

BACHELOR'S THESIS COMPUTING SCIENCE

# Level 1 BLAS routines in SaC

CASPER FABRITIUS  
s1023960

April 6, 2023

*First supervisor/assessor:*  
Prof. dr. Sven-Bodo Scholz

*Second assessor:*  
Dr. Pieter Koopman

Radboud University



## **Abstract**

This bachelor's thesis is about the creation and evaluation of an implementation of the level 1 BLAS routines in the programming language SAC. BLAS (Basic Linear Algebra Subprograms) is a specification for common linear algebra operations. The level 1 routines in particular specify operations on vectors and scalars. SAC (Single-Assignment C) is a purely functional programming language with imperative style syntax that specialises in operating on arrays. SAC is designed with three core goals in mind: 1. High performance rivalling hand-optimised low-level code. 2. High productivity allowing the programmer to write programs easily with high-level syntax. 3. High portability allowing a SAC program to run on many different architectures with a single source code. The goal of this thesis is to put the high performance promise to the test. By implementing the BLAS routines, comparisons can be made to other hand-optimised BLAS implementations. By comparing the performance of the BLAS implementation in SAC to such a hand-optimised implementation, conclusions can be drawn on the performance of SAC as a whole, specifically in relation to vectors.

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Single-Assignment C (SaC) . . . . .	3
1.2	Basic Linear Algebra Subprograms (BLAS) . . . . .	4
1.3	Research . . . . .	5
<b>2</b>	<b>Implementation</b>	<b>6</b>
2.1	Types . . . . .	6
2.2	Length Parameter . . . . .	7
2.3	Increment Parameter . . . . .	7
2.4	Loops . . . . .	8
2.5	Correctness . . . . .	9
2.6	Optimisation . . . . .	9
2.7	Routines . . . . .	10
2.7.1	drotg . . . . .	10
2.7.2	drotmg . . . . .	10
2.7.3	drot . . . . .	11
2.7.4	drotm . . . . .	12
2.7.5	dswap . . . . .	13
2.7.6	dscal . . . . .	14
2.7.7	dcopy . . . . .	14
2.7.8	daxpy . . . . .	15
2.7.9	ddot . . . . .	15
2.7.10	dsdot . . . . .	16
2.7.11	dnrm2 . . . . .	16
2.7.12	dznrm2 . . . . .	17
2.7.13	dasum . . . . .	18
2.7.14	idamax . . . . .	18
<b>3</b>	<b>Performance Analysis</b>	<b>19</b>
3.1	Performance Quantification . . . . .	20
3.2	Optimised BLAS Library . . . . .	20
3.3	Obtaining Performance Data . . . . .	21

<b>4</b>	<b>Results</b>	<b>22</b>
4.1	Input Size . . . . .	23
4.2	Multi-threading . . . . .	24
4.3	Increment Parameters . . . . .	27
4.4	Compiler Optimisations . . . . .	28
<b>5</b>	<b>Conclusions</b>	<b>29</b>
<b>A</b>	<b>System Specifications</b>	<b>32</b>
<b>B</b>	<b>Graphs</b>	<b>33</b>
B.1	Input Size . . . . .	33
B.2	Thread Count . . . . .	40
B.3	Increment Parameters . . . . .	46
B.4	Compiler Optimisations . . . . .	50

# Chapter 1

## Introduction

### 1.1 Single-Assignment C (SaC)

SaC is a purely functional programming language with imperative style syntax (similar to languages such as C or C++). SaC is also an array programming language, which can be seen in the following three key features of the language:

1. All data in SaC programs is represented by arrays. This entails that standard functions and operators (such as  $+$  and  $*$ ) work on arrays as well.
2. Arrays in SaC are *call by value*. In many other programming languages with similar syntax, such as C, C++, JavaScript or Python, when an array is passed to a function, the function only gets access to a reference pointer to the data of that array in memory. When changes to that array are made, those changes affect the original version of the array, thereby performing a side-effect. This is called *call by reference*. In SaC however, when an array is passed to a function, the function gets access to a copy of the array<sup>1</sup>. When changes are made to this array within the function, it does not affect the original array, thereby preserving the functional nature of the language.
3. There is support for truly multi-dimensional arrays in SaC[2]. For example, instead of accessing elements in an array through individual indices for every dimension, SaC supports so called *index vectors*, which can scale to any dimensionality.

---

<sup>1</sup>This only happens conceptually. In practice, the array is only copied as is necessary[1].

SAC is designed with three core goals in mind[2][3]:

1. High Performance: To deliver performance rivaling hand-optimised low-level code.
2. High Productivity: To write programs easily without burdening the programmer with low-level implementation details.
3. High Portability: To run on many different architectures with a single source code.

Programs written in SAC can be parallelised without requiring any changes to the source code. This is accomplished with one of its unique language constructs, the `with-loop`[1][2]. The `with-loop` works similarly to a list comprehension in other languages, but it is more flexible. It allows the programmer to set a specific range for the output array, to define multiple ranges which allows for different partitions of the output array to be calculated differently, and to generate multi-dimensional arrays within the same `with-loop`. In a `with-loop`, every value of the output array is computed independently of the other values. This guarantees that race conditions will not occur and is what allows the code to be parallelised without any modifications.

## 1.2 Basic Linear Algebra Subprograms (BLAS)

BLAS is a specification for a set of routines that perform common linear algebra operations. It was originally developed starting in 1979 as a library for Fortran[4]. Afterwards it was standardised in a reference implementation (which is also written in Fortran). While this reference implementation is relatively portable, it is not well optimised for performance. For this reason (and because BLAS is used *everywhere* in software), many different hand-optimised libraries have been made, all targeting various specific architectures. These optimised libraries sacrifice portability for better performance.

Additionally, all routines in the specification also come in four different variants, each variant using a different datatype; `single` (i.e. `float`), `double`, `complex` and `double complex`. For this thesis, only the `double` variant is used.

The specification is split into three different *levels*, each level focused on routines with increasing complexity. The level 1 routines only concern vector and scalar operations. The Level 2 routines build upon the level 1 routines by introducing matrix-vector operations. And the level 3 routines build upon the level 2 routines by introducing matrix-matrix operations. For this thesis, only the level 1 routines are used.

## 1.3 Research

The goal of this bachelor's thesis is to put the promise of high performance of SAC to the test. Implementing the BLAS routines in SAC provides a good benchmark to compare performance to hand-optimised low-level code. Also, since SAC is an array programming language and the BLAS routines operate on arrays, it is only fitting to implement the BLAS routines in SAC.

The performance is analysed by comparing several different variables to an optimised BLAS library. These variables are: The different routines themselves, the size of the input vector(s), multi-threading, different values for the increment parameters and different compiler optimisations. These are all explained in greater detail in chapter 3 and the increment parameter in particular is explained in section 2.3.

The document is structured as follows:

- In chapter 2, the implementation of the fourteen level 1 BLAS routines is explained.
- In chapter 3, the way in which the performance is measured is explained.
- In chapter 4, the results from the performance analysis are explained.
- In chapter 5, the conclusions drawn from the results and future work are discussed.

## Chapter 2

# Implementation

This SAC implementation is based on the Fortran reference implementation. For routines of which the implementation is not too long, the code is added to the subsection of the respective routine. The full code for this implementation can be found at the following repository:

<https://github.com/SacBase/BLAS/blob/master/src/blas/BlasLevel1.sac>

The code for the original reference implementation can be found at:

[https://netlib.org/blas/#\\_level\\_1](https://netlib.org/blas/#_level_1)

### 2.1 Types

Since there are minimal differences between the four different data types (`single`, `double`, `complex`, `double complex`) used in BLAS, only one datatype has been implemented for the SAC implementation. At first this was the `single` variant, but at a later stage this was changed to the `double` variant, because the added precision improves the performance measurements.

Every routine, aside from `drotg` and `drotmg`, has at least one vector parameter. In the SAC implementation these are represented by the `double[.]` type, where the `[.]` specifically denotes a one-dimensional array.

Another feature of some of the routines is that they return multiple pieces of data as output. In the reference implementation, this is accomplished by modifying the data via its corresponding pointer, because all data in Fortran is *call by reference*. However, since SAC is a functional language and all data is *call by value*, this approach cannot be taken. Instead, SAC supports returning tuples of data, which is how this is replicated in the SAC implementation.



## 2.2 Length Parameter

Every routine with a vector parameter also has a length parameter (called  $N$ ) to determine the number of elements that should be operated upon. For *lower-level* programming languages (such as C and Fortran) this parameter is useful because arrays in those languages generally do not store their own length. In SAC however, this use of the length parameter is not necessary. The length parameter also allows for a smaller number of elements to be specified, in which case the routine will stop pre-maturely before every element in the input vector has been operated upon.

In terms of expected values for the length parameter; It has to be smaller than the number of elements in the smallest input vector (since many routines take two vectors as input) and greater than zero.

## 2.3 Increment Parameter

Every routine has an additional increment parameter (called `incx` / `incy`) for each one of their vector parameters. These determine the increment size between elements of the vector. For example, by setting an increment parameter to three, only every third element of the corresponding vector will be accessed. This also means that the routine will iterate through the vector at three times the speed that it normally would, so to compensate, the length parameter must be adjusted to be at most a third of the total elements of this vector.

Another feature of this parameter is that by setting an increment parameter to a negative value, the vector will be accessed in reverse order. However, due to increased complexity, this behaviour has not been replicated in the SAC implementation.

Lastly, in the reference implementation, when an increment parameter is set to zero, behaviour depends on the routine, but the majority will repeatedly access the first element of the vector. Due to the fact that this behaviour appears inconsistent between routines, it has also not been replicated in the SAC implementation. For certain routines this behaviour coincidentally happens to be the same as in the reference implementation, which is why in the code for the SAC implementation, for some routines it is specified through a comment that one of the increment parameters is allowed to be zero.

## 2.4 Loops

In the reference implementation there are many instances where iterative loops are being used. In SAC however, for-loops cannot be parallelised. Therefore, tensor comprehensions[5] are used for every routine which operates on vectors (except idamax, which is explained in subsection 2.7.14). In SAC, tensor comprehensions are syntactic sugar for with-loops. They are much more concise than with-loops (syntax-wise they are more similar to list comprehension) and are therefore preferred over with-loops.

In order to explain how tensor comprehensions work, see the following example:

```
1 { iv -> 2*X[iv] | [0] <= iv < [n] step [2] ; iv -> X[iv] }
```

Starting from the left, an *index vector* is defined. This index vector can either be defined with a specific dimensionality (by for example defining it as [i] or [i,j] for a one- or two-dimensional array respectively), or it can be left as a single variable, like it is in this example. In this case the SAC compiler will try to infer the dimensionality of the index vector based on its usage in the tensor comprehension. In this example, the index vector is one-dimensional.

After the arrow comes the *generator-expression*. This expression defines what the output values of the tensor comprehensions will become. In this example, those output values are defined as the values of some vector **X** multiplied by two.

After the vertical bar comes the *generator-range*. This range determines which values get changed according to the generator-expression. In this example, the tensor comprehension ranges from zero to some value **n**. Additionally, by defining a *step*, values can be skipped over. In this example, by setting it to two, every second value will be skipped over. So only one in every two values of **X**, up to **n**, will be multiplied by two.

Values which fall outside of the defined range get passed on to the next generator-expression. For tensor comprehensions any number of generator-expressions can be defined, but since this example is meant to illustrate the way tensor comprehensions have been used for this BLAS implementation, only a single additional generator-expression is defined. This secondary generator-expression also does not have a corresponding generator-range defined, which means that the default range of every element of **X** will be used. This way, this second generator-expression acts as a sort of fallback and all remaining values will simply be left as their corresponding values in **X**.

## 2.5 Correctness

In order to ensure that the output of the routines in the SAC implementation is correct, output of the SAC routines, given various different inputs, has been tested and compared against output from the reference implementation. Aside from slight accuracy differences on input vectors with sizes larger than around  $10^7$  elements, the SAC implementation appears to perform correctly.

## 2.6 Optimisation

Since one of the goals of SAC is to offer high productivity, most of the optimisation work is handled automatically by the compiler. Only select few optimisations have been added by hand, which will be pointed out in the subsections of the respective routines that these optimisations are used in.

## 2.7 Routines

### 2.7.1 drotg

This routine performs preparatory work for a *Givens rotation*. A Givens rotation is a rotation in the plane spanned by two coordinates axes.

The SAC implementation:

```
1 double, double, double, double drotg(double a, double b) {
2   if(b == 0d) {
3     r = a;
4     z = 0d;
5     c = 1d;
6     s = 0d;
7   } else if(a == 0d) {
8     r = b;
9     z = 1d;
10    c = 0d;
11    s = 1d;
12  } else {
13    sigma = tod(abs(a) > abs(b) ? sign(a) : sign(b));
14    r = sigma * sqrt(a*a + b*b);
15    c = a/r;
16    s = b/r;
17    z = abs(a) > abs(b) ? s : c == 0d ? 1d : 1d/c;
18  }
19  return (r, z, c, s);
20 }
```

This routine is one of two routines, along with drotmg, that do not operate on any vectors. For this reason, optimisation is not a concern. Instead the focus is put into making the implementation as readable as possible.

### 2.7.2 drotmg

This routine also performs preparatory work, just like drotg, but this time for a *modified* Givens rotation. The main difference is that the input and output parameters are different.

The SAC implementation for this routine is too long to include here and can be found on the aforementioned repository.

This is the other routine which does not operate on any vectors. Likewise, optimisation is also not a concern for this routine. However, in the reference implementation, this routine is by far the longest out of all level 1 routines. Because of this, it turned out to be too difficult to significantly simplify it for the SAC implementation.

One aspect that is simplified from the reference implementation however is the use of control flow. In the reference implementation, many different control flow constructions are used which cannot be replicated in SAC (for example, early return statements out of if statements). Because SAC is a purely functional language, there is actually no control flow[1]. The language only allows for variables to be conditionally bound within if-else and while statements.

Lastly, this routine is one of two routines that use the `param` parameter. This routine sets up the parameter and returns it, and `drotm` takes it as a parameter. The parameter is explained in more detail in the section on `drotm`.

### 2.7.3 drot

This routine performs the Givens rotation, using values supplied by `drotg`.

The SAC implementation:

```

1 double[.], double[.] drot(int n, double[.] X, int incx, double
   [.] Y, int incy, double c, double s)
2 {
3   //expects n > 0, incx > 0, incy > 0
4   return (
5     {i -> c*X[i] + s*Y[(i*incy)/incx]
6     | [0] <= i < [n*incx] step [incx]
7     ; i -> X[i]}
8     ,
9     {i -> c*Y[i] - s*X[(i*incx)/incy]
10    | [0] <= i < [n*incy] step [incy]
11    ; i -> Y[i]}
12   );
13 }
```

This routine uses two tensor comprehensions, one for generating the output `X` vector and the other for generating the output `Y` vector. Both tensor comprehensions use the same features explained in the example in section 2.4, with `x` and `y` values being flipped between the two tensor comprehensions.

### 2.7.4 drotm

This routine performs the modified Givens rotation, using values supplied by drotmg.

The SAC implementation:

```
1 double[.], double[.] drotm(int n, double[.] X, int incx, double
   [.] Y, int incy, double[5] param)
2 {
3     //expects n > 0, -2 <= param[0] <= 1
4     flag = param[0];
5     h11 = param[1];
6     h21 = param[2];
7     h12 = param[3];
8     h22 = param[4];
9
10    if(n < 1 || flag == -2d) {
11        X_prime = X;
12        Y_prime = Y;
13    } else if(flag < 0d) {
14        X_prime = {i -> X[i]*h11 + Y[(i*incy)/incx]*h12
15                  | [0] <= i < [n*incx] step [incx]
16                  ; i -> X[i]};
17        Y_prime = {i -> X[(i*incx)/incy]*h21 + Y[i]*h22
18                  | [0] <= i < [n*incy] step [incy]
19                  ; i -> Y[i]};
20    } else if(flag == 0d) {
21        X_prime = {i -> X[i] + Y[(i*incy)/incx]*h12
22                  | [0] <= i < [n*incx] step [incx]
23                  ; i -> X[i]};
24        Y_prime = {i -> X[(i*incx)/incy]*h21 + Y[i]
25                  | [0] <= i < [n*incy] step [incy]
26                  ; i -> Y[i]};
27    } else {
28        X_prime = {i -> X[i]*h11 + Y[(i*incy)/incx]
29                  | [0] <= i < [n*incx] step [incx]
30                  ; i -> X[i]};
31        Y_prime = {i -> -X[(i*incx)/incy] + Y[i]*h22
32                  | [0] <= i < [n*incy] step [incy]
33                  ; i -> Y[i]};
34    }
35
36    return (X_prime, Y_prime);
37 }
```

This is the other routine that uses the **param** parameter. This parameter is a vector consisting of a flag (called **flag**) and four values (called **h11**, **h21**, **h12**, **h22**) which represent a 2x2 matrix. Despite the fact that this parameter is of type **double**, the flag is only ever expected to be either -2, -1, 0 or 1, and depending on the value of the flag, this routine changes which values of the 2x2 matrix are used in the computation.

For this reason the routine has four different cases, three of which have pairs of slightly different tensor comprehensions for both output vectors and a fourth case for when the flag is equal to -2 where vectors X and Y remain unchanged. This is particularly of note because in the reference implementation, the case where the flag is equal to -2 gets checked first and the case where the flag is equal to 1 is left as the else case. This means that if the flag is equal to any value other than -2, -1 or 0, it will be treated as 1. This behaviour has been replicated in the SAC implementation.

### 2.7.5 dswap

This routine swaps values between vectors X and Y.

The SAC implementation:

```

1 double[.], double[.] dswap(int n, double[.] X, int incx, double
   [.] Y, int incy)
2 {
3   //expects n > 0, incx > 0, incy > 0
4   return (
5     {i -> Y[(i/incx)*incy]
6     | [0] <= i < [n*incx] step [incx]
7     ; i -> X[i]}
8     ,
9     {i -> X[(i/incy)*incx]
10    | [0] <= i < [n*incy] step [incy]
11    ; i -> Y[i]}
12   );
13 }
```

This routine is implemented in an identical way to `drot`; with a pair of tensor comprehensions, one for each output vector. The only difference is that this routine does not modify values of the input vectors in the generator-expression. As mentioned in section 2.1, SAC allows for tuples of data to be returned, which is how the *swapping* of X and Y is achieved here. Due to the functional nature of SAC, it would not be possible to actually modify the two vectors, so copies are returned here instead.

### 2.7.6 dscal

This routine performs scalar-vector multiplication ( $x := a * x$ ).

The SAC implementation:

```
1 double[.] dscal(int n, double a, double[.] X, int incx)
2 {
3     //expects n > 0, incx > 0
4     return {i -> a * X[i]
5             | [0] <= i < [n*incx] step [incx]
6             ; i -> X[i]};
7 }
```

This routine is one, along with dnorm2, dznrm2, dasum and idamax, that operates on a single vector. As a result the tensor comprehension used here is simpler. It only multiplies the length parameter by the increment parameter to get the right index of the input vector, then sets the step size equal to the increment parameter.

### 2.7.7 dcopy

This routine copies values from vector X into vector Y.

The SAC implementation:

```
1 double[.] dcopy(int n, double[.] X, int incx, double[.] Y, int
    incy)
2 {
3     //expects n > 0, incx >= 0, incy > 0
4     return {i -> X[(i/incy)*incx]
5             | [0] <= i < [n*incy] step [incy]
6             ; i -> Y[i]};
7 }
```

Since SAC is a functional language, vector Y is not actually modified and instead a copy of the vector is returned by the routine.



### 2.7.8 daxpy

This routine performs scalar-vector multiplication, as well as vector-vector addition ( $y := a * x + y$ ).

The SAC implementation:

```
1 double[,] daxpy(int n, double a, double[,] X, int incx, double
  [,] Y, int incy)
2 {
3   //expects n > 0, incx >= 0, incy > 0
4   if(a == 0d) {
5     result = Y;
6   } else {
7     result = {i -> a * X[(i/incy)*incx] + Y[i]
8               | [0] <= i < [n*incy] step [incy]
9               ; i -> Y[i]};
10  }
11  return result;
12 }
```

The tensor comprehension used in this routine works exactly as explained in section 2.4. The only other detail about this implementation that is noteworthy is that a special case for when  $a = 0$  was added, because in this case the Y vector can be returned as is, since  $y := a * x + y = 0 * x + y = y$ .

### 2.7.9 ddot

This routine computes the dot product of its two vectors.

The SAC implementation:

```
1 double ddot(int n, double[,] X, int incx, double[,] Y, int incy)
2 {
3   //expects n > 0, incx >= 0, incy >= 0
4   return sum( {i -> X[i*incx] * Y[i*incy] | [0] <= i < [n]} );
5 }
```

The tensor comprehension in this routine does not need to change the step size, because it does not iterate over either of its input vectors, instead it generates an entirely new vector composed of values of both input vectors.

### 2.7.10 dsdot

This routine computes the dot product, just like `ddot`, but with *extended precision accumulation*.

The SAC implementation:

```
1 double dsdot(int n, float[.] X, int incx, float[.] Y, int incy)
2 {
3     //expects n > 0, incx >= 0, incy >= 0
4     return sum(
5         {i -> tod(X[i*incx]) * tod(Y[i*incy]) | [0] <= i < [n]}
6     );
7 }
```

This routine is identical to `ddot`, except that it takes two `float` vectors. The values from these vectors are converted to `double` values, then multiplied together. This way, if there is a large difference between the orders of magnitude of both elements, their product will produce a more accurate result, due to the upgrade from `float` to `double` values.

### 2.7.11 dnrm2

This routine computes the *Euclidean norm*. The Euclidean norm is the inner product of two vectors in Euclidean space.

The SAC implementation:

```
1 double dnrm2(int n, double[.] X, int incx)
2 {
3     //expects n > 0, incx > 0
4     return sqrt(sum(
5         {i -> X[i]*X[i] | [0] <= i < [n*incx] step [incx]}
6     ));
7 }
```

This routine, just like `dscal`, only operates on a single vector. However, unlike `dscal`, the tensor comprehension in this routine does not specify a secondary generator-expression, because without any other generator-expression to fall back on, the output values will be set to zero, which is actually what desirable here. The output of the tensor comprehension gets summed afterwards, so zero values will not change the result in any way.

### 2.7.12 dznrm2

This routine also computes the Euclidean norm, but for vectors of complex numbers.

The SAC implementation:

```
1 double dznrm2(int n, complex[,] X, int incx)
2 {
3     //expects n > 0, incx > 0
4     X_prime = {i -> X[i*incx]
5                 | [0] <= i < [n]
6                 ; i -> zero()
7                 | i < [n*incx]};
8     return sqrt(sum(normSq(X_prime)));
9 }
```

This routine works similar to `dnrm2` in theory, but in the actual implementation there are many differences. Firstly, this routine uses a vector of `complex` values. The `complex` datatype in SAC is defined as `double[2]`, which means an array with two values representing the real and the imaginary components of the complex number. In the reference implementation and in Fortran in general, complex values are defined in the same way.

Secondly, to correctly get the right elements from the input vector given the increment parameter, an extra step is needed for this routine. This is a tensor comprehension which generates complex zero values as a fallback. It does this through the standard library function `zero`, which simply returns an array with two values which are both zero. It does not matter that these values are not removed altogether, because they all get summed afterwards, so the zero values do not change the results, just like in `dnrm2`.

Lastly, SAC also has a function in its standard library for computing the squared norm of an array of complex numbers, called `normSq`, which was used in this implementation.

### 2.7.13 dasum

This routine computes the sum of absolute value of the given vector.

The SAC implementation:

```
1 double dasum(int n, double[,] X, int incx)
2 {
3     //expects n > 0, incx > 0
4     return sum(
5         {i -> abs(X[i]) | [0] <= i < [n*incx] step [incx]}
6     );
7 }
```

This routine simply converts every element (taking the length and increment parameters into account) to its absolute value, then sums every element together. For this reason a secondary generator-expression was not required, as the default value of zero can be used here.

### 2.7.14 idamax

This routine finds the first index of the largest absolute value.

The SAC implementation:

```
1 int idamax(int n, double[,] X, int incx)
2 {
3     //expects n > 0, incx > 0
4     max_i = 0;
5     max_v = 0d;
6     ix = 0;
7     for(i = 0; i < n; i++) {
8         current_v = abs(X[ix]);
9         if(current_v > max_v) {
10             max_i = i;
11             max_v = current_v;
12         }
13         ix += incx;
14     }
15
16     return max_i;
17 }
```

This routine is the only one which uses a for-loop in its implementation instead of a tensor comprehension. This was done because tensor comprehensions compute their output values independently of each other, which makes it difficult to know what the largest value is going to be. SAC does have a `max` function in its standard library, but this function returns the actual value instead of the first index, so it was not used in this routine.

## Chapter 3

# Performance Analysis

As mentioned in the introduction, several different variables have been investigated for the performance analysis:

1. Twelve out of the fourteen routines; Since `drotg` and `drotmg` do not operate on vectors, their performance is in a different class from the other routines and were therefore left out of the performance analysis.
2. Input size; The number of elements in the input vector(s).
3. Multi-threading; The number of threads that the program runs on.
4. Different values for the increment parameters.
5. Different compiler optimisations.

The system specifications of the machine on which the performance is measured can be found in appendix A.

### 3.1 Performance Quantification

To quantify the performance of the SAC implementation, the time taken to run the routines is measured and from this data, two different results are calculated: The number of floating point operations per second (flop/s) and the number of reads and writes per second (memory bandwidth). These results are calculated from formulas which are manually defined based on the implementation of each routine. For example, in the implementation of `ddot`:

```
1 sum( {i -> X[i*incx] * Y[i*incy] | [0] <= i < [n]} )
```

Here, the tensor comprehension will read a value from `X` and a value from `Y`, then perform one floating point operation, then write the result to memory, for all  $n$  elements of the vectors. This results in  $2*n + n$  reads and writes and  $n$  flops. After the tensor comprehension is done, the results are summed, which adds another  $n$  flops.

As a result of this method however, these calculated values do not take potential compiler optimisations into account. For instance in the above `ddot` example, it is assumed that the `sum` function will only be evaluated after the tensor comprehension is finished. However, the compiler will most likely combine these two operations into a single one which computes each element of the tensor comprehension and then immediately adds it to the partial sum. So for this thesis potential compiler optimisations are not taken into account.

### 3.2 Optimised BLAS Library

The performance of every individual routine (except `drotg` and `drotmg`) is compared against the performance of the same routine ran from an optimised BLAS library. Because the test system uses an AMD processor, AMD's optimised BLAS library, which is part of AOCL (AMD Optimizing CPU Libraries), has been selected[6]. AOCL actually implements the BLIS framework, but BLIS (BLAS-like Library Instantiation Software) is a superset of BLAS, so it implements every routine, uses the same interface and is therefore fully compatible with BLAS.

The library comes with precompiled routines for use in both Fortran and C programs. In order to make the comparison as fair as possible, the precompiled C-code is called from within the same SAC program which also calls the native SAC routines. This is possible because SAC programs first get compiled to C-code, which allows external C-code to be called from within a SAC program.

### 3.3 Obtaining Performance Data

In order to obtain the performance data, both native SAC and AOCL routines are run from the same SAC program. Unless stated otherwise this program is always compiled with the GCC compiler flag `-ffast-math`. This flag cuts certain corners to speed the program up. It has been enabled because the optimised BLAS library likely cuts similar corners, so enabling this flag should make comparisons fairer. The flag is explained further in section 4.4. In a single run of the program, every data point for a given routine is sampled 100 times to get an average in order to reduce variability in the results.

Another measure that is taken to reduce variability is that CPU frequency scaling has been disabled. CPU frequency scaling is a feature present in most modern CPU's which allows them to dynamically alter the clock frequency of their cores. This does however cause the performance of the routines to fluctuate in tandem with the clock speeds of the corresponding core(s), which is why it has been disabled. Despite these measures the results still show some variance. This is likely due to interference from other tasks that are running in the background. If this is indeed the case then there is little that could be done to reduce the variability further. The graphs show the variance at each data point by means of a vertical bar (which indicates the standard deviation from all 100 samples of that data point). The graphs are explained at the start of the next chapter.

Another aspect worth mentioning is that during the measuring process, the first time a routine is called, execution is slower than on subsequent calls. This happens because during the first call, the input parameters have not yet been loaded into the CPU's cache and they need to be loaded from RAM instead, which is referred to as *cold* cache. Then on every subsequent call to the routine, the input parameters have already been loaded into cache, which is referred to as *warm* cache. In this case the difference in performance between the first and subsequent runs is about a factor of ten. However, considering that every data point is sampled 100 times, this performance difference of the first run is deemed insignificant and has not been filtered out of the results (also, both native SAC and AOCL routines are measured the same way, so this does not give either version an advantage).

Lastly, for testing the multi-threading, increment parameters and compiler optimisations, the SAC program runs multiple times with different compiler settings to gather all data. This most likely does not impact the results in any way, but it should be mentioned just in case.

## Chapter 4

# Results

In order to accurately analyse the input size and multi-threading variables, both variables needed their own type of graph. The remaining two variables, increment parameters and compiler optimisations, did not need their own type of graph, since these variables are binary. These two variables reuse the graphs for input size. All graphs have been generated using Python's Matplotlib library.

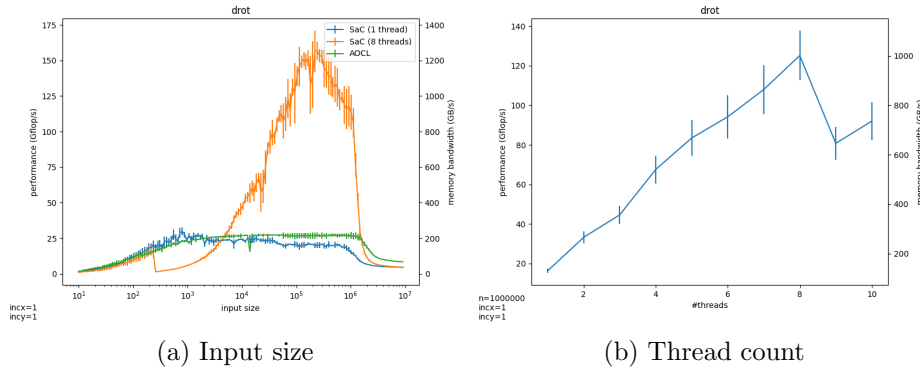


Figure 4.1: The two types of graph

In figure 4.1 both types of graph are shown. Both types have their respective variable plotted along the x-axis. The input size graphs have three lines; a blue line for SAC in single-threaded mode, a green line for the optimised BLAS library (AOCL) and an orange line for SAC in multi-threaded mode (with eight threads). In the thread count graphs only a single line is shown, which represents the SAC version. Both types of graphs have both the flop/s and memory bandwidth plotted along the y-axis. The dswap and dcopy routines only perform read/write operations, so for those graphs the axis with flop/s has been left out. All graphs can be found in appendix B.



## 4.1 Input Size

For half of the tested routines, the performance of the SAC implementation is on par with the AOCL implementation, as is the case in figure 4.2. Only with `ddot`, `dnrn2`, `dznrm2` and `idamax` did the AOCL implementation significantly outperform the SAC implementation.

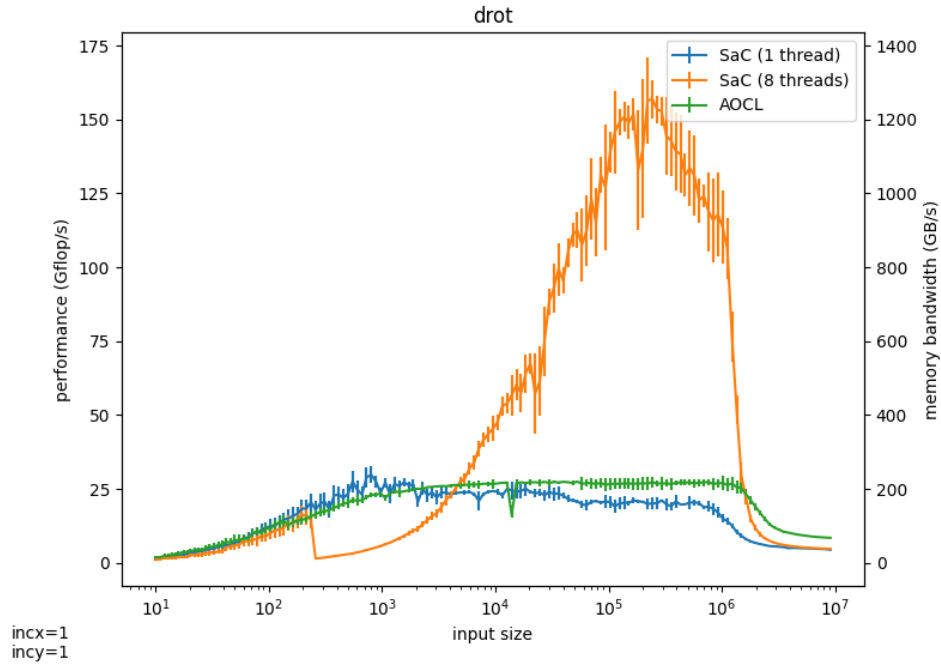


Figure 4.2: Example input size graph (drot)

For `idamax`, this performance difference is likely caused by the use of a for-loop in the SAC implementation. For the other three routines it is not entirely known why they perform so much worse. In the case of `dznrm2` it might be related to the use of the `complex` datatype, as this is the only routine which uses it. One feature that all three routines have in common is that they call `sum` on the output of a tensor comprehension, but those are not the only three routines that do this.

On the other hand, the SAC versions of `dsdot` and `dasum` actually outperformed their AOCL counterparts. This is an unexpected outcome, since both of these routines also call `sum` on the output of a tensor comprehension. Although it is also worth noting that for these two routines their performance in the AOCL version is relatively worse than the performance of the other AOCL routines. It is possible that these routines have not been optimised as well as the other routines. However, since I do not have access to the source code of the AOCL library, it is difficult to say for sure and these routines will have to be considered *optimised* and the results have to be taken as is.

Another detail that can be found in the results of every routine and happens for both SAC and AOCL implementations is that past  $2 * 10^6$  elements, the performance drops considerably. This is most probably caused by the fact that past this point, the input vectors no longer fit in L3 cache in their entirety. The test system has 32 megabytes of L3 cache. Two vectors of  $2 * 10^6$  elements each, each element taking up 8 bytes in memory (because the `double` datatype takes up 8 bytes), would lead to 32 megabytes. This calculation shows that this would be the limit on the size of the input vectors before they no longer fit in L3 cache and thus explains the performance drop in the graphs. Additionally, past this point the performance drops to about 50GB/s in terms of memory bandwidth, which is about the same as the bandwidth of the RAM on the test system. This also confirms that past this point the input data no longer fits in L3 cache and is loaded from RAM instead.

## 4.2 Multi-threading

Firstly, from the input size graphs (figure 4.2 and appendix B.1) it becomes clear that the performance of the SAC version in multi-threaded mode only gets better as the input size increases (until it no longer fits in the CPU's cache). This is because multi-threading always introduces overhead, since dividing up the work and scheduling also takes time.

Also observable in the input size graphs is a drop in performance of the multi-threaded version specifically at 200 elements. This is caused by a SAC compiler flag, called `-minmtnsize`. This compiler flag determines when operations on arrays should actually start being multi-threaded. This is done because the overhead of multi-threading makes it not worth using on smaller sized arrays, as can be seen in the graphs right after it is enabled, where the performance drops before gradually climbing back up again.

In regards to the thread count graphs; Initially it was intended for the multi-threaded SAC performance to be compared against the multi-threaded AOCL performance. However, after *much* trial and error I was unable to get the multi-threaded AOCL library working as intended on my machine. After installing the library and running the routines, they showed identical performance to the single-threaded version and upon closer inspection, also only used one CPU core. For this reason this comparison has been left out of the thesis and SAC's multi-threaded performance is only compared against its single-threaded performance, as can be seen in figure 4.3 and in appendix B.2. This does however still lead to some interesting findings.

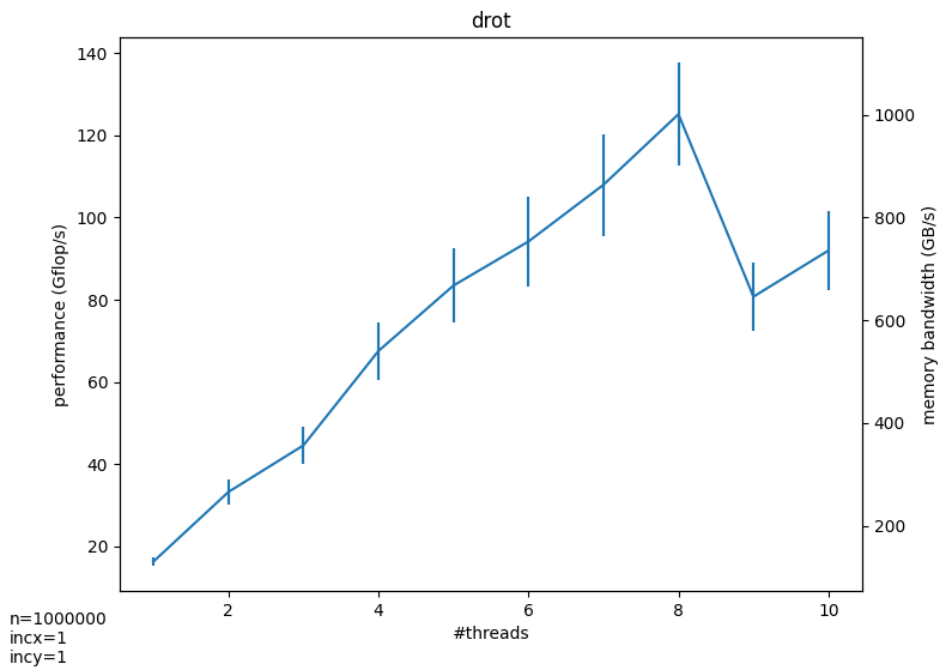


Figure 4.3: Example thread count graph (drot)

All routines showed results similar results to the example shown in figure 4.3. The only exception to this is idamax, which performs entirely differently, due to the use of a for-loop, which do not get parallelised by the SAC compiler. For all other routines performance generally scales linearly with the number of threads. Also noteworthy is that as the number of threads increase, so does the variability between the different samples.

The routines `ddot`, `dsdot`, `dnrm2`, `dasum` and `idamax` all have a performance drop going from single-threaded to multi-threaded with two threads. For `idamax` this is to be expected, since the implementation uses a for-loop. For the other four routines however, it is unknown why this happens. They do all have in common that they call `sum` on a tensor comprehension, but it is not clear why this would cause a drop in performance here.

For every routine (except `idamax`), at eight threads, the performance either does not increase as fast as it did before, or it even drops. This is due to the fact that the test system only has eight physical cores. After that, going from nine to ten threads, the performance increases once again, for every routine (including `idamax`). It is not quite known why this happens. For `idamax` this behaviour is even less explainable, because it happens despite the fact that this routine is not parallelised. For the other routines this behaviour past eight threads could be explained by hyper-threading.

### 4.3 Increment Parameters

Based on testing it turns out that these parameters only impact performance when changed from one to a value higher than one (based on testing there is no significant difference when going from two to three or from three to four). As explained in section 2.3, this SAC implementation does not support increment parameters to be set to zero or lower, so this has not been tested.

When an increment parameter is set to a value higher than one, it causes performance to universally get worse and this goes for both the SAC and AOCL versions, as can be in figure 4.4 and in appendix B.3). The only exception to this is AOCL dasum, which performs the same. This performance loss could be explained by the fact that when an increment parameter is higher than one, instead of accessing the contiguous array in sequence, values are skipped. It might be slower to access the array this way, instead of accessing contiguous sequences of values at once.

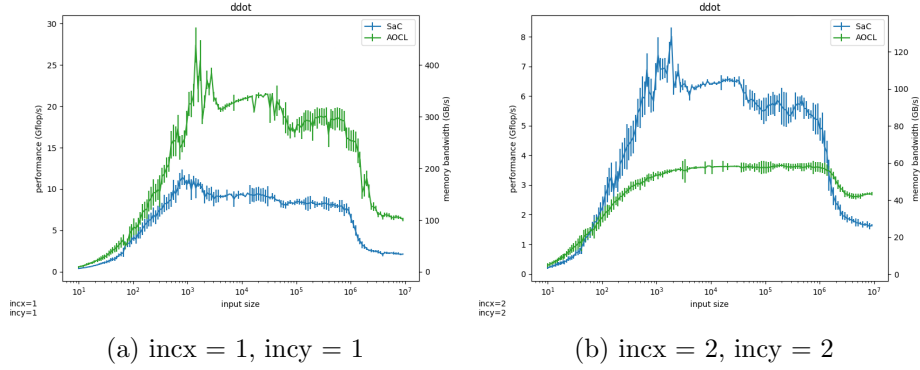


Figure 4.4: Example increment parameter comparison (`ddot`)

## 4.4 Compiler Optimisations

Initially, it was planned for multiple compiler flags to be tested, but in the end only `-ffast-math` has been tested. As mentioned before, this flag cuts certain corners in order to speed up the program. These *cut corners* include breaking compliance with the IEEE specification for floating point numbers and making certain assumptions which allows the compiler to skip many safety checks[7].

This flag only impacts the performance of `ddot`, `dsdot`, `dnrm2` and `dasum`. In all four cases it greatly improves the performance, as can be seen in figure 4.5 and in appendix B.4. It is unknown exactly why this is, but it might be related to the fact that all four of these routines call `sum` on a tensor comprehension (a common pattern that has occurred a few times before already).

Additionally, this compiler optimisation also only appears to work in single-threaded mode. One detail about the input size graphs that has not yet been discussed is that in the graphs for `ddot`, `dsdot`, `dnrm2` and `dasum` (these can be found in appendix B.1), the performance of the SAC version in multi-threaded mode at the start of the graph is actually worse than in single-threaded mode. As explained in section 4.2, for vector sizes smaller than 200 elements, the SAC compiler will (by default) not run the operations multi-threaded. This means that the performance here in multi-threaded mode is worse than in single-threaded mode, despite the fact that in multi-threaded mode the array operations are computed single-threaded. This anomaly can be explained by the fact that the `-ffast-math` compiler flag does not work in multi-threaded mode. It is unknown why this is the case.

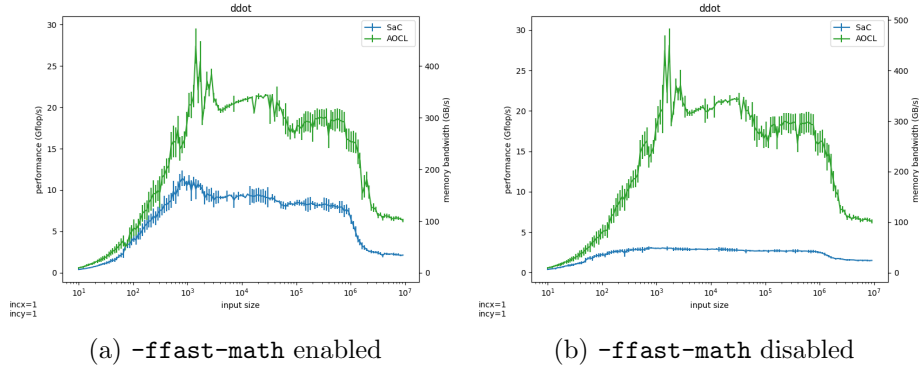


Figure 4.5: Example `-ffast-math` comparison (`ddot`)

## Chapter 5

# Conclusions

In terms of the three core design goals of SAC:

- High Performance: The SAC implementation performed comparable to the optimised BLAS library in half of the tested routines. two routines even outperformed this *hand-optimised* library.
- High Productivity: The SAC implementation has not been hand-optimised at all. Nor were there any other low-level implementation details to be concerned with.
- High Portability: The SAC implementation runs on single- & multi-threaded CPU's, GPU's, clusters and more, while the reference implementation will only work on single-threaded CPU's and the optimised BLAS library will work only on supported AMD CPU architectures (all with a single source code).

In conclusion, in regards to the level 1 BLAS routines, SAC definitely delivers on the promise of *high performance*.

As for future work, first and foremost would be further testing. Comparing more compiler optimisations and comparing multi-threaded SAC to a multi-threaded BLAS library have already been mentioned.

Another comparison that could be made is comparing the performance of the SAC implementation with the performance of the Fortran reference implementation. This was not done for this thesis, because the current performance measurements of both the SAC and optimised library versions are taken from within the same SAC file. This cannot be done for measurements of the reference implementation, because SAC cannot call native Fortran code. Therefore, it would require measurements to be taken from a separate Fortran program. And for these measurements it would be much more difficult to say for sure whether or not they can be directly compared

to the measurements taken from the SAC program (in terms of what kind of time is measured among other aspects).

Another possible aspect to analyse is chaining multiple routines together. Here SAC should start to outperform any optimised library, since the library can only call optimised individual routines, whereas the SAC compiler might be able to figure out more advanced optimisations when dealing with multiple routines. Some experimentation with this was done over the course of this thesis, but it was concluded that this would introduce too many variables (both in terms of number of chains and combinations of different routines) to also cover here.

For figuring out the details in the performance analysis that were left unanswered in this thesis, it would be beneficial to investigate the assembly code generated by the SAC compiler. This was also considered for this thesis, but was ultimately also left out.

Aside from further testing, some routines of this implementation can likely be optimised further. The implementation of `drotmg` can most certainly be simplified to have less lines of code, which would improve readability. And it might be possible to use a tensor comprehension in the implementation of `idamax`, which would make it perform much better than it does now.

As mentioned in section 2.3, this SAC implementation does not allow for the increment parameters to be negative. In the reference implementation negative increment parameters allow the corresponding input arrays to be traversed backwards. To implement this in the SAC version would require an extra bit of code inside every routine that checks if one of the increment parameters is negative and if so, either reverses the order of the corresponding vector, or leads to a second tensor comprehension which traverses the corresponding vector backwards. Either way it would add *many* extra lines of code for functionality that is absolutely non-essential, since one can simply reverse the vector before and after calling the routine. For this reason it has been left out of the implementation in this thesis.

Lastly, the level 2 and level 3 BLAS routines could be implemented next. Due to the fact that these routines operate on matrices, which, unlike vectors, are very difficult to optimise, it was decided that these were beyond the scope of this bachelor's thesis.



# Bibliography

- [1] Scholz, S.B., Herhut, S., Penczek, F., Grelck, C., Šinkarov, A. & Viessmann, H.N. *Single Assignment C Tutorial*, March 2022. Version 1.2.1, Retrieved from [https://www.sac-home.org/\\_media/docs/tutorial.pdf](https://www.sac-home.org/_media/docs/tutorial.pdf).
- [2] Grelck, C. Single Assignment C (SAC): High Productivity Meets High Performance. In V. Zsóík, Z. Horváth, and R. Plasmeijer, editors, *4th Central European Functional Programming Summer School (CEFP'11), Budapest, Hungary*, volume 7241 of *Lecture Notes in Computer Science*, pages 207–278. Springer, 2012. Retrieved from [https://www.sac-home.org/\\_media/publications/pdf:2012\\_1.pdf](https://www.sac-home.org/_media/publications/pdf:2012_1.pdf).
- [3] About SaC [SaC-Home]. Accessed April 6, 2023. <https://www.sac-home.org/index>.
- [4] Lawson, C.L., Hanson, R.J., Kincaid, D.R. & Krogh, F.T. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979. Retrieved from <https://dl.acm.org/doi/pdf/10.1145/355841.355847>.
- [5] Scholz, S.B. & Šinkarovs, A. Tensor comprehensions in SaC. In *Proceedings of the 31st Symposium on the Implementation and Application of Functional Programming Languages, IFL '19*, New York, NY, USA, 2019. ACM. Retrieved from [https://www.sac-home.org/\\_media/publications/pdf:tensor-comprehensions-ifl19.pdf](https://www.sac-home.org/_media/publications/pdf:tensor-comprehensions-ifl19.pdf).
- [6] AMD. *AOCL User Guide*, November 2022. Revision 4.0, Retrieved from [https://developer.amd.com/wp-content/resources/57404\\_AOCL\\_v4.0\\_GA\\_UG.pdf](https://developer.amd.com/wp-content/resources/57404_AOCL_v4.0_GA_UG.pdf).
- [7] Free Software Foundation. *Using the GNU Compiler Collection (GCC)*. Retrieved from <https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>.

# Appendix A

## System Specifications

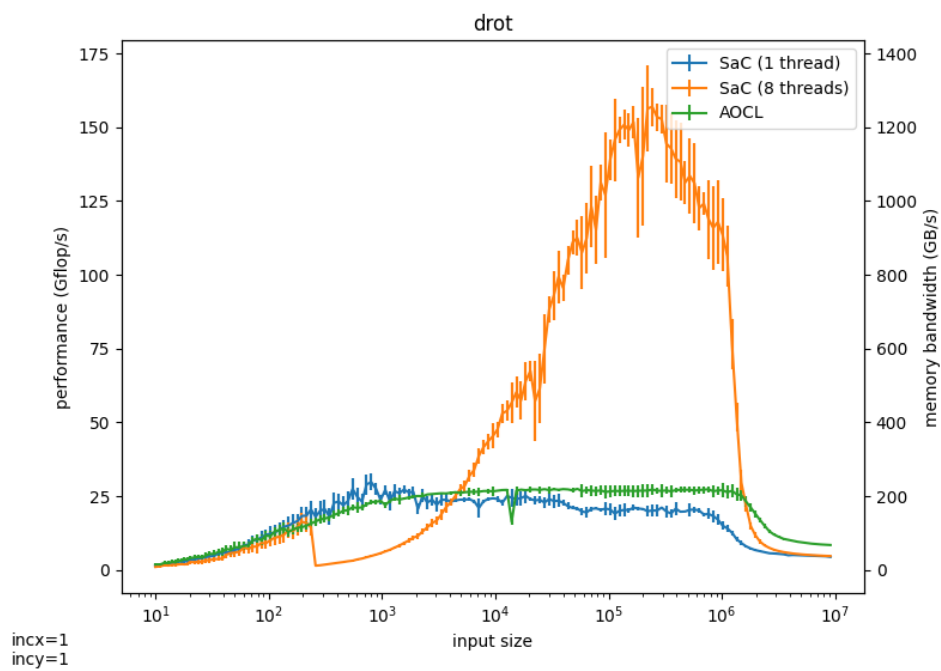
These are the specifications of the test system on which all performance measurements were taken:

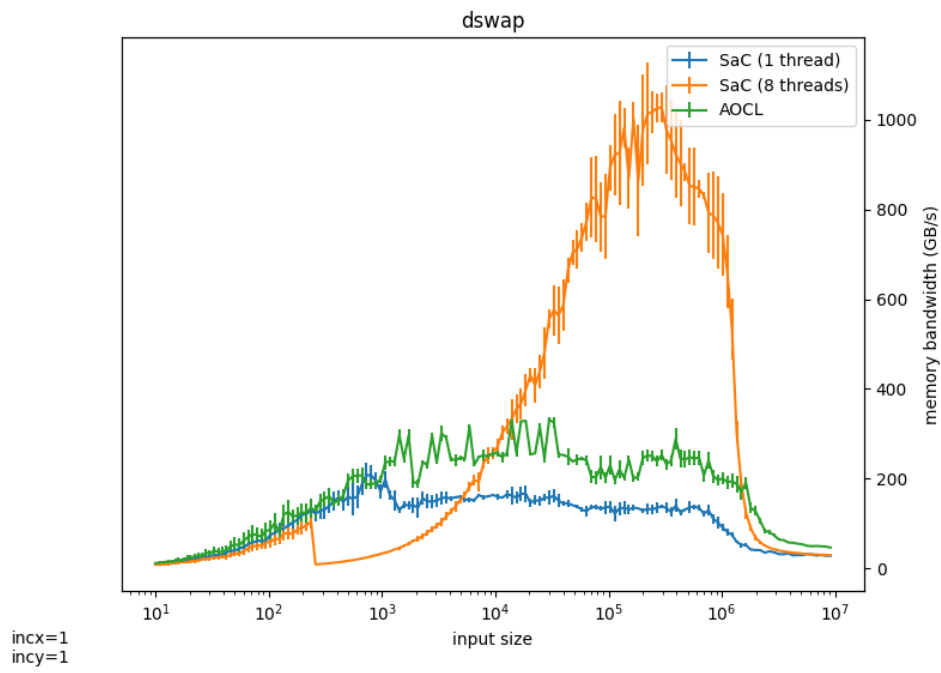
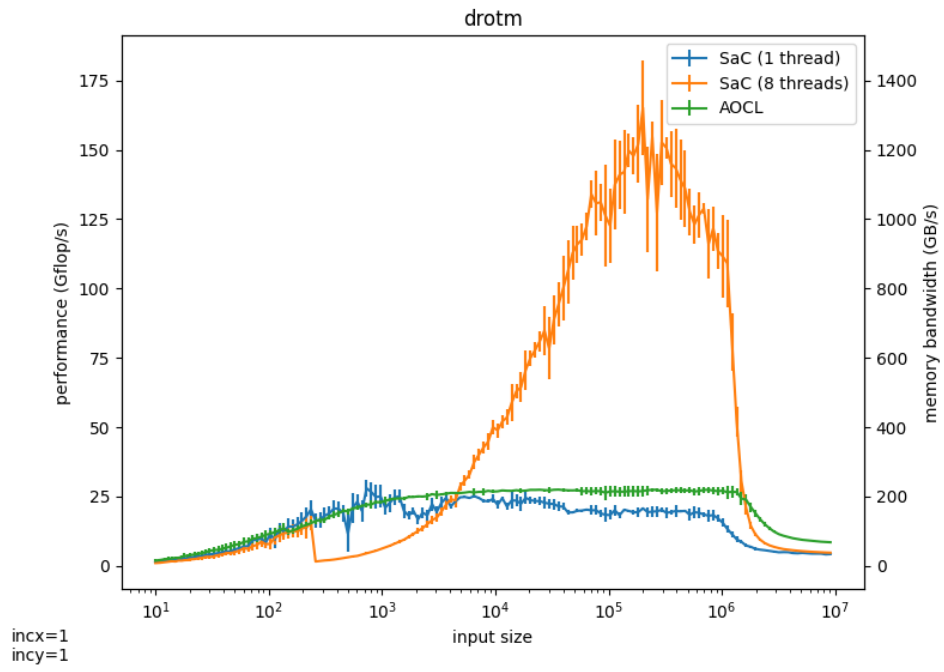
- CPU: AMD Ryzen 7 7700x
  - Zen4 architecture
  - 5.5GHz max clock speed (single core load)
  - 4.5GHz max clock speed (all core load)
  - 8 cores / 16 threads
  - Cache:
    - \* L1: 512KB (32KB per core)
    - \* L2: 8MB (1MB per core)
    - \* L3: 32MB (L3 cache is shared between cores)
  - AVX512 is supported
- Motherboard: Asrock x670e
- DRAM: 16GB DDR5 at 4800MT/s
  - $\sim 50$ GB/s bandwidth
- Operating system: Ubuntu 20.10
- SAC compiler version 1.3.3-MijasCosta-705-g41ed2
- C compiler: GCC 12.2.0
- Optimised BLAS library: AOCL 4.0

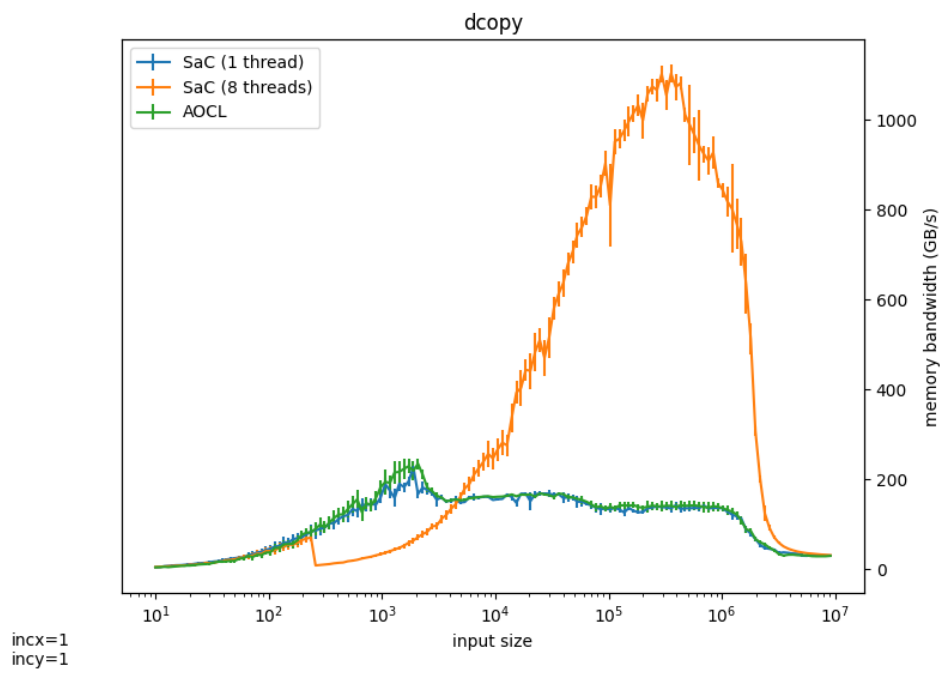
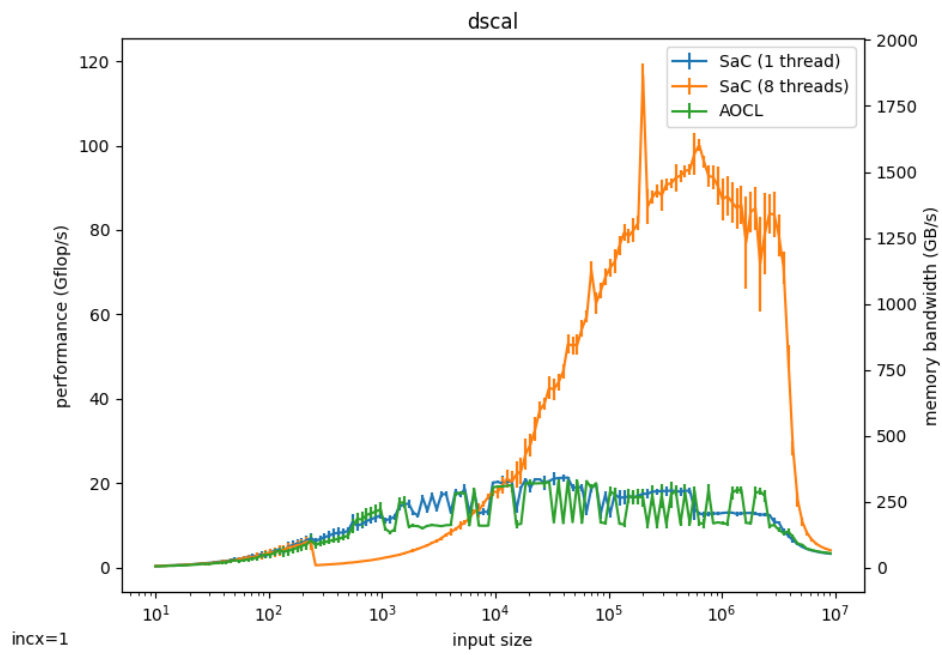
## Appendix B

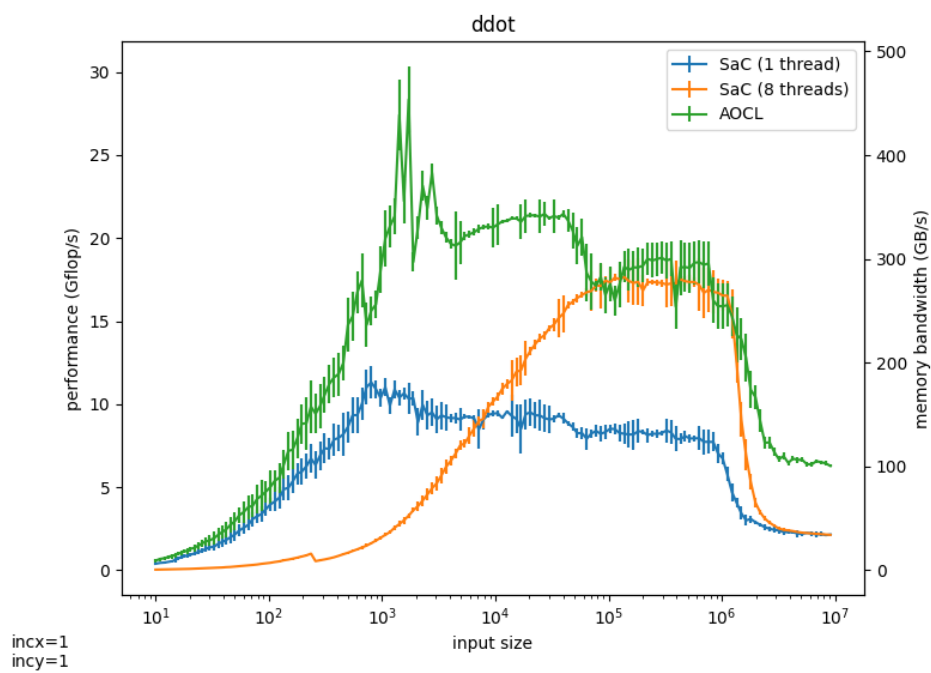
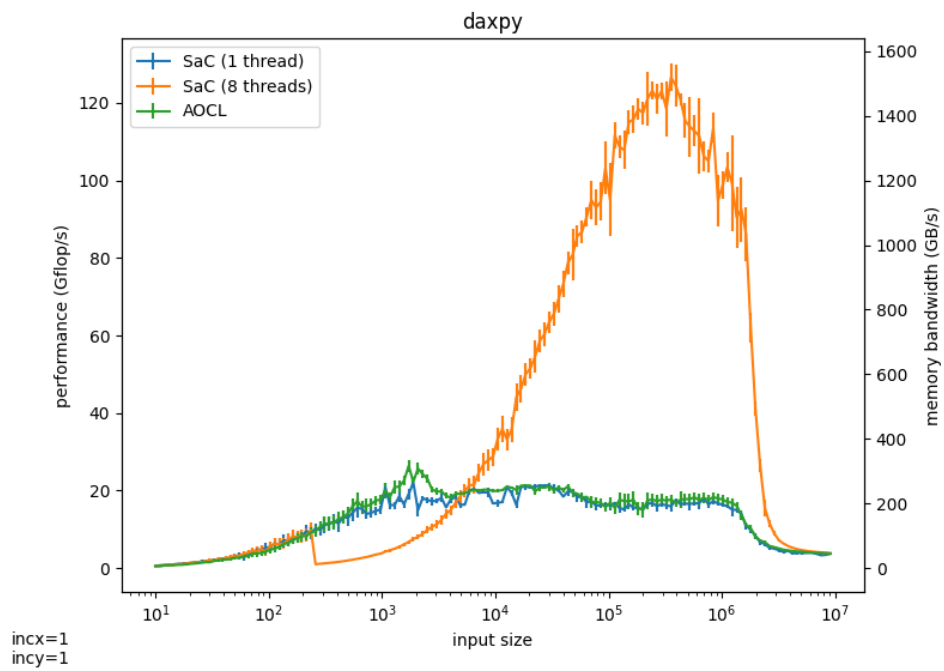
# Graphs

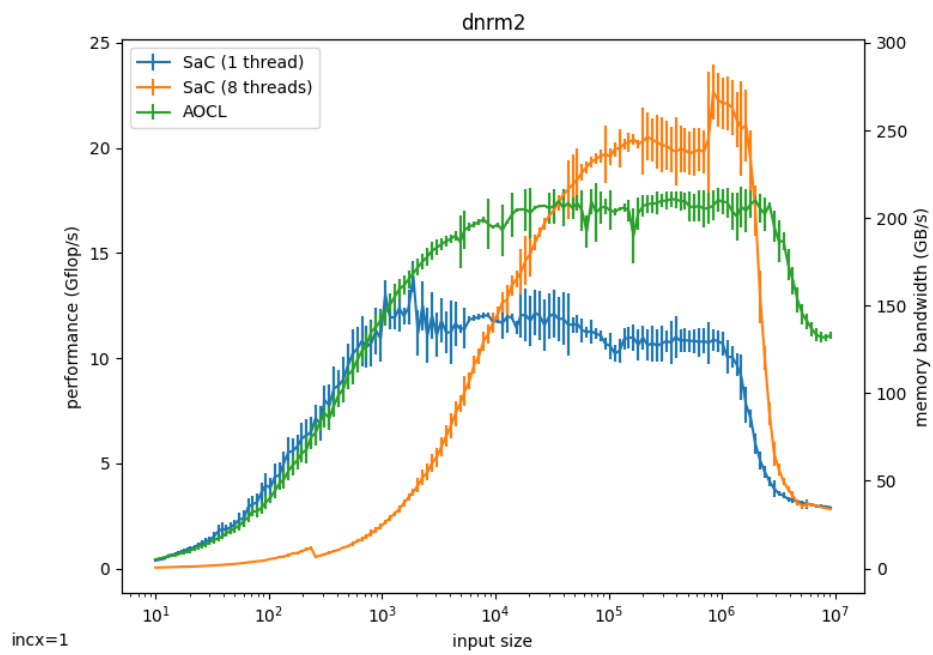
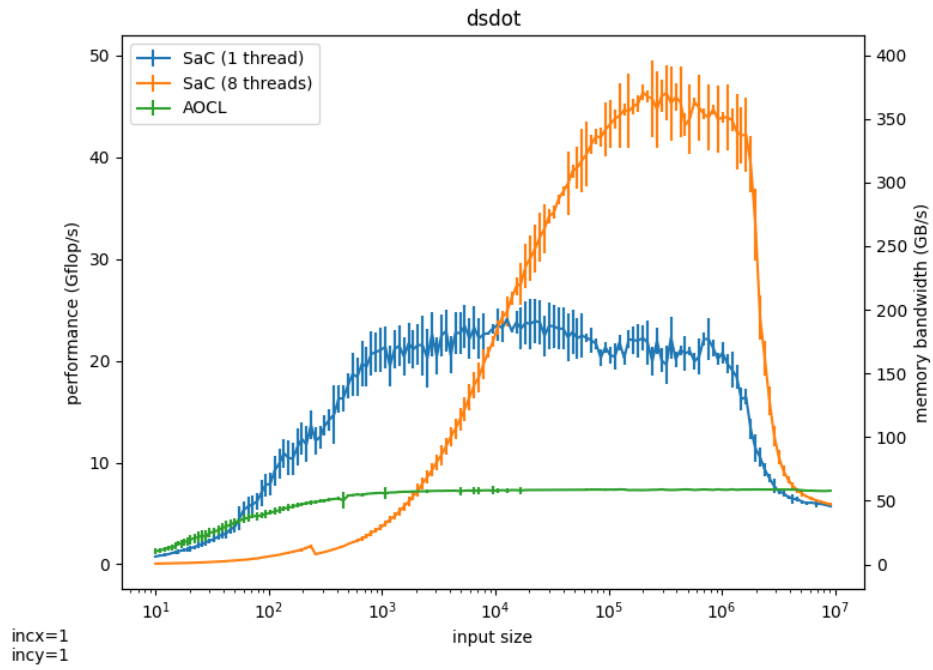
### B.1 Input Size

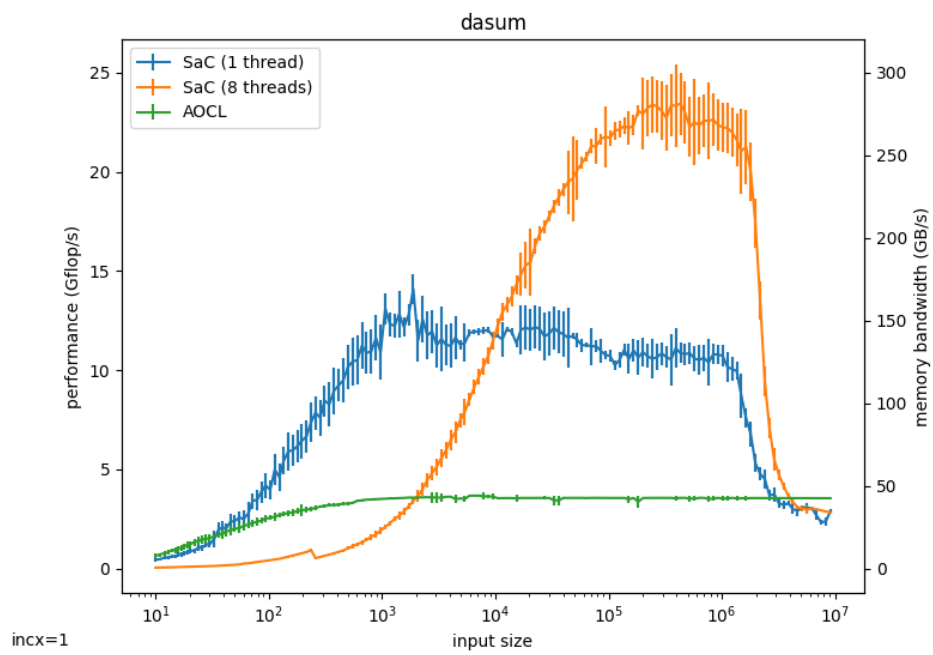
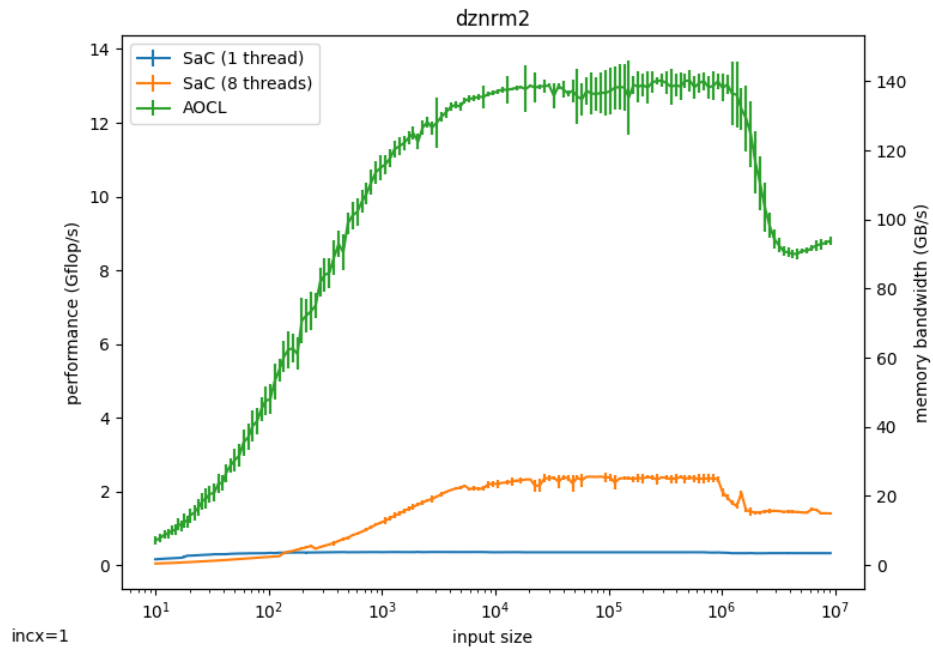




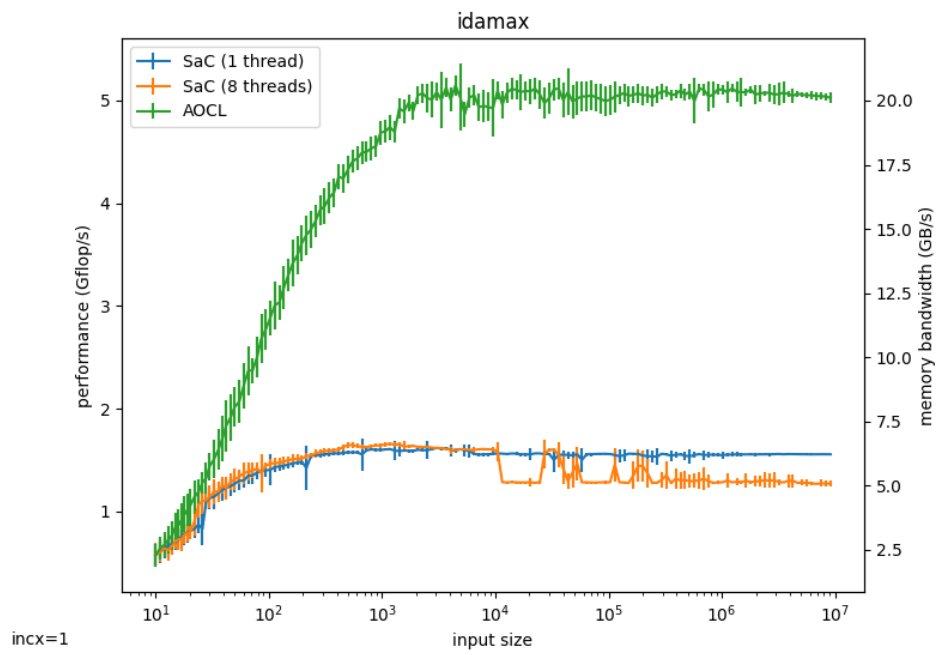




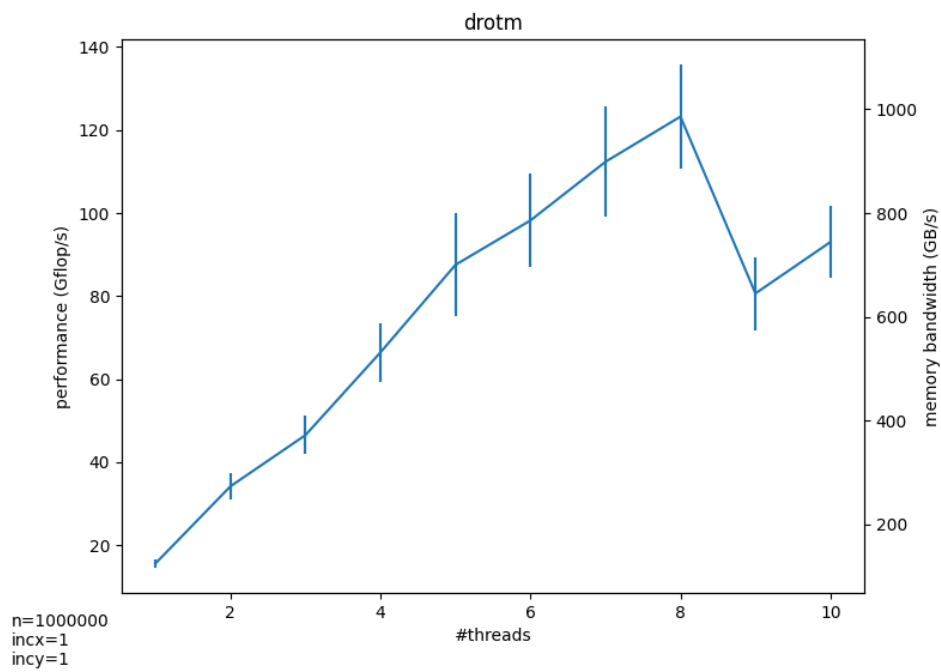
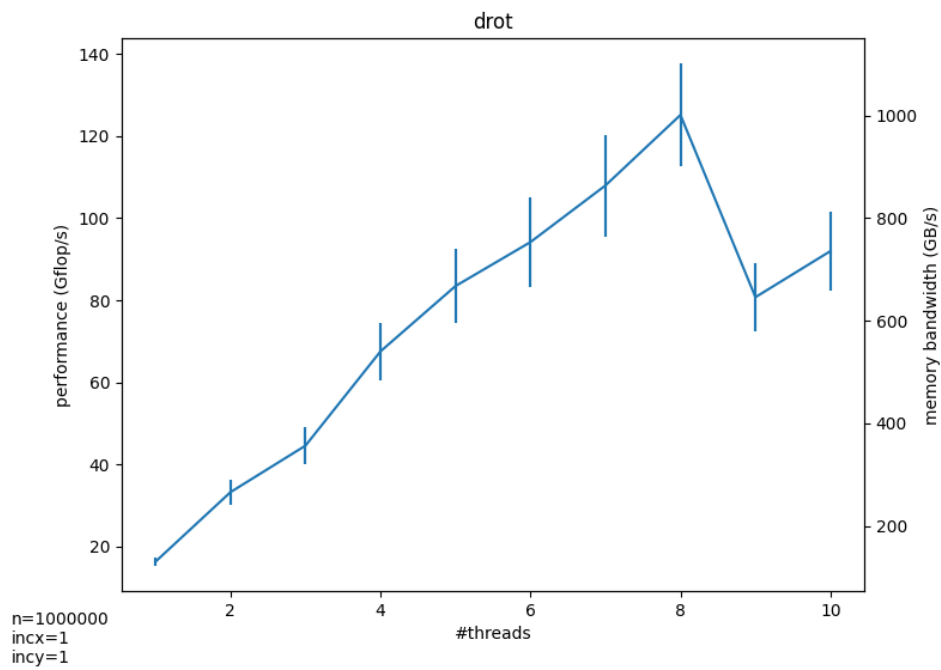


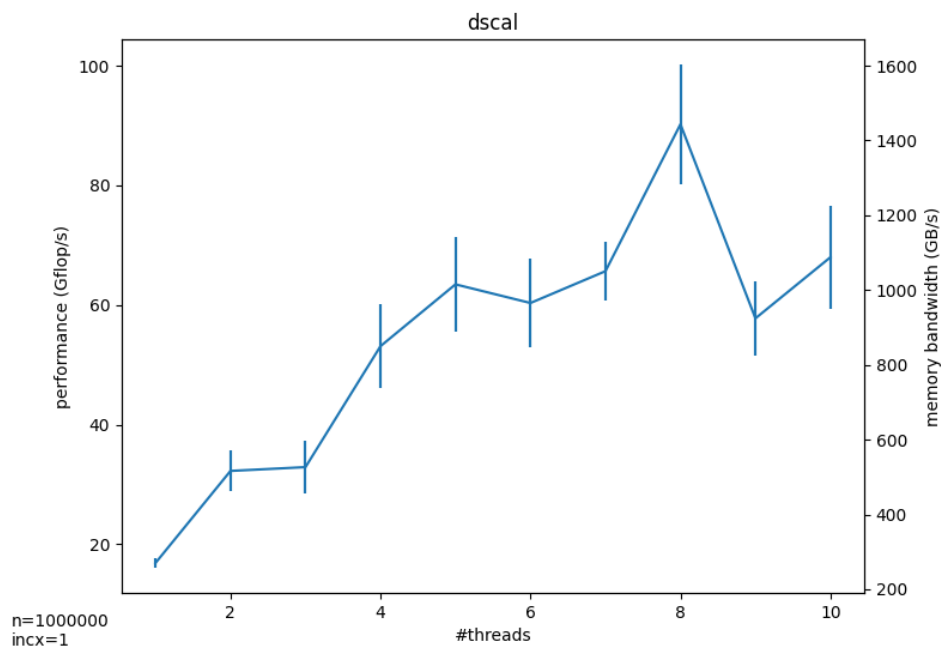
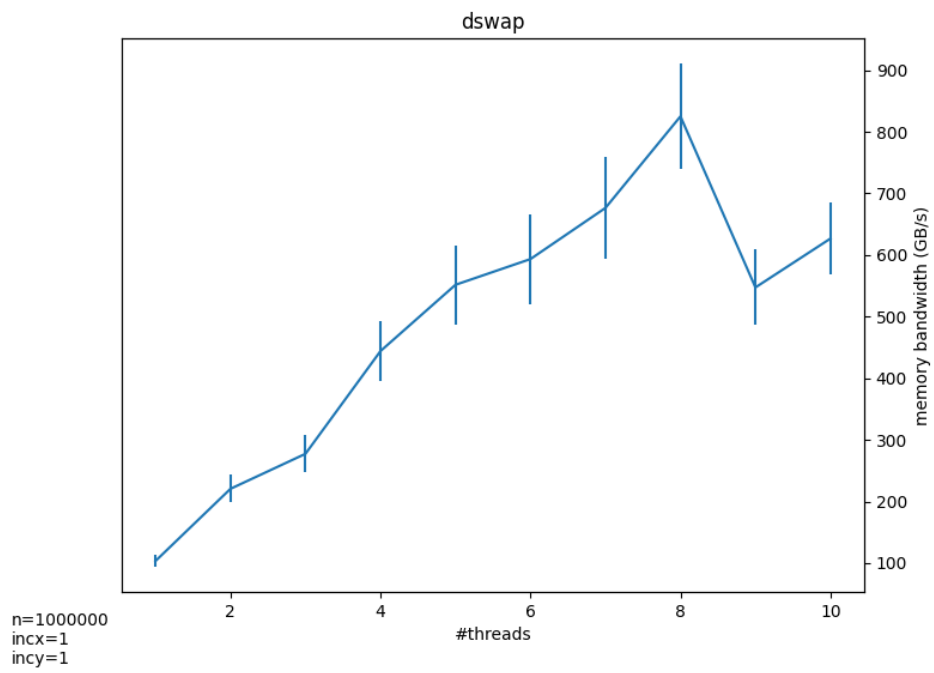


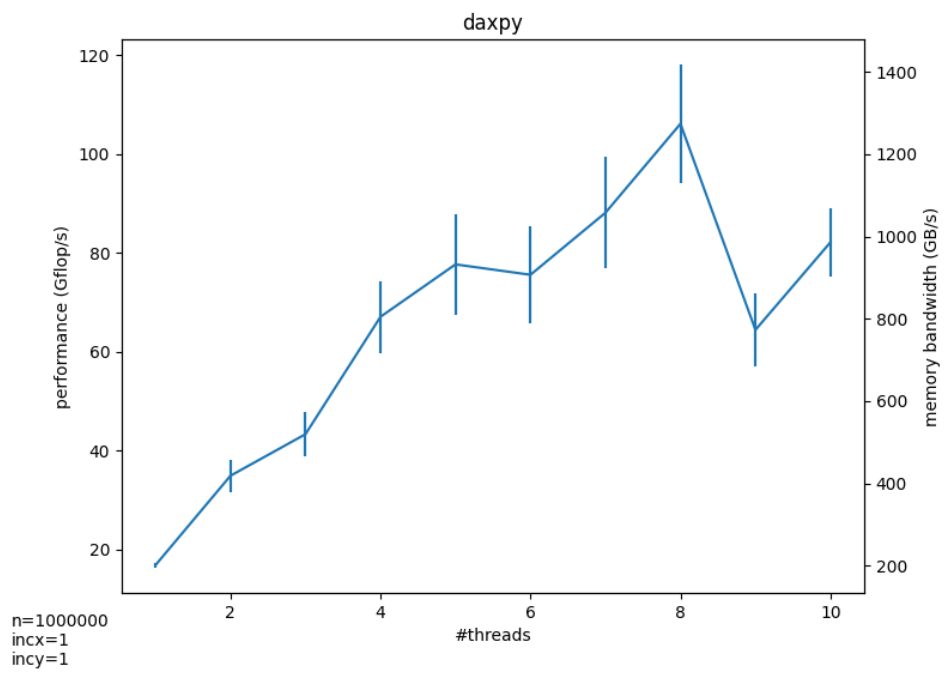
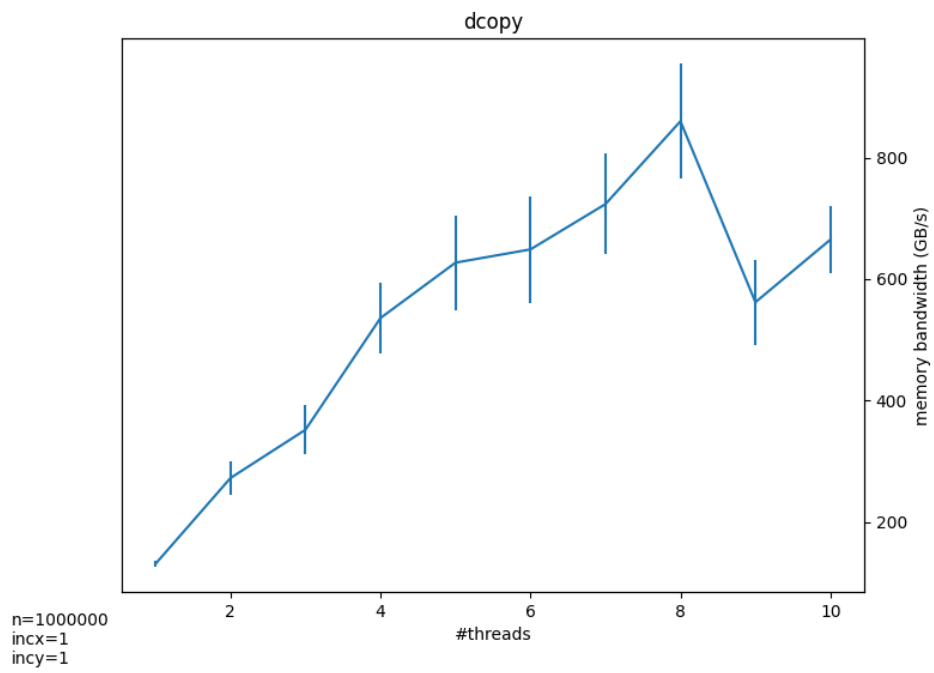


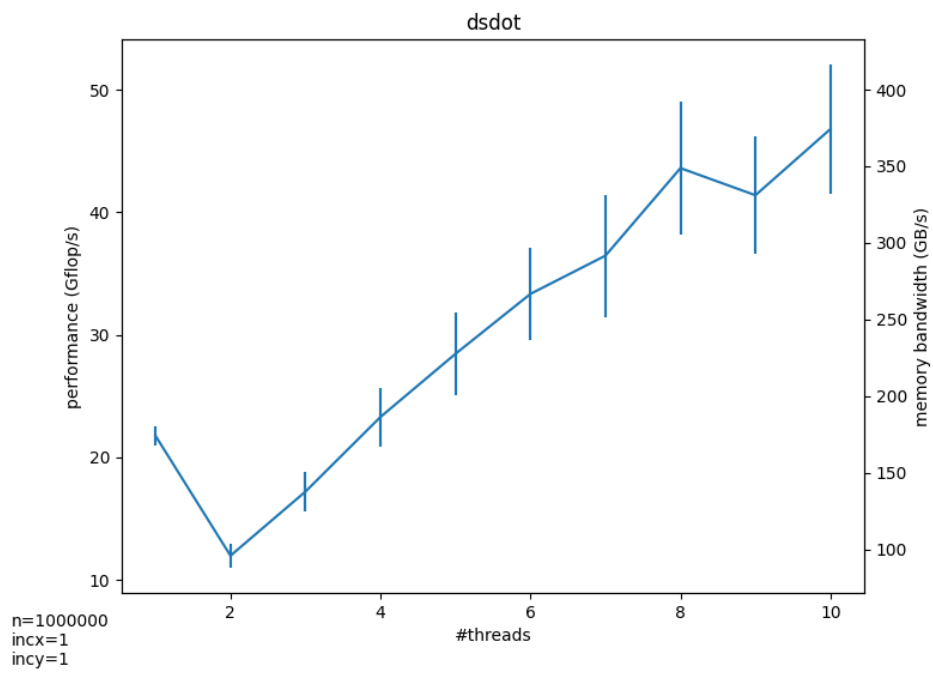
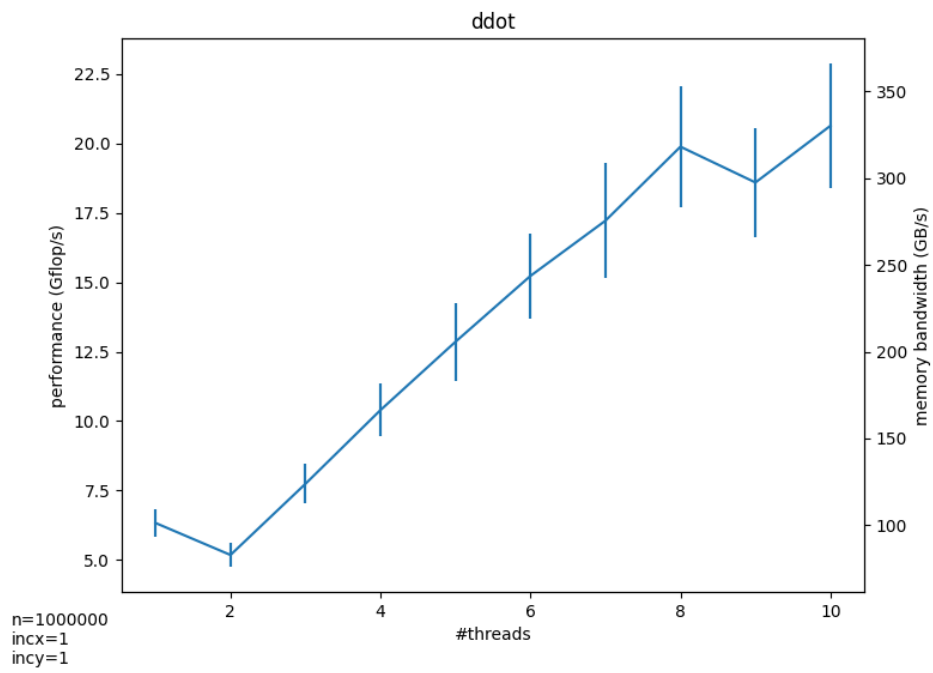


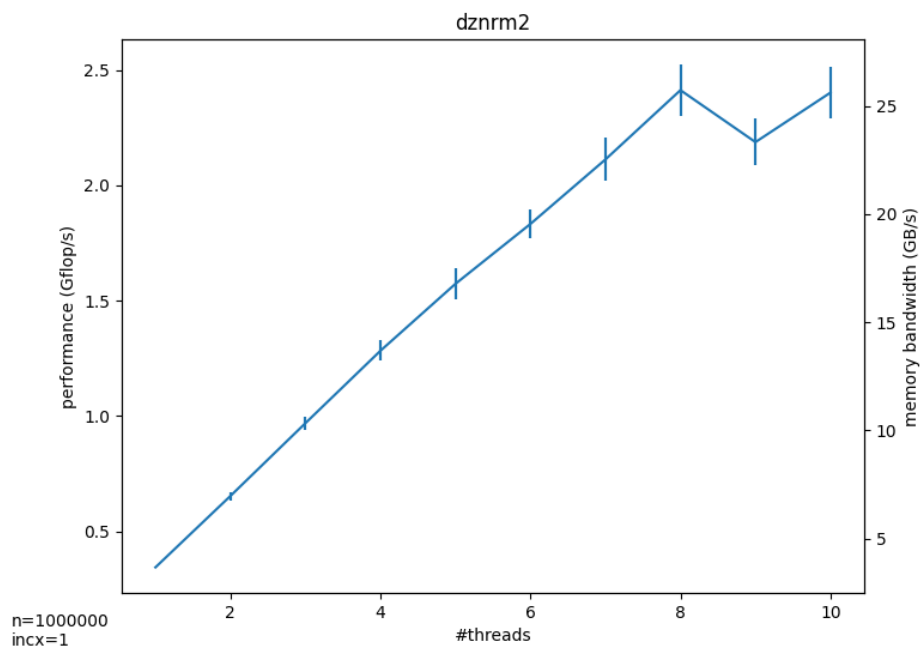
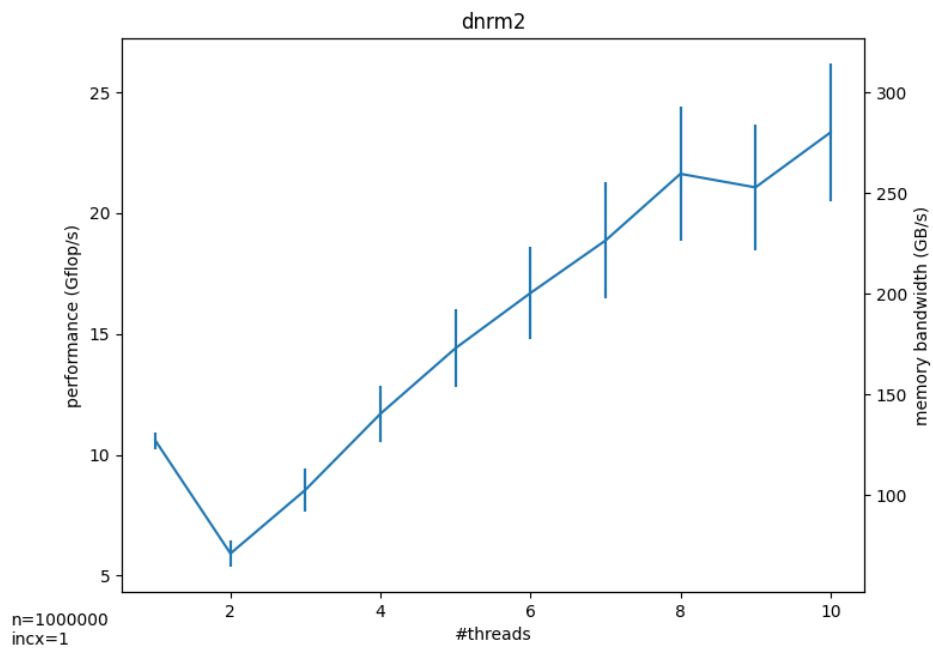
## B.2 Thread Count

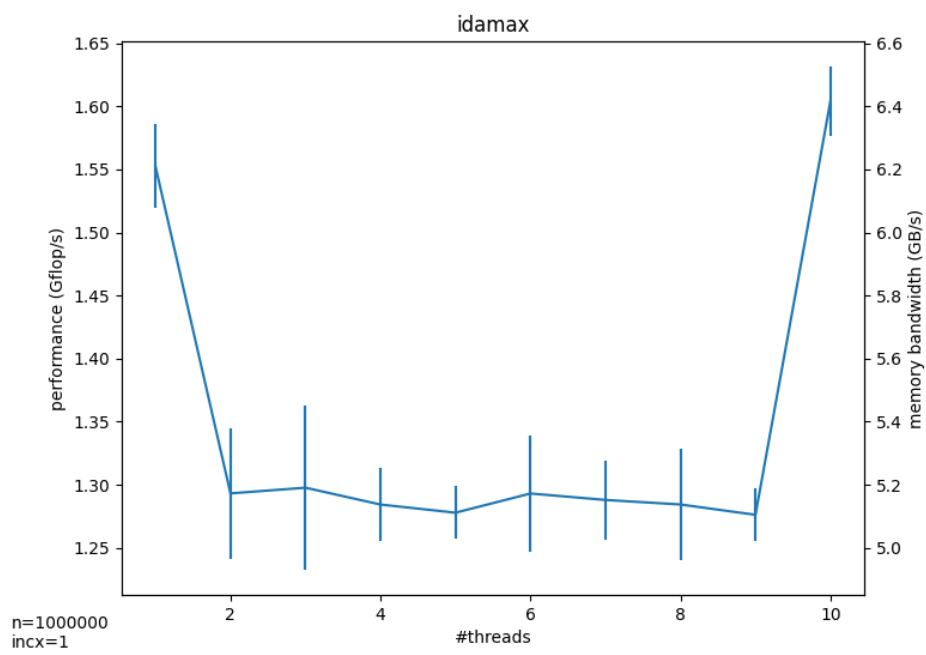
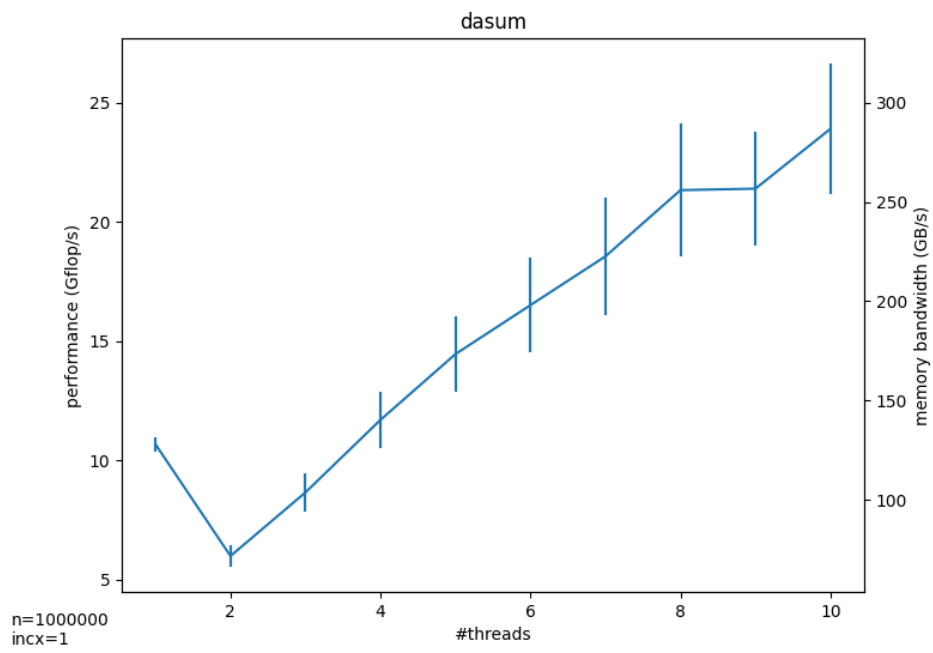




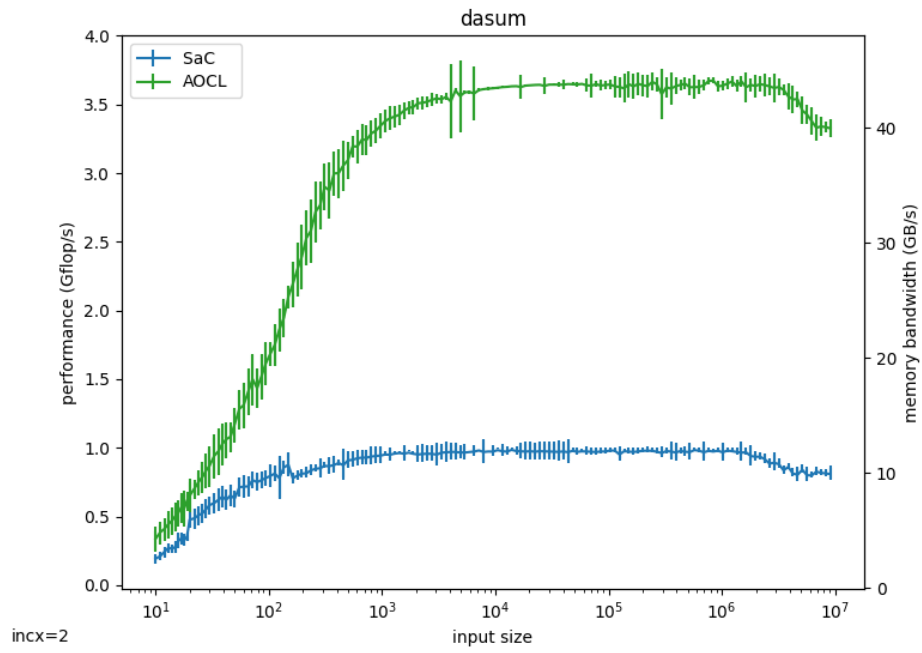
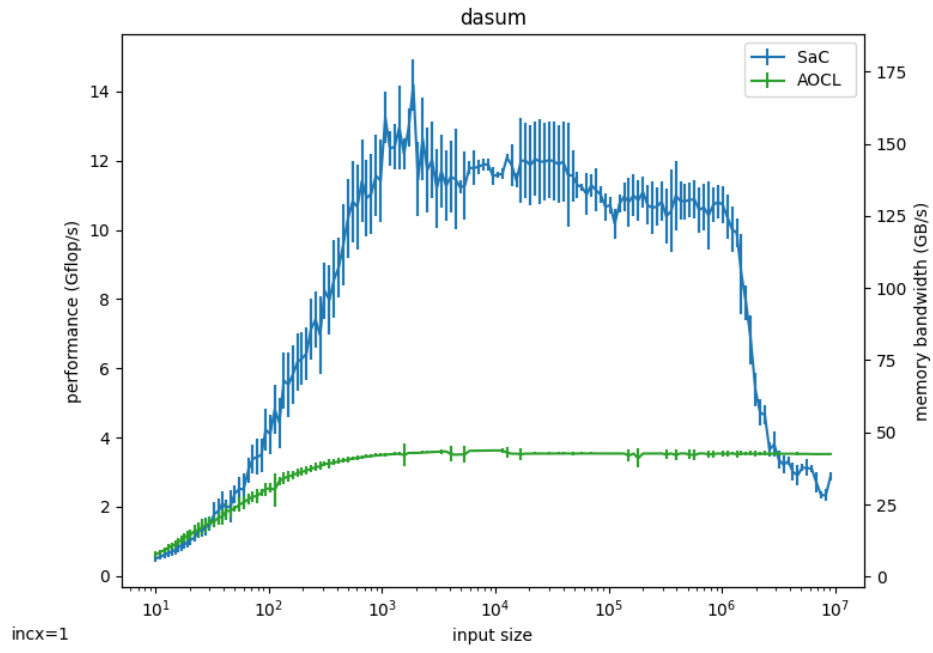




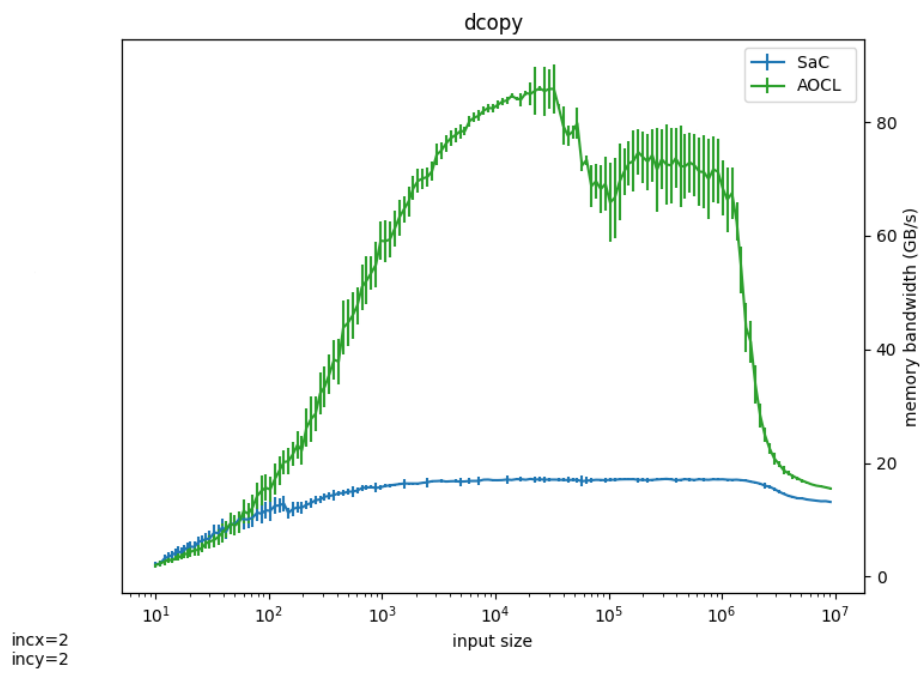
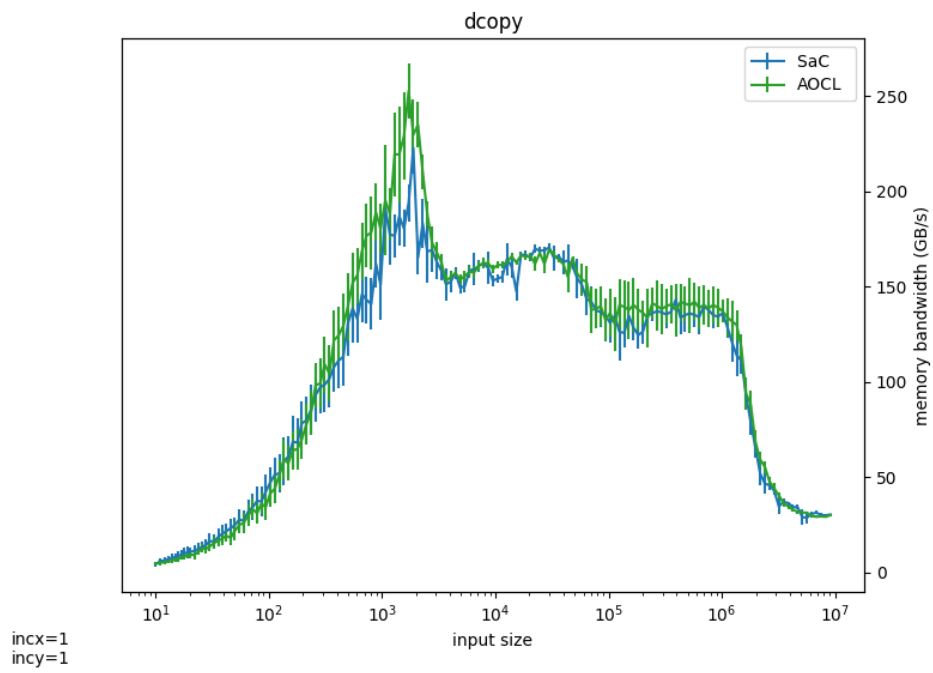


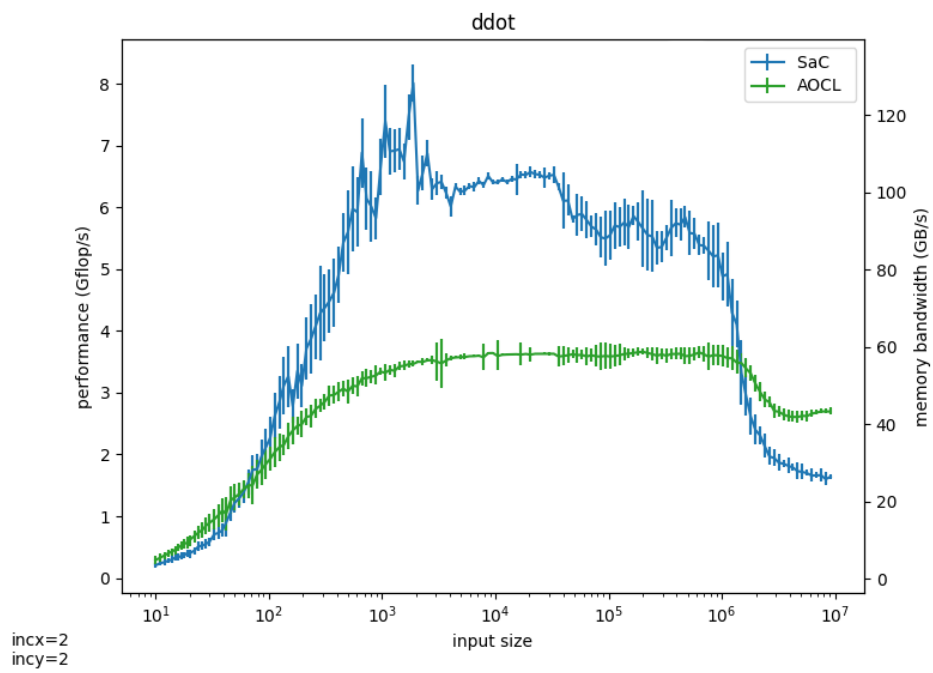
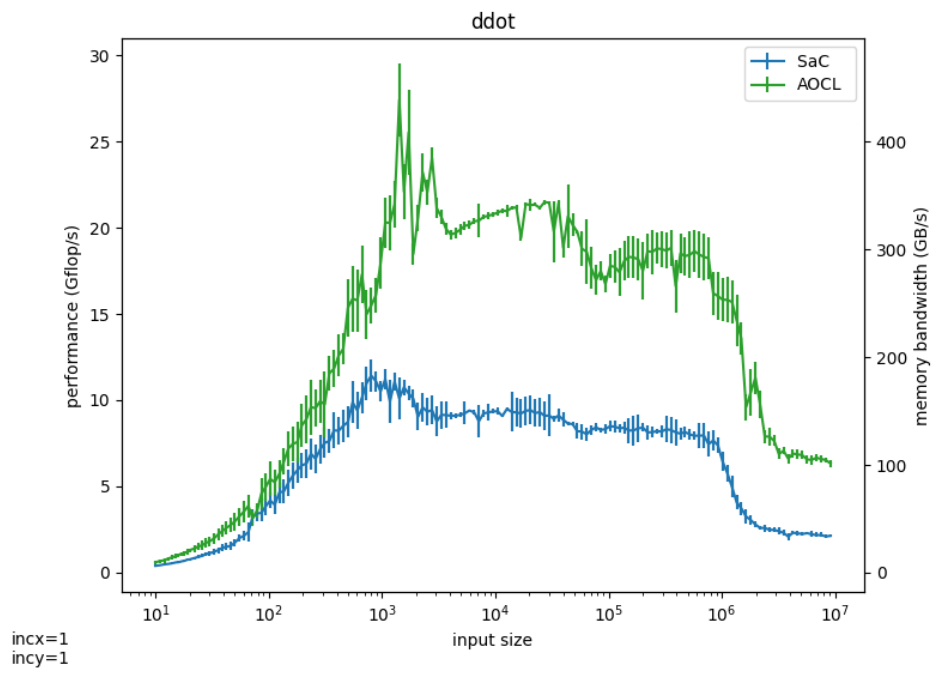


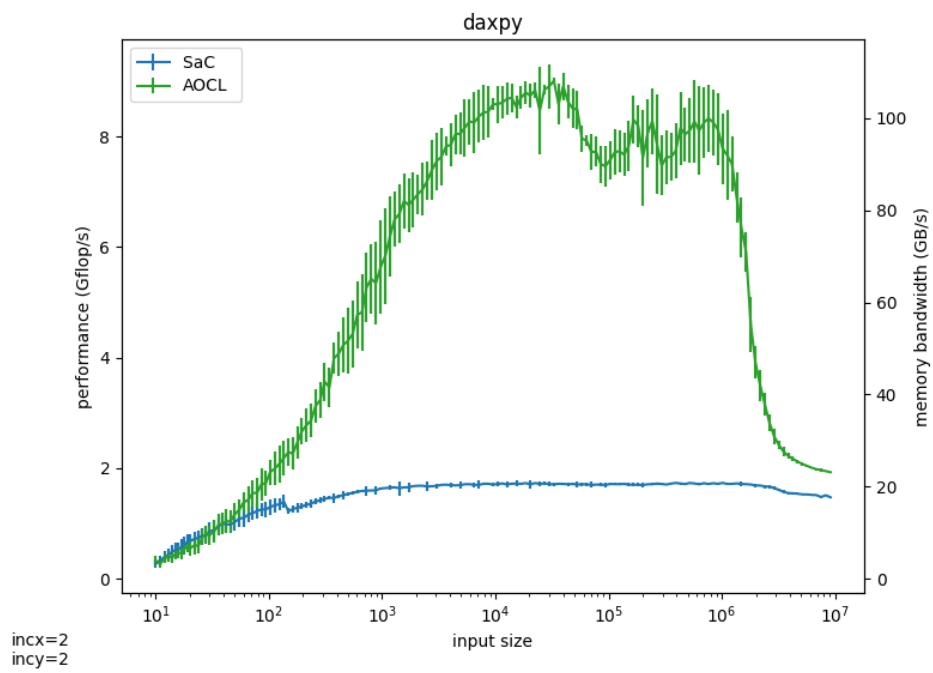
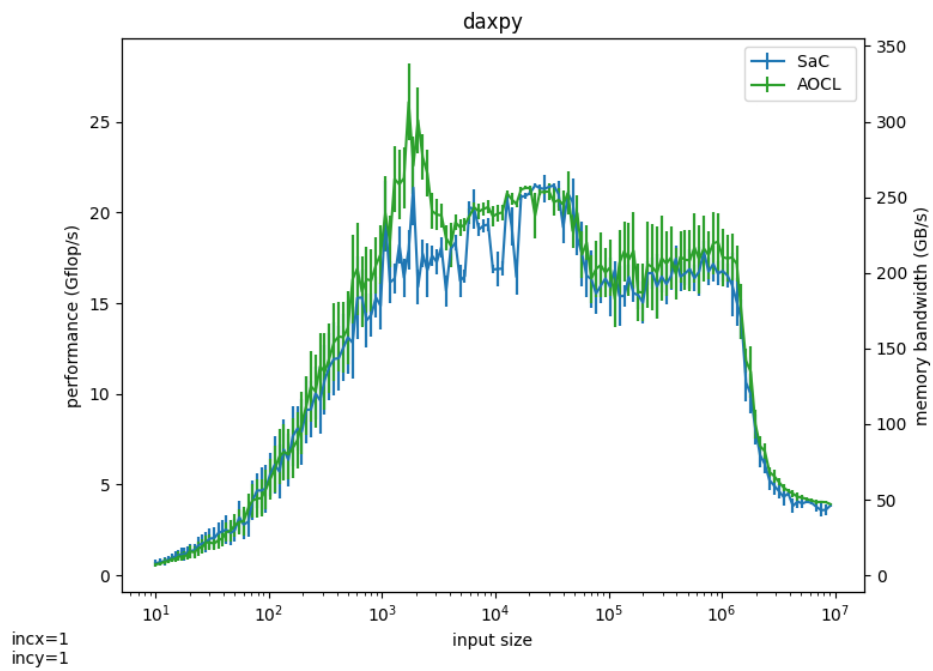
## B.3 Increment Parameters





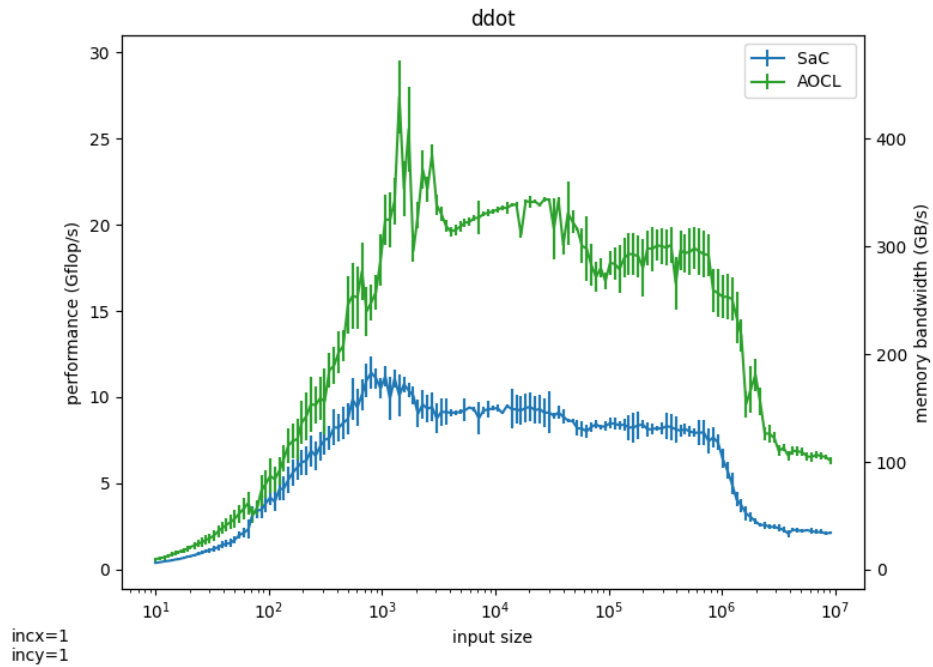




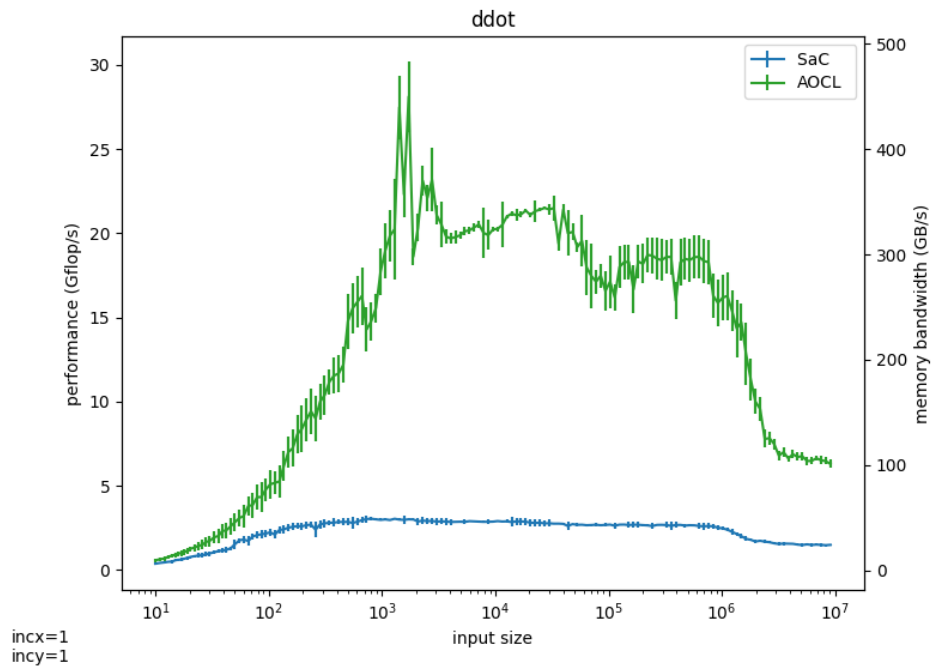


## B.4 Compiler Optimisations

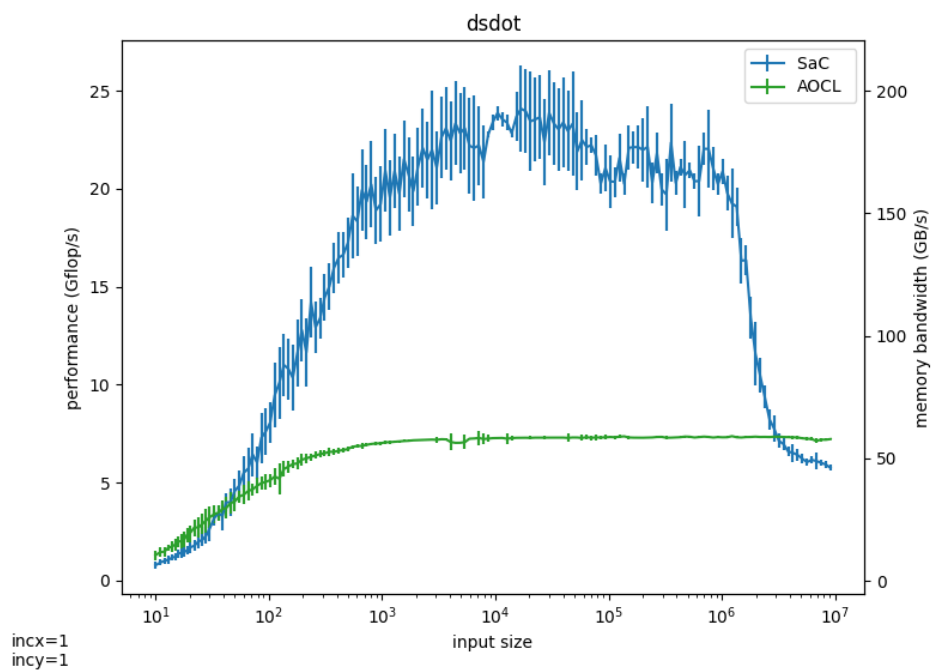
`-ffast-math` enabled:



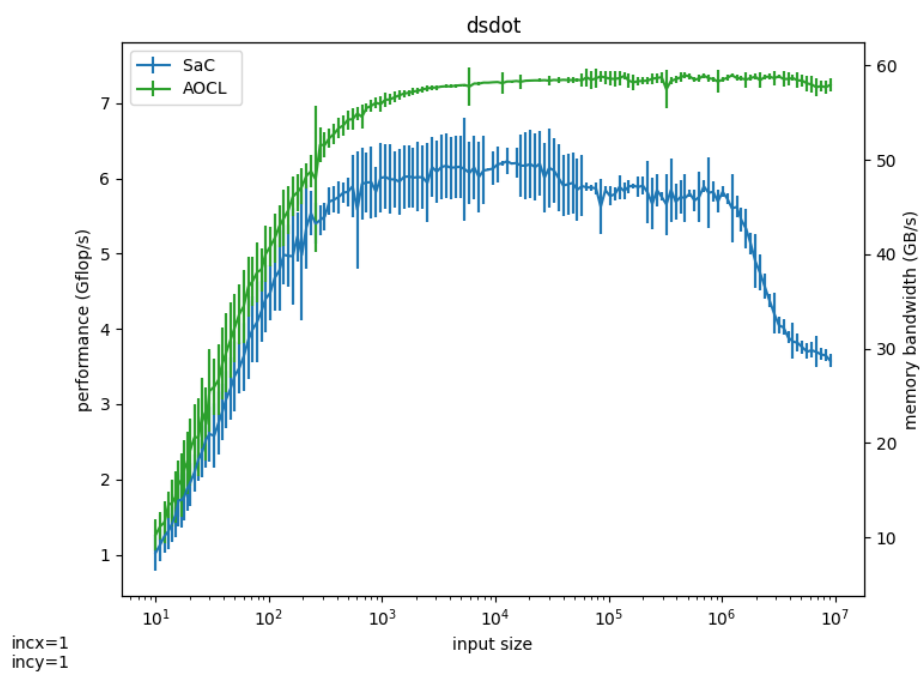
`-ffast-math` disabled:



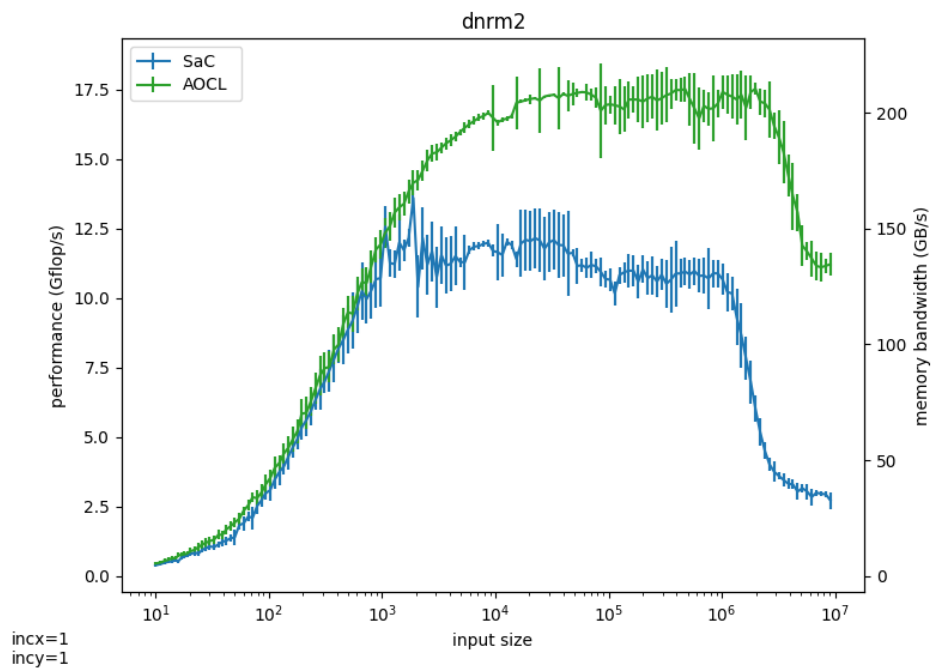
`-ffast-math` enabled:



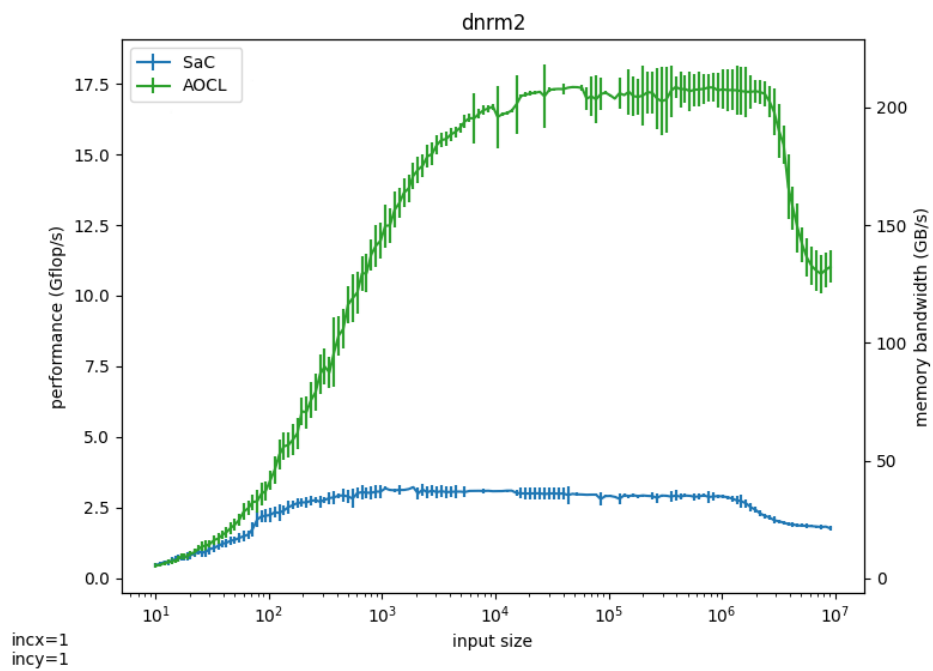
`-ffast-math` disabled:



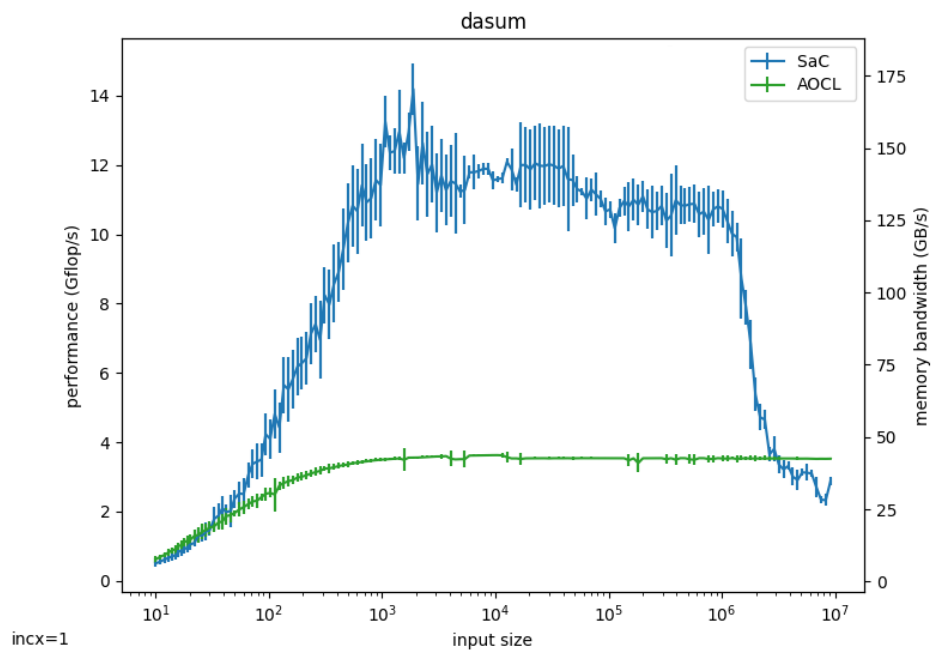
`-ffast-math` enabled:



-ffast-math disabled:



-ffast-math enabled:



-ffast-math disabled:

