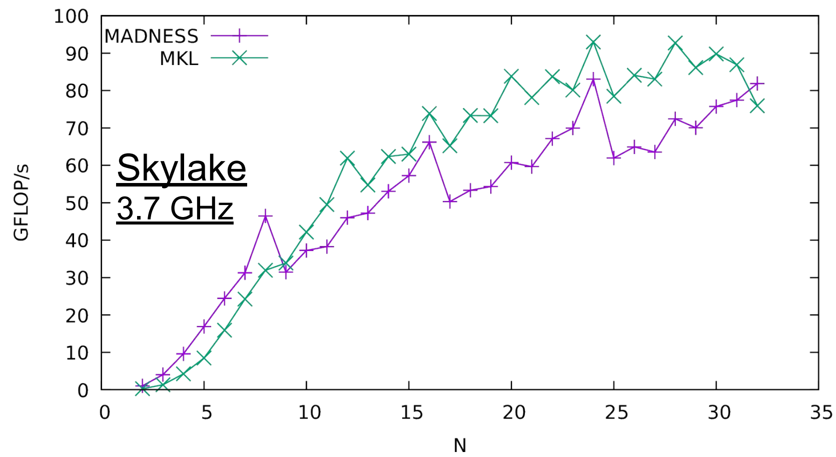
Single core small matrix transpose times matrix





Scilib 10.0.1; armpl 20.3



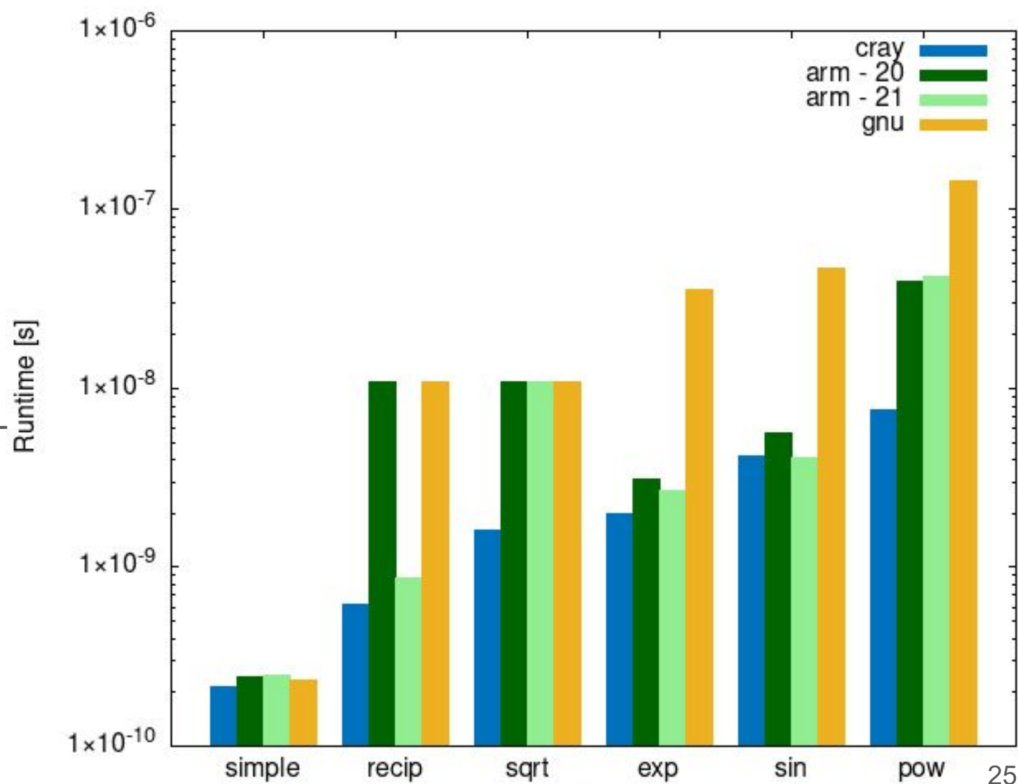


Single core small matrix transpose times matrix - II

- Speed difference relative to Skylake mainly arises from 2x difference in clock speed
    - 32 d.p. FLOP/cycle from both A64FX (SVE512) and Skylake (AVX512)
    - Same L1 cache *load* bandwidths relative to clock speed
    - Memory architecture beyond L1 cache differs but that is not crucial for MxM
- Observe similar ramp up in mTxm performance on both processors
    - Limited by the same algorithm and same code generator

Compiler vectorization

- ARM modified LLVM, recent GNU and Cray compilers all generate SVE
  - Mainline LLVM does not yet
  - All can vectorize with similar performance loops involving arithmetic and “if” tests
  - One main difference is math functions
- Reciprocal/square root
  - Cray uses Newton iteration whereas GNU and older ARM compiler use RECIP/SQRT instructions that are v. slow on A64FX
- GNU compiler vectorizes other functions via libmvec (glibc) which does not yet support SVE and there seems to be a deeper issue for a platform having both VLA (SVE) and fixed (NEON) SIMD.



Evaluation of the exponential function on A64FX

- The performance of many scientific kernels are limited by evaluation of math functions
- Initial investigation for double-precision exponential
  - GCC 10.2.0 - 32 cycles (correctly rounded)
  - ARM 20.3 - 6 cycles
  - CRAY 10.0.1 - 4.2 cycles
  - Intel Skylake icc 19.* - 1.6 cycles
- How fast can we go on A64FX?
  - Can we close the gap to Skylake?

Approximation of the exponential function

- For trigonometric and exponential functions common algorithms work by
  - reducing the argument to a standard small range,
  - using a series expansion to evaluate the function over that range, and
  - scaling the result back to the target value.
- Given `x` find integer `m` and value `r` s.t. `|r|<½ log2` and `x = m log2 + r`
- Then, $\texttt{exp(x)} = 2^{m}\ \texttt{exp(r)}$
- Exponentiating `r` can be done using a series expansion, with 13 terms being required to obtain the required accuracy in double-precision arithmetic.
- Multiplication by 2^{m} is accomplished by adding `m` to the binary exponent.
- Unless extended precision is used or some fix up is performed, the last bit(s) will not be correctly rounded. (1-4 ULPs common error in vector math lib)

SVE instruction FEPEXA - A=Acceleration

The double-precision variant copies the low 52 bits of an entry from a hard-wired table of 64-bit coefficients, indexed by the low 6 bits of each element of the source vector, and prepends to that the next 11 bits of the source element (src<16:6>), setting the sign bit to zero.

Uh?

How does FEPEXA accelerate?

- FEPXA accelerates exponentiation by reducing the number of terms in the series expansion to 5 by reducing the range of r by a factor of 64.
- Write $x = (m+i/64) \log 2 + r$, integer m and $0<=i<64$, with value $|r|<\log2 / 128$
- Then, $\exp(x) = 2^{m+i/64} \exp(r)$
- FEPXA computes $2^{m+i/64}$.
  - It takes 17 bits as input, interpreting the lower 6 bits as i and the upper 11 bits as m.
  - Well almost --- since the binary exponent in an IEEE-754 double-precision number is stored offset by 1023, FEPXA actually wants m+1023 as input.
  - Why 17 bits? Recall that $64=2^6$, and 11 bits are used to store the exponent of a double-precision number.

Reference C implementation

```
double myexp(double x) {
  static const double fac = 0.0108304246962491454596442518978; // log(2)/64
  static const double rfac = 92.3324826168936580710351795840;  // 1/fac
  static const double a0 = 1.0;
  static const double a1 = 1.0;
  static const double a2 = 0.5;
  static const double a3 = 0.1666666666666645339082562230955;
  static const double a4 = 0.0416666972130599706546300218462;
  static const double a5 = 0.00833333915169364528960093698321;

  int k = std::round(x*rfac);
  double r = x - k*fac;

  int m = floor(k/64.0);
  int i = k - m*64;

  return std::exp2(m+i/64.0)*(a0 + r*(a1 + r*(a2 + r*(a3 + r*(a4 + r*a5)))));
}
```

Horner form has minimal op count but Estrin form has more parallelism

SVE macros

```
#include <arm_sve.h>
#include <cmath>

// SIMD vector types
#define F64 svfloat64_t
#define I64 svint64_t
#define U64 svuint64_t
#define MASK svbool_t

// Mask values depending on vector length
#define EVERYTHING svptrue_b64()

// FP ceil, floor, round operations
#define CEIL(mask,v) svrintp_x(mask,v)
#define ROUND(mask,v) svrinta_x(mask,v)
#define FLOOR(mask,v) svrintm_x(mask,v)

// FP convert to integer
#define INT(mask,v) svcvt_s64_x(mask, v)
#define UINT(mask,v) svcvt_u64_x(mask, v)

// Integer convert to FP
#define FLOAT(mask,v) svcvt_f64_x(mask, v)
```

```
// Integer shift to right rounding to -infinity
// i.e., int(floor(value/2**shift))
// shift can be immediate value or vector of values
#define ASR(mask,v,shift) svasr_x(pg,v,shift)

// Duplicate scalar across all elements in vector
#define IDUP(value) svdup_s64(value)
#define FDUP(value) svdup_f64(value)

// Load and store
#define LOAD(mask,ptr) svld1(mask, ptr)
#define STORE(mask,ptr,vec) svstnt1(mask, ptr,
vec);

// result = a*b + c
#define FMA(mask,a,b,c) svmad_f64_x(mask,a,b,c)
#define IMA(mask,a,b,c) svmad_s64_x(mask,a,b,c)

// result = a*b
#define MUL(mask,a,b) svmul_x(mask,a,b)
```

Macro to initialize constants

```
#define INITIALIZE \
  static const double fac = -0.0108304246962491454596442518978;\
  static const double rfac = 92.3324826168936580710351795840;\
  static const double a0 = 1.00000000000000000109448766559;\
  static const double a1 = 1.00000000000000000054724376115;\
  static const double a2 = 0.499999999999328180895493906552;\
  static const double a3 = 0.166666666666517373549704816583;\
  static const double a4 = 0.0416667277594639384346492115235;\
  static const double a5 = 0.00833334351546532331159118269769;\
  F64 vfac = FDUP(fac);\
  F64 vrfac = FDUP(rfac);\
  F64 va0 = FDUP(a0);\
  F64 va1 = FDUP(a1);\
  F64 va2 = FDUP(a2);\
  F64 va3 = FDUP(a3);\
  F64 va4 = FDUP(a4);\
  F64 va5 = FDUP(a5);\
  I64 v1023 = IDUP((int64_t(1023)<<6))
```

Macro for loop body

```
define BODY \
      F64 vx = LOAD(pg, xvec+j); \
      F64 vdk = ROUND(pg, MUL(pg, vx, vrfac));  \
      I64 vk = INT(pg, vdk); \
      F64 vr = FMA(pg,vdk,vfac,vx); \
      F64 vr2 = MUL(pg,vr,vr); \
      F64 vr45 = FMA(pg,va5,vr,va4); \
      F64 vr23 = FMA(pg,va3,vr,va2); \
      F64 vr01 = FMA(pg,va1,vr,va0); \
      F64 vr2345 = FMA(pg,vr45,vr2,vr23); \
      F64 vr012345 = FMA(pg,vr2345,vr2,vr01); \
      vk = svadd_x(pg, vk, v1023); \
      F64 vexpa = svexpa_f64(svreinterpret_u64(vk)); \
      STORE(pg, yvec+j, MUL(pg,vexpa,vr012345))
```

VLA code

```
void vexp_varloop(int64_t n, const double* __restrict__ xvec,
                  double* __restrict__ yvec) {
  INITIALIZE;
  int64_t j;
  MASK pg;
  for (j=0, pg=svwhilelt_b64(j, n);
       svptest_any(svptrue_b64(),pg);
       j+=svcntd(), pg=svwhilelt_b64(j,n)) {
    BODY;
  }
}
```

2.2 cycles/element

512-bit fixed-width code

```
void vexp_varloop(int64_t n, const double* __restrict__ xvec,
                  double* __restrict__ yvec) {
  INITIALIZE;
  int64_t rem = n&7l;
  int64_t n8 = n-rem;
  MASK pg = EVERYTHING;

  for (int64_t j=0; j<n8; j+=8) {BODY;}
  if (rem) {
      int64_t j = n8;
      MASK pg = svwhilelt_b64(j, n);
      BODY;
  }
}
```

2.0 cycles/element which corresponds to about 1.5 cycles less per iteration

<u>2-way unrolling yields 1.9 cycles/element to be compared with 1.6 on Skylake</u>

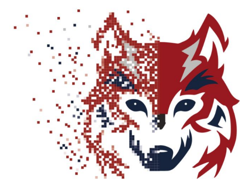
Missing ingredients

- About 6 ulp precision - mostly good enough; better is possible
- Not been tested at the edges of permissible input values
- Some additional masking necessary to ensure out of range large positive values are evaluated to be either NaN or infinity.
- Processing denormalized numbers is very expensive on A64FX, so large negative arguments perhaps should be evaluated directly as zero.
- Some more optimizations are possible
  - Unrolling the Estrin form twice gave only a modest speedup from 2.0 to 1.9 cycles/element.
  - Unrolling the Horner form twice runs at 2.0 cycles/element
- Sleef is a high-quality, portable, vectorized math library that supports SVE
  - https://sleef.org/

Summary

- Many aspects of A64FX performance fully accessible from compiled code
  - Pick the right compiler
  - Vectorizable code
  - Multithreaded code with attention to memory layout and thread binding
  - But there are still gaps from all compilers and especially math/linear algebra libraries
- SVE intrinsics still valuable for
  - Accessing peak performance more consistently
  - For gaps in compiler/library performance
- SVE instrinsics
  - In VLA easier to code than AVX intrinsics and fully portable across all SVE implementations
  - On A64FX VLA is not quite optimal, but the gap is only circa 1-2 cycles/iteration

https://www.stonybrook.edu/ookami/



OOKAMI