A Transformational Approach to Scientific Software: The Mathematics of Array (MoA) Fast Fourier Transform (FFT) with OpenACC

September 15, 2021
OpenACC Annual Summit 2021

Lenore Mullin

Emeritus Professor
University at Albany, SUNY

Chief Technology Officer
MoA: Provably Optimal Tensors

lmullin@albany.edu

Wileam Phan

Scientific Computing Software Engineer
Lawrence Berkeley National Laboratory

wyphan@lbl.gov

Abstract

We extend a methodology for designing efficient parallel and distributed scientific software for GPUs introduced in Rosenkrantz *et al*. [1]. This methodology utilizes sequences of mechanizable algebra-based optimizing transformations. Starting from a high-level algebraic algorithm description in "*A Mathematics of Arrays*" (MoA) [2], abstract multiprocessor plans are developed and refined to specify which computations are to be done by each processor. Starting with the OpenMP program in Fortran 90 produced in Rosenkrantz *et al*. [1], we extend it to include OpenACC for GPU support. Our studies show what is needed in OpenACC, support in Fortran compilers for GPUs, and what issues we encountered and resolved.

[1] Harry B. Hunt, Lenore R. Mullin, Daniel J. Rosenkrantz, and James E. Raynolds (2008),
"A Transformation-Based Approach for the Design of Parallel and Distributed Scientific Software: The FFT"
arXiv:0811.2535

[2] Lenore R. Mullin (1988), "A Mathematics of Arrays", PhD dissertation. Syracuse University. doi:10.5555/915213

What is Mathematics of Arrays (MoA)?

- **Mathematics of Arrays** (**MoA**) is a **theory** of **arrays**, where ***everything*** can be written in terms of ***array shapes*** and ***index composition*** (“***Psi***” ***reduction***).
- MoA makes an ***ideal*** formulation of computation when combined with **Lambda calculus** [3].
- Both MoA and Lambda calculus possess the **Church–Rosser property** [4].

[3] Klaus Berkling, “Arrays and the Lambda Calculus”, (1990), Technical Report 93, Syracuse University. https://surface.syr.edu/eecs_techreports/93/

[4] Benjamin Chetioui, Lenore Mullin, Ole Abusdal, Magne Haveraaen, Jaakko Järvi, and Sandra Macià (2019), “Finite difference methods fengshui: alignment through a mathematics of arrays”, *ARRAY 2019: Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*. doi:10.1145/3315454.3329954

What is Mathematics of Arrays (MoA)? *(cont'd)*

- MoA syntax is heavily inspired by the array algebra in **Ken Iverson**'s **APL programming language** [5].
- When designing and implementing the APL machine, **Phil Abrams** introduced the idea of using ***array shapes*** [6].
- **Lenore Mullin** added ***closure*** on the algebra [2].

[2] Lenore R. Mullin (1988), "A mathematics of arrays", PhD dissertation. Syracuse University. doi:10.5555/915213

[5] Kenneth E. Iverson (1980), "Notation as a tool of thought", 1979 ACM Turing Award lecture, *Communications of the ACM*, **23** (8), 444–465. doi:10.1145/358896.358899

[6] Philips S. Abrams, "An APL machine" (1970), technical report SLAC-114. Stanford University. https://www.slac.stanford.edu/pubs/slacreports/reports07/slac-r-114.pdf

The Cooley–Tukey Fast Fourier Transform (FFT) Algorithm

From van Loan [7], p. 46, Algorithm (1.6.2):

$$
\begin{aligned}
& x \leftarrow P_n x \\
& \text{for } q = 1 : t \\
& \qquad L \leftarrow 2^t; r \leftarrow n/L \\
& \qquad x_{L \times r} \leftarrow B_L x_{L \times r} \\
& \text{end}
\end{aligned}
$$

$$
B_L = \begin{bmatrix} I_{L_0} & \Omega_{L_0} \\ I_{L_0} & -\Omega_{L_0} \end{bmatrix}
$$

$$
\begin{aligned}
L &= 2^q \\
r &= n/L \\
L_0 &= L/2
\end{aligned}
$$

$$
\Omega_{L_0} = \begin{bmatrix} 1 & & & \\ & \omega_L & & \\ & & \ddots & \\ & & & \omega_L^{L_0 - 1} \end{bmatrix}
$$

[7] Charles van Loan (1992), “Computational Frameworks for the Fast Fourier Transform”, Frontiers in Applied Mathematics, SIAM. doi:10.1137/1.9781611970999

The Cooley–Tukey Fast Fourier Transform (FFT) Algorithm *(cont'd)*

In principle, any FFT algorithm could be chosen,
but we started with this particular one because:

- It uses **arrays** effectively.
- It uses the **Kronecker product** (= outer product in MoA).
- It is a **simple** operation (1-dimensional radix-2 FFT) that can be easily extended to *N* dimensions.
- It is one of the **most popular** algorithms for the FFT.
- Van Loan is a respected expert on arrays.

MoA and Transformations

- MoA uses a ***systematic***, ***algebraic*** design methodology to reduce the FFT algorithm into a semantic **denotational normal form** (**DNF**).
- Next, we employ "***dimension lifting***" to get the **operational normal form** (**ONF**).
  - It describes the problem and ***partitioning*** of ***data*** over ***processors*** and ***memory hierarchies***.
  - It allows one to mathematically prove the ***efficiency*** and ***correctness*** of a given algorithm as measured in terms of a set of metrics (such as processor/network/memory speeds).
- Such an approach allows the average programmer to achieve ***high-level optimizations*** similar to those used by compiler writers, e.g. the notion of tiling.
- We envision scientific programs in the future will be developed in an ***interactive*** development environment that combines ***human judgment*** with ***compiler-like analysis***, such that transformation and verification of its correctness can be done ***mechanically***.

The bit reversal permutation

- Notice the first step of the Cooley-Tukey algorithm is the ***bit reversal*** permutation $x \leftarrow P_n x$.
- van Loan [7], p. 39, Algorithm (1.5.1):

  For $t \geq 1$ and $0 \leq k < 2^t$,

  $j \leftarrow 0; m \leftarrow k$

  for $q = 0 : t - 1$

  $\quad s \leftarrow \text{floor}(m/2)$

  $\quad \{b_q = m - 2s\}$

  $\quad j \leftarrow 2j + (m - 2s)$

  $\quad m \leftarrow s$

  end

- We developed this algorithm with MoA:

$$P_n = \vec{i}\,\psi \bigcirc\!\!\!\!\!\backslash \Big((\langle t \rangle\, \rho\, 2)\, \rho\, (\iota 2^t) \Big) \qquad 0 \overset{*}{\leq} \vec{i} \overset{*}{<} (\langle t \rangle\, \rho\, 2)$$

- The implementation uses bit shifts:

```
revivec = 0; k = i
DO j = 1, t
  revivec = ISHFT( revivec, 1 )
  revivec = IOR( revivec, IAND( k, 1_dl ) )
  k = ISHFT( k, -1 )
END DO
revivec = k
```


[7] Charles van Loan (1992), “Computational Frameworks for the Fast Fourier Transform”, Frontiers in Applied Mathematics, SIAM. doi:10.1137/1.9781611970999

The base CPU-only version

- Rosenkrantz *et al*. preprint [1] details the rationale and development from the original Cooley-Tukey algorithm to the MoA version.
- The Operational Normal Form (ONF) from MoA enables one to choose the block size that gives the ***best performance*** for any individual machine -- assuming intentional information can be processed by a compiler.
- The generic design of the program is then implemented in Fortran with three versions: **sequential**, shared memory (**OpenMP**), and distributed memory (**MPI**).

[1] Harry B. Hunt, Lenore R. Mullin, Daniel J. Rosenkrantz, and James E. Raynolds (2008), "A Transformation-Based Approach for the Design of Parallel and Distributed Scientific Software: The FFT", arXiv:0811.2535

The GPU-enabled version with OpenACC

- The GPU has its *unique* memory hierarchy, which is arguably a ***mix*** between shared memory and distributed memory.
  - Device memory is distinct from host memory (unless when using unified memory, which we chose *not* to use).
  - Global device memory is shared between execution units (SMs on NVIDIA, CUs on AMD, etc.)
  - On NVIDIA GPU, CUDA shared memory is shared between threads in the same block.
- For this reason, we start with the ***shared memory model*** and the OpenMP version of the code, and port it for GPUs using OpenACC.

Why use OpenACC?

- **Ease of use**: just “sprinkle” some directives at the right places, and the code is GPU-enabled!
- Ability to maintain a **single source code** for both the CPU-only and the GPU-enabled versions.
- **Portability**: same code can run on both NVIDIA GPU (`pgf90/nvfortran`) and AMD GPU (`gfortran` ≥ 10).
  Rumor has it that Intel GPU support is coming soon too.

Favorite OpenACC features

- ***Full Fortran support*** (unlike HIP or DPC++)
- Manual ***control*** over ***grid*** and ***block size*** via `num_gangs`, `num_workers`, and `vector_length` clauses

```
:
!$acc parallel num_gangs(ngang) num_workers(nworker) vector_length(veclen)
:
```

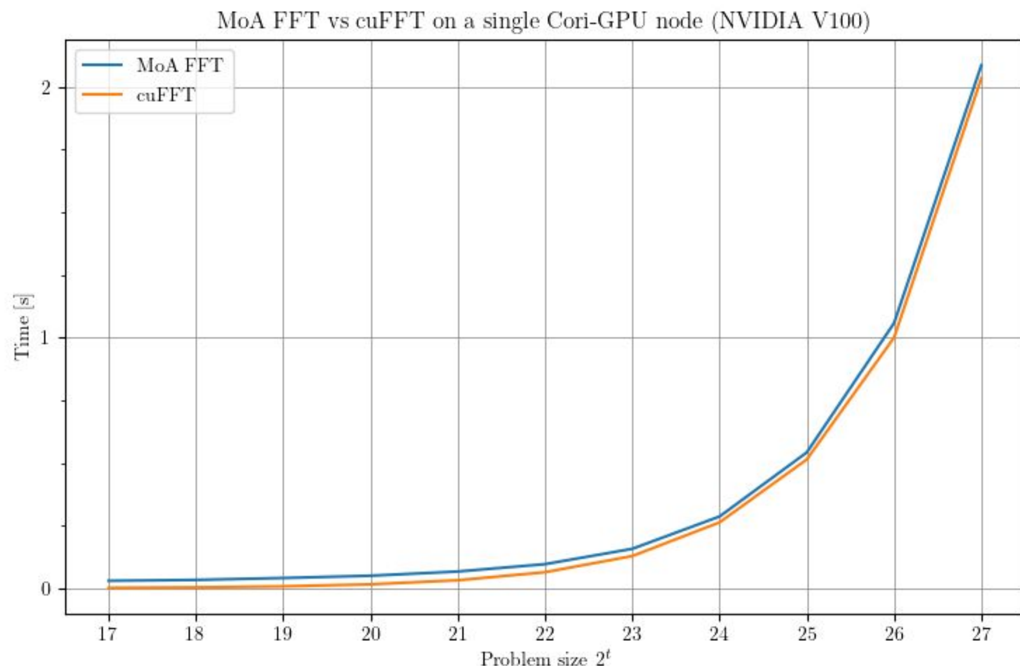
```
116, Generating Tesla code
    121, !$acc loop gang(1024)   ! blockidx%x
    123, !$acc loop worker(8)    ! threadidx%y
    125, !$acc loop vector(128)  ! threadidx%x
```

Favorite OpenACC features *(cont'd)*

- Competitive ***performance*** w.r.t. CUDA on NVIDIA GPUs

```
Reading times-moafft-size-210907.dat :
log2n init       h2d      bitrev    fft_fwd    d2h
:
20   0.007540  0.030121  0.000293  0.009980  0.003912
21   0.015312  0.033479  0.000464  0.011414  0.008008
22   0.029750  0.039017  0.000761  0.012614  0.015826
23   0.060749  0.050531  0.001401  0.014182  0.032252
24   0.124182  0.070106  0.002615  0.026617  0.064424
:

Reading times-cufft-size-210908.dat :
log2n init       h2d       plan     fft_fwd    d2h
:
20   0.009177  0.003474  0.001058  0.000209  0.003718
21   0.018347  0.006826  0.001065  0.000333  0.007214
22   0.035533  0.013427  0.001902  0.000578  0.014378
23   0.071028  0.027496  0.002069  0.001195  0.028409
24   0.147658  0.054567  0.002066  0.002322  0.057793
:
```



Favorite OpenACC features *(cont'd)*

- cache directive for using CUDA ***shared memory***

```
!$acc loop vector collapse(2) private( c, d ) independent
do i = 0, size-1, Lnew
  do j = 0, (Lnew/2)-1
    !$acc cache( weight_p, zblock_p )
    c = weight_p(j) * zblock_p( i + j + Lnew/2 )
    d = zblock_p( i + j )
    zblock_p( i + j ) = d + c
    zblock_p( i + j + Lnew/2 ) = d - c
  end do ! j
end do ! i
```

OpenACC wishlist

- ***Array reduction*** over an arbitrary axis (supposedly introduced in OpenACC 2.7, but we can't seem to get it to work…)

```
integer :: i, j, m, n
real, allocatable :: vec(:), mat(:,:)
:
allocate( vec(m), mat(m,n) )
!$acc data create( vec, mat )
:
!$acc parallel
!$acc loop gang
do i = 1, m
  vec(i) = 0.0
  !$acc loop vector reduction(+:vec)
  do j = 1, n
    vec(i) = vec(i) + mat(i,j)
  end do
end do
```

OpenACC wishlist

- ***Array reduction*** over an arbitrary axis (supposedly introduced in OpenACC 2.7, but we can’t seem to get it to work…)
- Manual ***synchronization*** routines

```
__syncwarp()

__syncthreads()

cudaStreamSynchronize() == !$acc wait

cudaDeviceSynchronize()
```

OpenACC wishlist

- ***Array reduction*** over an arbitrary axis (supposedly introduced in OpenACC 2.7, but we can't seem to get it to work…)
- Manual ***synchronization*** routines
- No "***hidden***" cost of OpenACC runtime library initialization

```
! Notice we have moved OpenACC runtime
! initialization (~300 ms) outside of
! the timing region for initialization

!$acc init

t0 = now()

do i = 0, n - 1
  z(i)= cmplx( i, 0._dd, kind=dz )
end do

t1 = now()
time_init = t1 - t0
```

OpenACC wishlist

- ***Array reduction*** over an arbitrary axis (supposedly introduced in OpenACC 2.7, but we can't seem to get it to work...)
- Manual ***synchronization*** routines
- No "***hidden***" cost of OpenACC runtime library initialization
- Support for ***pinned memory***

```
! CUDA Fortran
ATTRIBUTES(pinned) :: x
```

OpenMP target offload?

- Once compiler support matures for OpenMP target offload,
  we will consider porting from OpenACC into OpenMP target offload.
- Doing this would enable ***concurrent parallelism*** for both multicore CPUs and accelerators (GPUs) under the ***same programming model***.
- Automated ***runtime translation*** tools (e.g. `clacc` and `flacc` [8]) are welcome too!

[8] https://csmd.ornl.gov/project/clacc

Acknowledgements

This research is supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research program under Award Number DE-SC-0021515.

We are thankful for NERSC resources (Cori-GPU), training, and technical support that provided the venue for our research.

In particular, we would like to thank Max Katz (NVIDIA) and
Rishi Khan (Extreme Scale Solutions) for their expert advice relating to GPUs.

We also thank all members in MoA Global Team on Slack for insightful conversations that have contributed to our success.

We are grateful to the OpenACC User Group (OpenACC Hackathons on Slack) and the Fortran community (Discourse forum) for working with us.
Without their valuable feedback, we would not be able to create a product useful for all.