Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik
Programming Languages and Software Technology

Bachelor's Thesis

Automatic locality-friendly array interface extension of numerical functions through C++ template metaprogramming

Selim Arslanbek
selim.arslanbek@student.uni-tuebingen.de

19-11-2015

Supervisor

Prof. Dr. Klaus Ostermann
klaus.ostermann@uni-tuebingen.de

Reviewer

Paolo Giarrusso, M.sc.
paolo.giarrusso@uni-tuebingen.de

Arslanbek, Selim:
Automatic locality-friendly array interface extension of numerical functions through C++ metaprogramming
Bachelor's Thesis, Computer Science
Eberhard Karls Universität Tübingen
Period: 19.07.2015-19.11.2015

Abstract

In numerical applications it is often necessary to access array elements using specific patterns. Previous research showed that (automatic) translation of existing `C` code avoids the need for a copy of the data but introduces many changes to the code. Using `C++` template metaprogramming, we can separate the access pattern from the computing code and reduce the need for modifications. The resulting binary offers the same perfomance benefits as hand-written `C` code.

Acknowledgements

I would like to thank

- **Prof. Klaus Ostermann**
  for suggesting a topic that matches my interests very well, taking great interest in its progress, discussing results and giving me the outlook of publishing a paper on it.

- **Paolo Giarrusso**
  for providing helpful expert programming and writing advice and taking the time to discuss the results

- **The (free/open) software communities** behind
  Linux, GNU, clang, texlive, texmaker

The following template was used as a starting point for this thesis:
`http://it.inf.uni-tuebingen.de/theses/BachelorLatexEnglisch.zip`

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Existing Work . . . 3

1.3 New Approach . . . 3

1.4 Contributions . . . 6

2 Transforming Functions **7**

2.1 Transformations . . . 7

2.1.1 Striding . . . 7

2.1.2 Blockstriding . . . 7

2.1.3 Pointer Arithmetic . . . 8

2.1.4 SSE Intrinsics . . . 10

3 Implementation **12**

3.1 Iterator Class . . . 12

3.1.1 Strided Access . . . 13

3.1.2 Blockstrided Access . . . 13

3.2 SSE functions . . . 16

3.2.1 Strided Access . . . 16

3.2.2 Blockstrided Access . . . 18

3.3 Alignment Checks . . . 20

3.4 Full Examples . . . 22

3.4.1 Simple C Function . . . 22

3.4.2 Recursive Function with Pointer Arithmetic 23
3.4.3 SSE: Single Precision 24
3.4.4 SSE: Double Precision 25

4 Evaluation 26

4.1 Test Procedure 26
4.2 Function Variants 27
4.3 Tested Functions 27
4.4 Criteria 28
4.5 Setup 29

5 Results and Conclusion 30

5.1 Interpretation 30
5.1.1 Summary 31
5.1.2 Details 31
5.2 Limitations/Future Work 33
5.2.1 Functions 33
5.2.2 Iterator Operations and Chaining 33
5.2.3 Recursion, Pointers and Blockstriding 33
5.2.4 SSE code duplication 33
5.3 Conclusion 34

Appendices 36

A Manual 36

B Complete Plots 37

List of Tables 53

List of Figures 53

References 57

Chapter 1

Introduction

1.1 Motivation

Numerical libraries written in `C` often operate on one- or multidimensional arrays and expect data to be contiguous in memory. More abstract access patterns are then implemented by copying the required data to and from a linear array. This often introduces a significant overhead in large computations, because RAM access is much slower than using cached values, and memory bandwidth is limited.

Consider the simple function in fig. 1.1: if we want it to read only every second element of the input, we have to copy the input as in fig. 1.2.

```
double sum(double *in, size_t len) {
   double sum = 0;
   for (size_t i = 0; i < len; ++i)
      sum += in[i];
   return sum;
}
```

Figure 1.1: a function with linear access

```
len /= 2; // the number of elements effectively accessed
for (size_t i = 0; i < len; ++i)
   strided[i] = in[i*2];
double result = sum(strided, len);
```

Figure 1.2: applying striding to the original function via copying

The need for such abstract access patterns can often be seen in image processing, where operations need to be performed on a single color channel or section of an array of pixels.

Figure 1.3: original image

Figure 1.4: image with manipulated red color channel

I used *strided* access (see section 2.1.1) on the pixel data of the image in fig. 1.3 to access the red color channel and multiply its value, ignoring overflows. The result is fig. 1.4.

When using a function that performs a small number of accesses to data, copying all the data needed beforehand is more costly than accessing it in-

place and relying on caching. A way of preventing the memory bottleneck and making use of cache locality of the data is to include the access pattern inside the library function.

```
1 double sum(double *in, size_t len, int stride) {
2    double sum = 0;
3    for (size_t i = 0; i < len; ++i)
4       sum += in[i*stride];
5    return sum;
6 }
```

Figure 1.5: including the strided access pattern in a function

```
1 double result = sum(in, len, 2);
```

Figure 1.6: using the transformed function

1.2 Existing Work

To avoid manually adapting each function, Hess, Püschel and Gross (ETH Zurich) described a custom source-to-source transformation tool written in `python`, using the `clang C` language parser [6]. This tool also handles pointer arithmetic in recursive calls and `SSE` intrinsics[1] , where packed load and store instructions are translated into separate ones when the values are not contiguous or appropriately aligned. Beni Hess' Master's thesis explains the implementation of this tool [5]. Figure 1.7 illustrates the workflow for the creation of a program using this method. If the base function is changed or a different

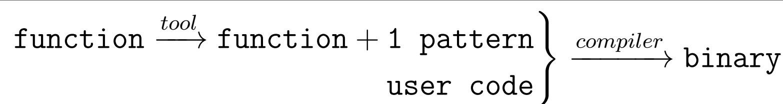


Figure 1.7: illustration of source-to-source translation usage

access pattern has to be applied, the source code must be processed by the translation tool again.

1.3 New Approach

In this thesis I investigate the possibility of generalizing array access and improving performance using `C++` metaprogramming functionality (templates,

[1]SSE: Intel Streaming SIMD (Single Input, Multiple Data) extensions

function/operator overloading and iterators) without the necessity of a source-to-source transformation tool. The implementation can be summarized in the following steps:

1. Move the index calculation of the access pattern to a separate iterator class (shown in section 3.1) that contains a pointer to the array and overloaded array access/pointer arithmetic operators.

2. (Where needed,) overload `SSE` functions with a wrapper that takes an iterator instead of a memory address. (Section 3.2)

3. Modify the original function's signature using a `typename` template for the array parameter. The function is now pattern-agnostic and uses the methods of the iterator to access the array. (Figure 1.9)

4. User code instantiates an iterator for the desired pattern and passes it as a parameter to the function. (Section 3.4)

Figure 1.8 illustrates the workflow for a program that uses the new approach; Figure 1.9 shows the enhanced example function.

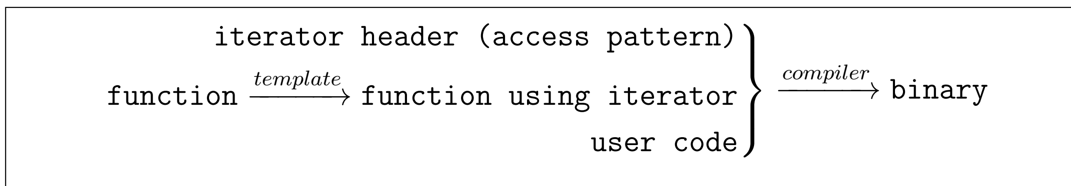


Figure 1.8: illustration of iterator template usage

```
template <typename input>
double sum(input& in, size_t len) {
   double sum = 0;
   for (int i = 0; i < len; ++i)
      sum += in[i];
   return sum;
}
```

Figure 1.9: enhanced function using an abstract array interface

Figures 1.10 and 1.11 show benchmarks where the metaprogramming approach leads to the same performance as plain source translation. The green and blue bars show the timings for the function enhanced via metaprogramming (see section 3.1 for a distinction), the orange bar shows the timing for the translated `C` function as shown in fig. 1.5 that would be produced by the tool introduced in [6]. The purple bar corresponds to the original function

in fig. 1.1 and the yellow bar to the original function applied to a copy as in fig. 1.2; these two serve as a reference showing the overhead of copying and the possible speedup achieved by the interface extension.

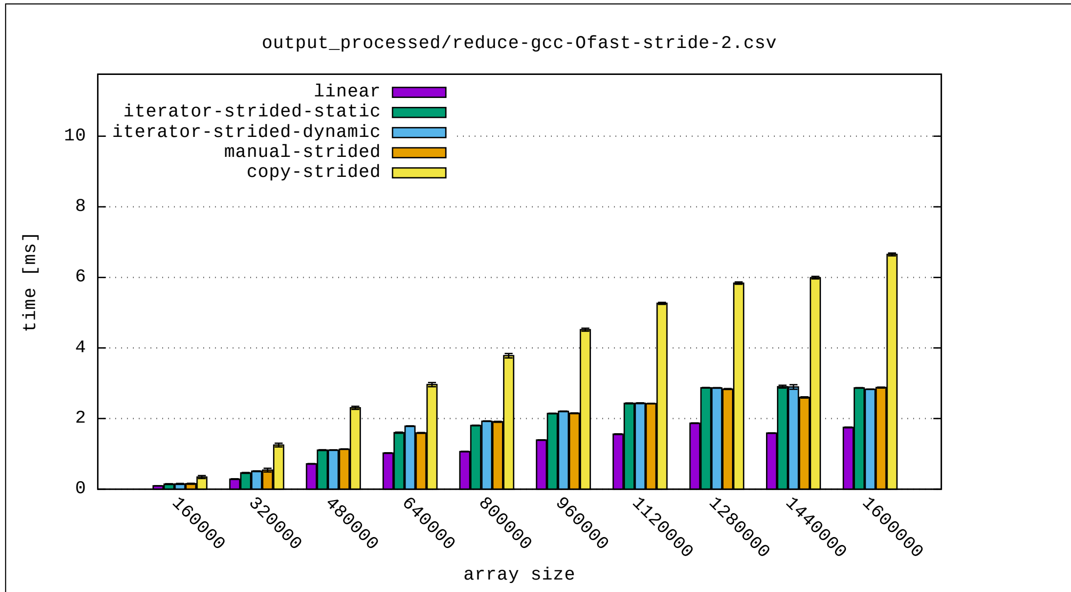


Figure 1.10: selected results: summation of an array with stride distance 2

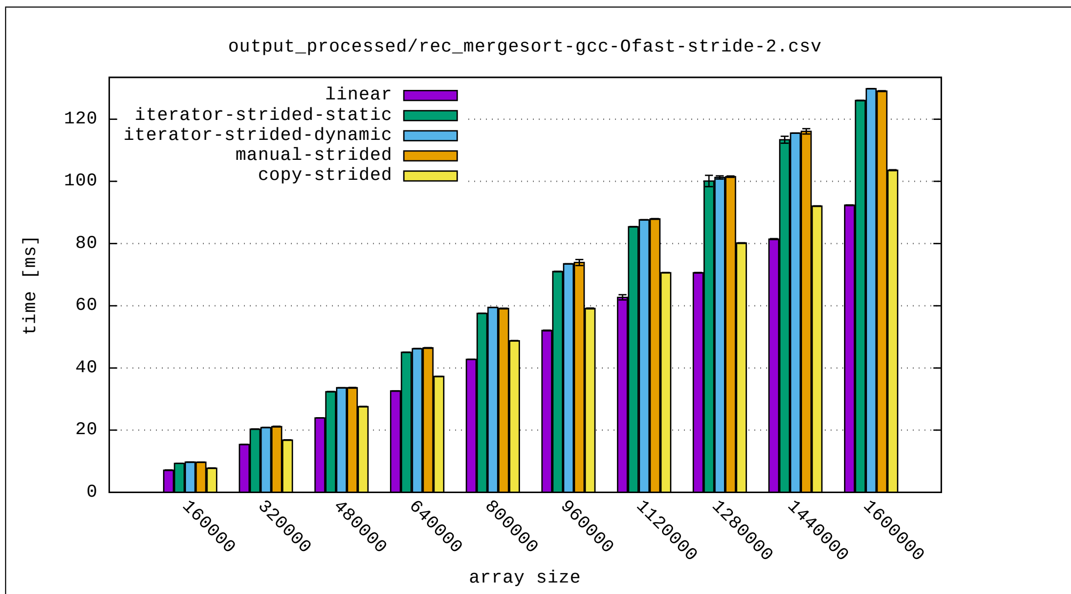


Figure 1.11: selected results: recursive mergesort

I designed the test environment to be similar to the one described in [6, p.7]: benchmarks of the same functions and randomized input of the same sizes (see chapter 4). A comparison of the runtime of new `C++` code with the transformed `C` code shows that the same performance can be achieved. In

functions whith many recurring accesses, copying is still clearly better than any optimization attempt, as can be seen in fig. 1.11.

The access patterns covered in my tests are *striding* and *blockstriding* (see section 2.1). These often occur when dealing with calculations of parts of matrices like in in image processing applications (images/subimages). [6]

1.4 Contributions

The performance and functionality is largely comparable to `C` source translation, with the following improvements for the user:

- Reduced amount of source modification. Only the function signature needs to be changed, and only once. (see section 3.4)
- Access patterns can be added and modified afterwards in a central place with no change to the library code. (see section 3.1)
- The enhanced function supports any input/output access pattern provided by the iterator.
- Abstract interface: Details of the access pattern are separated from function code.
- Full support for `#define`s and macros because no separate parser is used.
- Language support is extended to `C++`.

These aspects should help in keeping user code simple and well maintainable.

Chapter 2

Transforming Functions

The following section introduces the changes to array addressing and SSE operations that lead to the desired abstraction of array accesses. These are the enhancements made by hand or via source-to-source translation as explained in [6] that I will later add through metaprogramming (chapter 3).

2.1 Transformations

2.1.1 Striding

Striding means accessing the elements of an array that are s steps apart. Figure 1.5 showed a function implementing this access pattern. Array indexing needs to be modified in the following way:

$$\texttt{array[}i\texttt{]} \longrightarrow \texttt{array[}i * s\texttt{]}$$
$$\text{striding distance } s > 0$$

2.1.2 Blockstriding

Blockstriding means accessing the elements in blocks of size b that are s steps apart. This can be paraphrased as a mix of linear access and striding (see section 2.1.1). Figure 2.1 shows our `sum` function transformed to perform blockstriding.

Array indexing needs to be modified in the following way:

$$\texttt{array[}i\texttt{]} \longrightarrow \texttt{array[}\tfrac{i}{b} * s + i \mod b\texttt{]}$$
$$\text{striding distance } s,\ s > 0$$
$$\text{block size } b,\ 0 < b \leq s$$

```
1  double sum(double *in, size_t len, int s, int b) {
2     double sum = 0;
3     for (size_t i = 0; i < len; ++i)
4        sum += in[i/b*s + i%b];
5     return sum;
6  }
```

Figure 2.1: including the blockstrided access pattern in a function

2.1.3 Pointer Arithmetic

The function in fig. 2.2 is also a summation function, but it is recursive and uses pointer arithmetic to pass a pointer to the current element to itself.

```
1  double sum(double *x, int size) {
2     double tmp1, tmp2;
3     if (size == 1) {
4        return x[0];
5     } else if (size == 2) {
6        return x[0] + x[1];
7     } else {
8        tmp1 = sum(x, size/2);                        // left
9        tmp2 = sum(x + size/2, size/2 + size%2); // right
10       return tmp1 + tmp2;
11    }
12 }
```

Figure 2.2: a recursive summation function using pointers

Striding

Figure 2.3 shows a manually modified version of the funtion in fig. 2.2 that performs *strided* access. The following modification was applied (in analogy to section 2.1.1):

`array + i` ⟶ `array + i*s`

Blockstriding

An application of blockstrided access in a recursive function taking only a pointer to the current element is not possible, because there is no way of knowing the position within the current block. Different blocks would be

```
double sum(double *x, int size, int stride) {
   double tmp1, tmp2;
   if (size == 1) {
      return x[0];
   } else if (size == 2) {
      return x[0] + x[stride];
   } else {
      tmp1 = sum(x, size/2);
      tmp2 = sum(x + size/2*stride, size/2 + size%2);
      return tmp1 + tmp2;
   }
}
```

Figure 2.3: a recursive summation function with strided access using pointers

selected if the starting point is moved. The necessary transformation would be

$$\texttt{(ptr + i) + j} \rightarrow \texttt{ptr + (i+j)} = \texttt{ptr + (i+j)/b*s + (i+j)\%b}$$

This requires `ptr` to be the starting address for a block. However, the recursive function that wants to add `j` so far has no way of knowing what `i` was. The solution in `C` can be to add a parameter for the offset from the first element of the array to the function and always pass the same pointer like so (fig. 2.4):

```
double sum(double *x, int size, int stride, int block,
      int offset) {
   double tmp1, tmp2;
   if (size == 1) {
      return x[offset/block*stride + offset%block];
   } else if (size == 2) {
      return x[offset/block*stride + offset%block]
         + x[(offset+1)/block*stride + (offset+1)%block];
   } else {
      tmp1 = sum(x, size/2, stride, block, 0);
      tmp2 = sum(x, size/2 + size%2, stride, block, size/2);
      return tmp1 + tmp2;
   }
}
```

Figure 2.4: a recursive summation function with strided access using pointers

I left out the application of blockstriding to recursive functions using pointer

arithmetic in my implementation, but will propose one possible solution in section 5.2.3.

2.1.4 SSE Intrinsics

`SSE` intrinsics make it easy to use vector instructions available in modern processors without resorting to assembly language. An `SSE` register is 128 bits wide, and can therefore contain 4 32-bit `float`s or 2 64-bit `double`s. A good source for information on the C functions providing access to `SSE` intrinsics is the interactive online manual by Intel [1]. Many SSE instructions expect the memory address to be aligned in a suitable way, which has to be ensured by the programmer to prevent runtime errors (segmentation faults).

In the case of *strided* access, the packed `_mm_load_p*` instructions must be replaced by a series of separate `_mm_load_s*` and `_mm_shuffle_p*` calls [5, p.14]. The same goes for `_mm_store_p*` calls at the end of computation, which are replaced by successive `_mm_store_s*` and `_mm_shuffle_p*` calls. For values of type `double`, the operations `_mm_load{h,l}_*` and `_mm_store{h,l}_*` can be used to access single elements, avoiding the need to shuffle. I will illustrate the necessary changes below.

Blockstriding may allow the use of packed load/store instructions depending on suitable parameters and alignment. This distinction has to be made using a test function before deciding with method to use. The test function also depends on how the data will be read: in a linear or multidimensional fashion. Section 3.3 introduces these checks.

Single Precision load/store

The code in fig. 2.5 has to be replaced with the one in fig. 2.6 (load). The code in fig. 2.7 has to be replaced with the one in fig. 2.8 (store).

```
1 __m128 four_values = _mm_load_ps(array + index);
```

Figure 2.5: single precision packed load instruction

Double Precision load/store

Double precision load/store instructions are simpler to replace, since there are dedicated instructions to fill the upper and lower halves of a vector register. Figure 2.10 Shows the replacement for the load in fig. 2.9. Figure 2.12 shows the replacement for the store function in fig. 2.11.

```
__m128 a0, a1, a2, a3;
a0 = _mm_load_ss(array +   index * s);
a1 = _mm_load_ss(array + (index+1)*s);
a2 = _mm_load_ss(array + (index+2)*s);
a3 = _mm_load_ss(array + (index+3)*s);
__m128 a0a1 = _mm_shuffle_ps(a0, a1, 0);
__m128 a2a3 = _mm_shuffle_ps(a2, a3, 0);
__m128 result = _mm_shuffle_ps(a0a1, a2a3, 136);
       // 136 == (1000 1000)_2: elements 2 and 0 of each
```

Figure 2.6: single precision strided load instruction

```
_mm_store_ps(array + index, four_values);
```

Figure 2.7: single precision packed store instruction

```
_mm_store_ss(array + index * s   , x);
x = _mm_shuffle_ps(x, x, 57);// 57==(0011 1001)_2 -> 0321 rotate
_mm_store_ss(array + (index+1)*s, x);
x = _mm_shuffle_ps(x, x, 57); // rotate
_mm_store_ss(array + (index+2)*s, x);
x = _mm_shuffle_ps(x, x, 57); // rotate
_mm_store_ss(array + (index+3)*s, x);
x = _mm_shuffle_ps(x, x, 57); // (restore original, optional)
```

Figure 2.8: single precision strided store instruction

```
__m128d two_values = _mm_load_pd(array + index);
```

Figure 2.9: double precision packed load instruction

```
__m128d two_values;
two_values = _mm_loadh_pd(two_values, array +  index * s    );
two_values = _mm_loadl_pd(two_values, array + (index + 1)*s);
```

Figure 2.10: double precision load instructions for strided access

```
_mm_store_pd(array + index, two_values);
```

Figure 2.11: double precision packed store instruction

```
_mm_storeh_pd(array +  index * s    , two_values);
_mm_storel_pd(array + (index + 1)*s, two_values);
```

Figure 2.12: double precision store instructions for strided access

Chapter 3

Implementation

Hess et al. implemented the changes discussed in the previous chapter inside the original code by replacing accesses and function calls [6]. I included them in a separate class/header file that I use where needed. The header file contains `SSE` packed load/store functions that wrap the real functions and depend on the access pattern. The `SSE` function calls in the original source can thus stay the same. With this approach I can limit the modification of the original function to the signature (`typename` template and parameter type). The original function has to be changed only once. It can then use any patterns implemented in an iterator class, because the interface of the class emulates a regular `C` array.

3.1 Iterator Class

Instead of changing the original function code to perform one specific access pattern, I propose an interface that provides an abstract view (*strategy pattern*) of an array and implements the access pattern. The class of this iterator is then used as a typename template in the function's signature.

The iterators have corresponding `SSE` functions, which are shown later (section 3.2).

The overloaded operator `[]` returns a reference to the value at the resulting index of the array instead of the value itself. This makes it possible to use the operator in assignment statements, too. A comparison of the performance for accesses showed no difference, leading to the conclusion that the compiler can deduce when the referenced value is needed directly.

The overloaded operator `+` returns a new view of the array starting at the given offset. This is why, in recursive functions, the iterator has to be passed by value, because each nested call will create a new iterator that stores the current address and emulates a pointer (See section 5.2.3 for a discussion of

limitations). When the original function is not recursive, the new iterator should be passed as a reference to achieve better performance.

Parameters: static/dynamic There are two ways of specifying the *striding factor* and *block size*: either as contructor arguments to the class at runtime, or as template parameters at compile time. The template version leads to better performance, because it allows for more optimization (see the summary of results in 5.1.1). Both versions are presented here for each case.

3.1.1 Strided Access

Figures 3.1 and 3.2 show the respective classes for strided access.

```
template <typename T>
class Strided {
   T* array;
   int stride;
public:
   Strided<T>(T* _array, int _stride):
                array(_array), stride(_stride) {}
   T& operator[](int index) {
      return array[index * stride];
   }
   Strided<T> operator+(int offset) {
      return Strided<T>(array + offset * stride, stride);
   }
};
```

Figure 3.1: striding iterator, with parameter defined at runtime

3.1.2 Blockstrided Access

Figures 3.3 and 3.4 show the respective classes for blockstrided access.

```
template <typename T, int stride>
class CStrided {
   T* array;
public:
   CStrided<T, stride>(T* _array):
                  array(_array) {}
   T& operator[](int index) {
      return array[index * stride];
   }
   CStrided<T, stride> operator+(int offset) {
      return CStrided<T, stride>(array + offset * stride);
   }
};
```

Figure 3.2: striding iterator, with parameter defined at compile time

```
template <typename T>
class BlockStrided {
   T* array;
   int stride, block;
public:
   BlockStrided<T>(T* _array, int _stride, int _block):
      array(_array), stride(_stride), block(_block) {}
   T& operator[](int index) {
      int bnum = index / block;
      return array[bnum * stride + index % block];
   }
   BlockStrided<T> operator+(int offset) {
      return BlockStrided<T>(
            array + (offset/block*stride + offset%block),
            stride, block);
   }
};
```

Figure 3.3: blockstriding iterator, with parameters defined at runtime

```
template <typename T, int block, int stride>
class CBlockStrided {
   T* array;
public:
   CBlockStrided<T, block, stride>(T* _array):
                  array(_array) {}
   T& operator[](int index) {
      int bnum = index / block;
      return array[bnum * stride + index % block];
   }
   CBlockStrided<T, block, stride> operator+(int offset) {
      return CBlockStrided<T, block, stride>(
            array + (offset/block*stride + offset%block));
   }
};
```

Figure 3.4: blockstriding iterator, with parameters defined at compile time

3.2 SSE functions

The iterator class does not overload the following use of the address operator (which would also be non-trivial to implement due to fixed operator precedence):

`&array[i]`

The SSE function wrappers need it as a way to obtain real memory addresses, since the + operator cannot be used in this context anymore (it returns an iterator object instead of a memory address).

The following examples show SSE wrappers for classes with runtime-defined parameters. By replacing the argument type, they can be applied to iterators with compiled-in parameters.

3.2.1 Strided Access

Figures 3.5 and 3.6 show the functions emulating packed load/store when accessing disparate elements, for types double and float.

```
__m128d _mm_load_pd(Strided<double> access) {
   __m128d ret;
   ret = _mm_loadh_pd(ret, &access[0]);
   ret = _mm_loadl_pd(ret, &access[1]);
   return ret;
}
__m128d _mm_loadu_pd(Strided<double> access) {
   return _mm_load_pd(access);
}
void _mm_store_pd(Strided<double> access, __m128d x) {
   _mm_storeh_pd(&access[0], x);
   _mm_storel_pd(&access[1], x);
}
void _mm_storeu_pd(Strided<double> access, __m128d x) {
   _mm_store_pd(access, x);
}
```

Figure 3.5: strided SSE double precision load/store functions

```
__m128 _mm_load_ps(Strided<float> access) {
   __m128 a0, a1, a2, a3;
   a0 = _mm_load_ss(&access[0]);
   a1 = _mm_load_ss(&access[1]);
   a2 = _mm_load_ss(&access[2]);
   a3 = _mm_load_ss(&access[3]);
   a0 = _mm_shuffle_ps(a0, a1, 0);
   a2 = _mm_shuffle_ps(a2, a3, 0);
   return _mm_shuffle_ps(a0, a2, 136);
        // 1000 1000: elements 2 and 0 of each
}
__m128 _mm_loadu_ps(Strided<float> access) {
   return _mm_load_ps(access);
}
void _mm_store_ps(Strided<float> access, __m128 x) {
   _mm_store_ss(&access[0], x);
   x = _mm_shuffle_ps(x, x, 57); // 0011 1001 -> 0321: rotate
   _mm_store_ss(&access[1], x);
   x = _mm_shuffle_ps(x, x, 57); // rotate
   _mm_store_ss(&access[2], x);
   x = _mm_shuffle_ps(x, x, 57); // rotate
   _mm_store_ss(&access[3], x);
   x = _mm_shuffle_ps(x, x, 57); // (restore original, optional)
}
void _mm_storeu_ps(Strided<float> access, __m128 x) {
   _mm_store_ps(access, x);
}
```

Figure 3.6: strided SSE single precision load/store functions

3.2.2 Blockstrided Access

Unaligned when the access parameters do not allow packed load/store, the function bodies are the same as for strided access.

Aligned I use a different class name (`BlockStridedAligned`) and corresponding functions if the the data are correctly aligned (see section 3.3). The functions for aligned load/store are then just wrappers for the standard function. Figures 3.7 and 3.8 show the respective functions for the aligned case.

```
__m128d _mm_load_pd(BlockStridedAligned<double> access) {
   return _mm_load_pd(&access[0]);
}
__m128d _mm_loadu_pd(BlockStridedAligned<double> access) {
   __m128d ret;
   ret = _mm_loadh_pd(ret, &access[0]);
   ret = _mm_loadl_pd(ret, &access[1]);
   return ret;
}
void _mm_store_pd(BlockStridedAligned<double> access,
                 __m128d x) {
   _mm_store_pd(&access[0], x);
}
void _mm_storeu_pd(BlockStridedAligned<double> access,
                 __m128d x) {
   _mm_storeh_pd(&access[0], x);
   _mm_storel_pd(&access[1], x);
}
```

Figure 3.7: aligned blockstriding SSE double precision load/store functions

```
__m128 _mm_load_ps(BlockStridedAligned<float> access) {
    return _mm_load_ps(&access[0]);
}
__m128 _mm_loadu_ps(BlockStridedAligned<float> access) {
    __m128 a0, a1, a2, a3;
    a0 = _mm_load_ss(&access[0]);
    a1 = _mm_load_ss(&access[1]);
    a2 = _mm_load_ss(&access[2]);
    a3 = _mm_load_ss(&access[3]);
    a0 = _mm_shuffle_ps(a0, a1, 0);
    a2 = _mm_shuffle_ps(a2, a3, 0);
    return _mm_shuffle_ps(a0, a2, 136);
         // 1000 1000: elements 2 and 0 of each
}
void _mm_store_ps(BlockStridedAligned<float> access, __m128 x) {
    _mm_store_ps(&access[0], x);
}
void _mm_storeu_ps(BlockStridedAligned<float> access, __m128 x) {
    _mm_store_ss(&access[0], x);
    x = _mm_shuffle_ps(x, x, 57); // 0011 1001 -> 0321: rotate
    _mm_store_ss(&access[1], x);
    x = _mm_shuffle_ps(x, x, 57); // rotate
    _mm_store_ss(&access[2], x);
    x = _mm_shuffle_ps(x, x, 57); // rotate
    _mm_store_ss(&access[3], x);
    x = _mm_shuffle_ps(x, x, 57); // (restore original, optional)
}
```

Figure 3.8: aligned blockstriding SSE single precision load/store functions

3.3 Alignment Checks

A memory address is aligned with respect to SSE when it is divisible by 16, i.e. it lies on a 16-byte boundary. When performing *blockstriding*, we have to check the *stride* distance and *block* size as well. Additionally, when viewing the array in 2 dimensions, by defining *width* and *height*, we have to search for possible violations of alignment. I did not include the runtime of this check in the benchmark because it has to be performed before running any of the implementations of a function that uses SSE.
Figures 3.9 and 3.10 show the checks for both views.

```
1  bool aligned(float *src, int stride, int block) {
2     if ( ((uintptr_t) src) % 16 == 0
3          && block  % 4 == 0
4          && stride % 4 == 0)
5        return true;
6     return false;
7  }
8
9  bool aligned(double *src, int stride, int block) {
10    if ( ((uintptr_t) src) % 16 == 0
11         && block  % 2 == 0
12         && stride % 2 == 0)
13       return true;
14    return false;
15 }
```

Figure 3.9: 1D alignment checks for single and double precision

```
bool aligned2d(float *src, int width, int height,
               int stride, int block) {
   if (block % 4 != 0 || stride % 4 != 0)
      return false;
   for (int y = 0; y < height; ++y) {
      for (int x = 0; x < width; x += 4) {
         int index = x + y*width;
         index = index/block*stride + index%block;
         if ((uintptr_t) &src[index] % 16 != 0)
            return false;
      }
   }
   return true;
}

bool aligned2d(double *src, int width, int height,
               int stride, int block) {
   if (block % 2 != 0 || stride % 2 != 0)
      return false;
   for (int y = 0; y < height; ++y) {
      for (int x = 0; x < width; x += 2) {
         int index = x + y*width;
         index = index/block*stride + index%block;
         if ((uintptr_t) &src[index] % 16 != 0)
            return false;
      }
   }
   return true;
}
```

Figure 3.10: 2D alignment checks for single and double precision

3.4 Full Examples

3.4.1 Simple C Function

Figure 3.11 shows `fir`, a simple C function that calculates a finite impulse response of a signal as the average of the current value and the previous one.

```
void fir(double *in, double *out, int len) {
        for (int i = 0; i < len-1; i++)
                out[i] = (in[i+1] + in[i])/2;
}
```

Figure 3.11: fir: original version

To enable the use of the new iterator classes with this function, I have to replace only the signature, as shown in fig. 3.12. (The function is renamed because I use it in the same source file as the original one.)

```
template <typename in_iter, typename out_iter>
void fir_iterator(in_iter& in, out_iter& out, int len) {
        for (int i = 0; i < len-1; i++)
                out[i] = (in[i+1] + in[i])/2;
}
```

Figure 3.12: fir: enhanced version

The user can call `fir_iterator` with `in` and `out` being objects of the presented iterator classes. Any combination of input and output access patterns supported by the classes is thus possible.

3.4.2 Recursive Function with Pointer Arithmetic

`rec_reduce` recursively calculates the sum of the elements of an array. Although it may be of little practical use, it serves well as an example for the functionality of the iterator. Figure 3.13 shows the `C` version. I obtain the enhanced version in `C++` (fig. 3.14) by also just using a templated signature.

```
double rec_reduce(double *x, int size) {
        double tmp1, tmp2;
        if (size == 1) {
                return x[0];
        } else if (size == 2) {
                return x[0] + x[1];
        } else {
                tmp1 = rec_reduce(x, size/2);
                tmp2 = rec_reduce(x + size/2, size/2 + size%2);
                return tmp1+tmp2;
        }
}
```

Figure 3.13: rec_reduce: original version

```
template <typename in_iterator>
double rec_reduce_iterator(in_iterator x, int size) {
        double tmp1, tmp2;
        if (size == 1) {
                return x[0];
        } else if (size == 2) {
                return x[0] + x[1];
        } else {
                tmp1 = rec_reduce_iterator(x, size/2);
                tmp2 = rec_reduce_iterator(x + size/2,
                                size/2 + size%2);
                return tmp1+tmp2;
        }
}
```

Figure 3.14: rec_reduce: enhanced version

3.4.3 SSE: Single Precision

`fir_sse_float` performs the same operations as `fir`, but on 4 `float` elements at a time. Figures 3.15 and 3.16 also differ only in the function signature.

```
void fir_sse_float(float *in, float *out, int len) {
        int i = 0;
        __m128 i0, i1, two = _mm_set_ps1(2.0);
        for (; i < len-5; i+=4) {
                i0 = _mm_load_ps(in + i);
                i1 = _mm_loadu_ps(in + (i+1));
                i0 = _mm_add_ps(i0, i1);
                i0 = _mm_div_ps(i0, two);
                _mm_store_ps(out + i, i0);
        }
        for (; i < len-1; ++i)
                out[i] = (in[i+1] + in[i])/2;
}
```

Figure 3.15: fir_sse_float: original version

```
template <typename in_iter, typename out_iter>
void fir_sse_float_iterator(in_iter& in, out_iter& out,
                int len) {
        int i = 0;
        __m128 i0, i1, two = _mm_set_ps1(2.0);
        for (; i < len-5; i+=4) {
                i0 = _mm_load_ps(in + i);
                i1 = _mm_loadu_ps(in + (i+1));
                i0 = _mm_add_ps(i0, i1);
                i0 = _mm_div_ps(i0, two);
                _mm_store_ps(out + i, i0);
        }
        for (; i < len-1; ++i)
                out[i] = (in[i+1] + in[i])/2;
}
```

Figure 3.16: fir_sse_float: enhanced version

3.4.4 SSE: Double Precision

`fir_sse_double` performs the same operations as `fir`, but on 2 `double` elements at a time. I get fig. 3.16 by changing the signature of fig. 3.15.

```
void fir_sse_double(double *in, double *out, int len) {
        int i = 0;
        __m128d i0, i1;
        __m128d two = _mm_set_pd(2.0, 2.0);

        for (; i < len-3; i+=2) {
                i0 = _mm_load_pd(in + i);
                i1 = _mm_loadu_pd(in + (i+1));
                i0 = _mm_add_pd(i0, i1);
                i0 = _mm_div_pd(i0, two);
                _mm_store_pd(out + i, i0);
        }
        for (; i < len-1; ++i)
                out[i] = (in[i+1] + in[i])/2;
}
```

Figure 3.17: fir_sse_double: original version

```
template <typename in_iter, typename out_iter>
void fir_sse_double_iterator(in_iter& in, out_iter& out,
                int len) {
        int i = 0;
        __m128d i0, i1;
        __m128d two = _mm_set_pd(2.0, 2.0);

        for (; i < len-3; i+=2) {
                i0 = _mm_load_pd(in + i);
                i1 = _mm_loadu_pd(in + (i+1));
                i0 = _mm_add_pd(i0, i1);
                i0 = _mm_div_pd(i0, two);
                _mm_store_pd(out + i, i0);
        }
        for (; i < len-1; ++i)
                out[i] = (in[i+1] + in[i])/2;
}
```

Figure 3.18: fir_sse_double: enhanced version

Chapter 4

Evaluation

In the following sections I describe the testing environment, benchmark parameters, tested functions and the evaluation procedure I used to compare the different approaches to add access patterns to existing functions.

4.1 Test Procedure

I run the tests for each function several times over a range of input sizes filled with randomized values, to be able to check for differences resulting from the input size. Where possible, I picked the same test parameters as described in [6, p.89] to get comparable test conditions.

Input The input is a floating point array (`float` or `double`, depending on the function) with pseudorandom values in the range $[-1, 1]$. Before each run, the values are randomized again.

Input Sizes Where applicable, input sizes of the underlying array are

$$160000 * n, 1 \leq n \leq 10$$

The effective input sizes depend on the access pattern. Let s be the size of the underlying array. *Strided* access with distance d leads to the effective input size

$$s_{stride} = \frac{s}{d}$$

Blockstrided access whith distance d and block size b leads to the effective input size

$$s_{blockstride} = \left(\frac{s}{d}\right) * b + \left(\frac{s}{d}\right) \mod b$$

For DFT functions (`dft_512`, `dft_1048576`) the effective input size is fixed.

Access Patterns I implemented and compared the performance with the following access patterns:

- **strided** with distances 2,4 and 8
- **blockstrided** with (stride, blocksize) $\in \{(4,2),(8,4)\}$

Scaling operations are discussed in [6]; I left them out in my benchmark. Pre-/postscaling only adds one operation per access. The two patterns listed above should already provide a good basis for performance evaluation.

I confirmed the correctness of each function variant and iterator with manual test cases while implementing the benchmark. The outputs during the benchmark are not compared, because the input is randomized for each run.

4.2 Function Variants

The naming convention for the function variants I tested is:

- `linear`
  The original function applied to an array of the effective size of the acccess patterns used in the other variants. The values are not the same and this serves only as a rough reference.
- `iterator-<pattern>-static`
  The enhanced function used with an iterator where parameters are set via template arguments.
- `iterator-<pattern>-dynamic`
  The enhanced function used with an iterator where parameters are set on instatiation.
- `manual-<pattern>`
  The original function translated to use the desired access pattern. This is the method presented in [5, 6].
- `copy-<pattern>`
  The original function applied to a copy of the input according to the access pattern. The output is then copied to a linear array again.

4.3 Tested Functions

To be able to compare the `C++` approach presented here with the source translation approach, I mostly [1] (10 of 13, with 4 new functions added) adapted

[1] The spiral-generated functions fftfwd, sse_fftfwd and dct40 are missing, but two spiral-generated discrete Fourier transforms are present.

the tested functions from those in the original paper [6]. Table 4.1 is a list of these functions.

Table 4.1: tested functions

function	**description**	**source**
`contrast`	normalize range of values in an image	[5, 6]
`contrast_sse`	contrast using SSE	[5, 6]
`copy_dummy`	just copy the array	custom
`dft_512`	discrete Fourier transform with input size 1024	[2, 5, 6]
`dft_1048576`	dft with input size 2097152	[2, 5, 6]
`filter`	generic 2d convolution with a kernel of size 3x3	[5, 6]
`fir`	simple FIR filter	[5, 6]
`fir_sse_double`	`fir` using `SSE` and `double` type	custom
`fir_sse_float`	`fir` using `SSE` and `float` type	custom
`quicksort`	recursive in-place sorting	custom
`rec_mergesort`	recursive mergesort using a temporary array	[5, 6]
`rec_reduce`	recursive sum reduction of an array	[5, 6]
`reduce`	sum reduction of an array	[5, 6]
`scan`	prefix sum of a vector	[5, 6]
`sse_mvm`	(2x2) matrix vector multiplication using `SSE`: $y = x * M + y$	[5, 6]

4.4 Criteria

Compilers I use both `gcc` and `clang` as compilers, with the same flags: `"-Ofast -march=native -std=c++11"`. I included the results side by side in appendix B. They are in most cases comparable, with a few exceptions showing different strengths and weaknesses in optimization.

Statistical Interpretation The tests are run $n = 10$ times with input separately generated for each run. Then I compute the sample mean $\overline{x}$ of these results and the confidence interval assuming a t-distribution (because of the small number of samples) and using the sample variance s as suggested in [4, p.6, section 3.2.2].

$$s = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \overline{x})^2}{n-1}}$$

I chose a confidence level of 95% and used a precomputed table of t values [3] for the confidence interval $[c_1, c_2]$.

$$\delta = t_{0.95;n-1}\frac{s}{\sqrt{n}}$$
$$c_1 = \overline{x} - \delta$$
$$c_2 = \overline{x} + \delta$$

These confidence intervals are represented as a black lines on top of the colored histogram bars. Where these intervals overlap, we can assume a comparable performance. If they do not overlap, the performance of the function with the lower value can be assumed to be better with 95% confidence.

4.5 Setup

Platform I used `GNU/Linux` as the platform for this benchmark. It should be easy to port it to other platforms (UNIXes or Windows/Cygwin) and use other compilers. The benchmark consists of a collection of `C++` test programs and `bash` shell scripts. For each function, a separate binary is created. Output is saved as text files and processed to create plots.

Table 4.2: sofware and hardware used in the benchmark

```
 software         | version
------------------+-------------------------------------
 Linux            | 4.2.3 SMP PREEMPT x86_64, with ASLR
 gcc              | 4.9.3, with PIE patches
 clang            | 3.5.0 (tags/RELEASE_350/final)
 GNU bash         | 4.3.39(1)-release
 gnuplot          | 5.0 patchlevel 1
 GNU coreutils    | 8.23
 GNU grep         | 2.21
 GNU gawk         | 4.0.2
 texlive          | 2015

CPU: 2 cores, Intel(R) Core(TM)2 Duo CPU P8400 @ 2.26GHz
L1d cache:              32K
L1i cache:              32K
L2 cache:               3072K

frequency scaling governor: ondemand

memory: 4GB DDR2 @ 666MHz
```

Chapter 5

Results and Conclusion

I will now give an overview of the benchmark's results and discuss them. The main aspect is the performance for the implementation of access patterns via metaprogramming compared to the method of source translation previously discussed. I described the naming of the functions variants (`iterator`, `manual` etc.) in section 4.2.

5.1 Interpretation

Legend Table 5.1 provides a quick overview of the findings for each function. The meanings of the columns are as follows:

- `iterator/manual, static iterator`
  The overall performance of `iterator` compared to `manual`, and the relative performance of the `static iterator`, which is often much faster. **comparable** means the performance can be seen at least as identical.
- `copy`
  The relative speed of `copy` when compared to the other approaches. **slow** means it is alwas better to avoid copying; **fast** means copying is always more efficient.
- `gcc/clang`
  If one of the compilers used offers an advantage for the `iterator` approach, it is listed here.
- **plot**
  A link to the page containing the plots for this function.
- **ref.**
  Link to a discussion of this result.

function	iterator/manual, static iterator	copy	gcc/clang	**plot**	**ref.**
contrast	comparable, faster	slow	gcc	B.1	5.1.2
contrast_sse	comparable, faster	slow	-	B.2	-
copy_dummy	comparable, faster	slow	-	B.3	-
dft_1048576	mixed	slow	-	B.4	5.1.2
dft_512	comparable, faster	slow	-	B.5	-
filter	comparable, faster	-	gcc	B.6	5.1.2
fir	comparable, faster	slow	-	B.7	-
fir_sse_double	comparable, faster	slow	-	B.8	-
fir_sse_float	comparable, faster	slow	-	B.9	-
quicksort	comparable, faster	fast	-	B.10	5.1.2
rec_mergesort	comparable, faster	fast	-	B.11	5.1.2
rec_reduce	faster	slow	-	B.12	-
reduce	comparable, faster	slow	-	B.13	-
scan	comparable, faster	slow	-	B.14	-
sse_mvm	faster	slow	-	B.15	-

Table 5.1: overview of results

5.1.1 Summary

The results are in most cases as expected: in the wide majority of test cases, using an `iterator` to implement the access pattern and keeping the function body unchanged leads to the same performance as translating the source code directly to implement the access pattern inside the function's body. Using the `static` version of the iterator improves performance even more, most dramatically for complex patterns like *blockstriding*. Functions with many accesses (loops) still perform better when the input is copied before processing: these are represented by the two sorting functions `quicksort` and `rec_mergesort`.

5.1.2 Details

If the results shown in table 5.1 need more explanation, I included it here.

contrast

Blockstriding in `iterator` or `manual` performs better when compiled with `gcc -Ofast` than with `clang -Ofast`.

dft_1048576

This is the only case where the results are the opposite of my expectations: `iterator` is mostly much slower than `manual`. I compared the outputs of the variants and found no anomalies.

One explanation might be that the compiler omits inlining in this case because the function is very long (> 1000 lines). The other, smaller DFT function `dft_512` performs as expected. Compilers use heuristics to decide whether to inline small functions, and it is possible that the accesses are not inlined in this function, even though the methods are defined inside the class block. It may be possible to change this behavior with compiler attributes.

filter

While using the `static iterator` is still more efficient (with `gcc`) than `copying`, the `dynamic iterator` and `manual` are too slow when performing complex accesses like *blockstriding*.

quicksort, rec_mergesort

In sorting algorithms, the number of accesses introduces a significant penalty when an index has to be calculated for each access. In addition, accessed locations are often far apart, which increases cache pressure. Copying the input and output to/from a contiguous array is always the best choice here.

5.2 Limitations/Future Work

The results I presented in the previous section showed that the objective of abstracting array access with the same performance as plain source modification was reached. I will now look at some of the limitations of and possible improvements for the metaprogramming approach and the test procedure.

5.2.1 Functions

The set of test function is very limited and abstract. It would be helpful to see an implementation of this approach in real applications like image/video or audio processing. There are also still functions missing in the comparison with [6], but since these are both Spiral-generated Fast Fourier or Discrete Cosine Transforms, we can expect similar results to the already included DFTs.

5.2.2 Iterator Operations and Chaining

Pre/postprocessing operations like scaling are missing in my implementation, but since they introduce a fixed number of arithmetic operations for each access, we can expect them not to influence the results of the comparisons much. They can easily be added to an existing or new iterator.

It may be useful to write iterators that can abstract access to other iterators, so that they can be chained together. One would have to test if automatic inlining still works as well in this case.

5.2.3 Recursion, Pointers and Blockstriding

In recursive functions, the starting point of the array has to stay the same with *blockstriding*, as discussed in section 2.1.3. It should be possible to overcome this limitation by storing an offset from the first element in the iterator returned by `operator+` and use it for index calculation, so that block boundaries are not moved in subsequent calls to the function. This would of course slow the access down, so the performance of this code would have to be investigated. Iterative functions should be preferred to recursive ones because of the lower overhead.

If recursive calls of this sort are to be prevented and detected at compile time, one could make `operator+` return an `iterator` that lacks this operator so that recursion is not supported and violations lead to compiler errors.

5.2.4 SSE code duplication

```
// function for "static" stride
template <int stride>
void _mm_store_ps(CStrided<float, stride> access, __m128 x) {
    /* code A */
}

// function for "dynamic" stride
void _mm_store_ps(Strided<float> access, __m128 x) {
    /* copy of code A */
}
```

Figure 5.1: duplicated SSE code for different iterators

```
// does not work (yet)
template <typename Iter>
void _mm_store_ps(Iter access, __m128 x) {
    /* code A */
}
```

Figure 5.2: desired combination of SSE functions

The two versions of *striding* iterators I implemented need two sets of `SSE` functions (see fig. 5.1), because templating does not work when the argument counts are not the same. It should be possible to make a generalized function along the lines of fig. 5.2 work (using helper structures/templates and type traits) and reduce code duplication, if it is an issue.

5.3 Conclusion

The results of the benchmark show that using `C++` metaprogramming to include array access patterns in numerical functions preserves the performance achieved when including these patterns by translating `C` source code, and even enables to improve on it. When the access pattern's parameters are set at compile time using template arguments, I can considerably improve the performance with more complex access patterns (e.g. *blockstriding*).

A side effect of this approach is the expansion of the supported language to `C++`. I do not have to rely on a separate parser library and tool, since only a single line, the function signature, has to be modified once. This can quickly be done by hand. The implementation of specific patterns is cleanly separated from user code and can be changed without touching the user code. The iterators can be distributed as a library/header.

If the platform/compiler supports `C++`, the results are in favor of this ap-

proach. It may be necessary to try compiler-specific attributes in the iterator with very big functions to force inlining. If there is only support for `C`, translating the source by hand or with a custom tool is the remaining solution.

Functions performing many repeated and scattered accesses, like sorting, should still be used with a contiguous copy of the data, if performance is a concern. Else the overhead will be noticeable.

The two compilers used produce binaries that have comparable performance with the new approach, with `gcc` having a little edge when compared to `clang` in some rare cases with the versions used in the benchmarks (table 4.2).

The comparison was worth it, because it supported the hypothesis that `C++` metaprogramming for the abstraction of array accesses leads to a good performance. It should be of practical use for efficient computation and reduce the effort because the abstractions can easily be reused and adapted without mixing array access and access pattern code. See section 1.4 for an overview of advantages for the programmer.

Appendix A

Manual

The source of the benchmark program and data presented here can be cloned from:
`https://gitlab-ps.informatik.uni-tuebingen.de/SArslanbek/array-interface-benchmark.git`

It may be useful to test newer CPUs and other compilers.

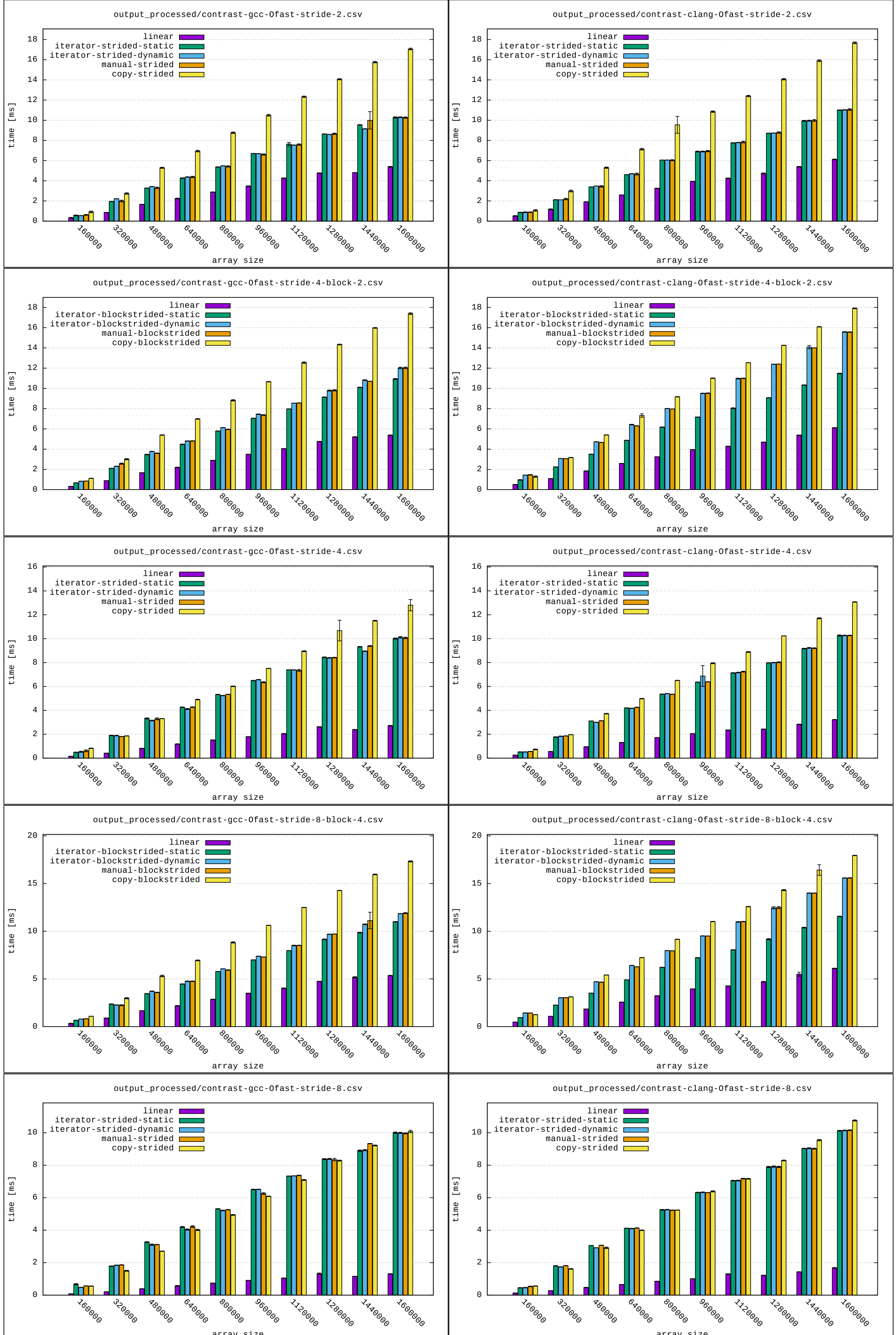
See table 4.2 for a list of dependencies to run the benchmark.

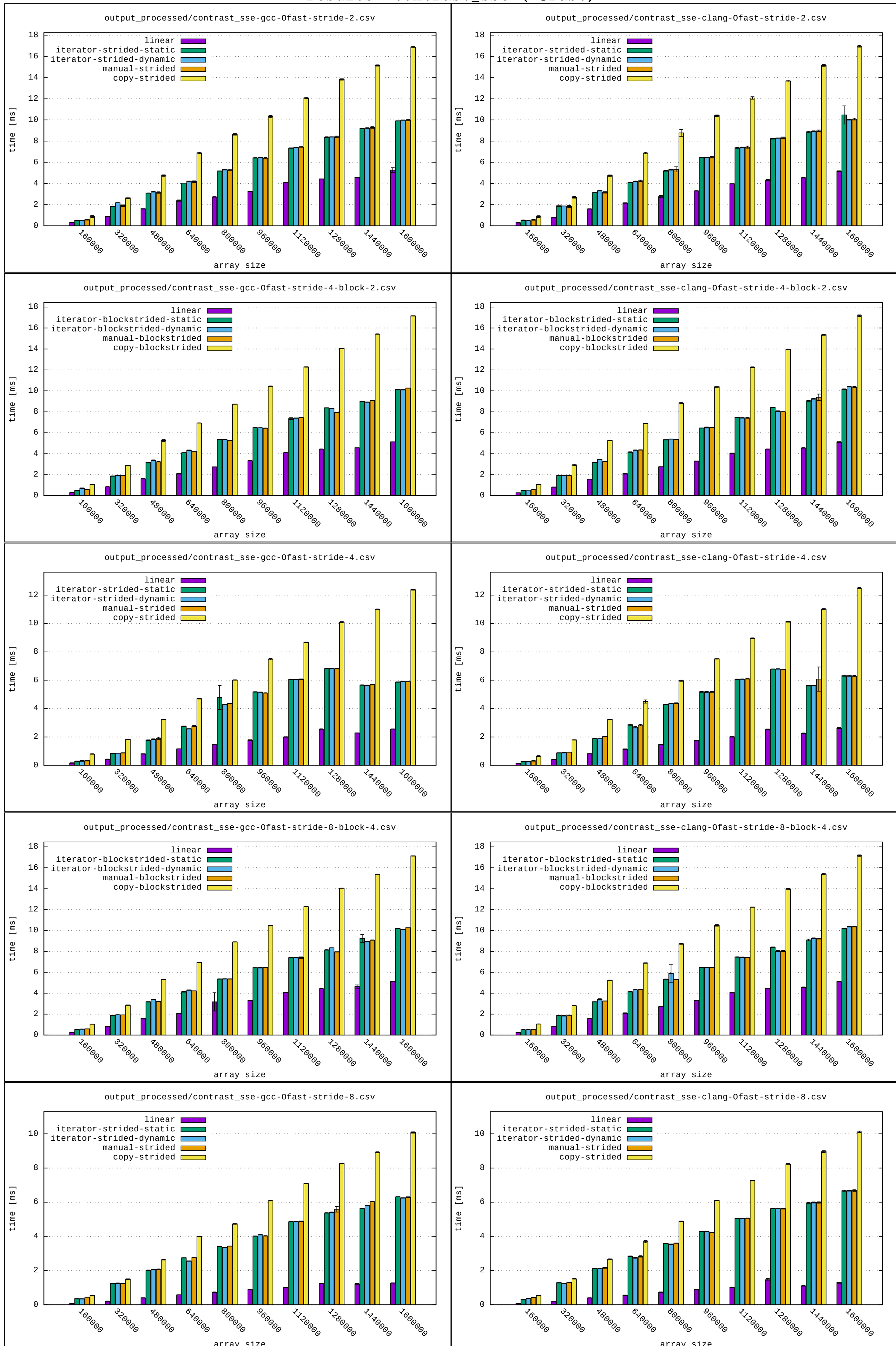
The `README` file provides an overview of the structure and a quick guide.

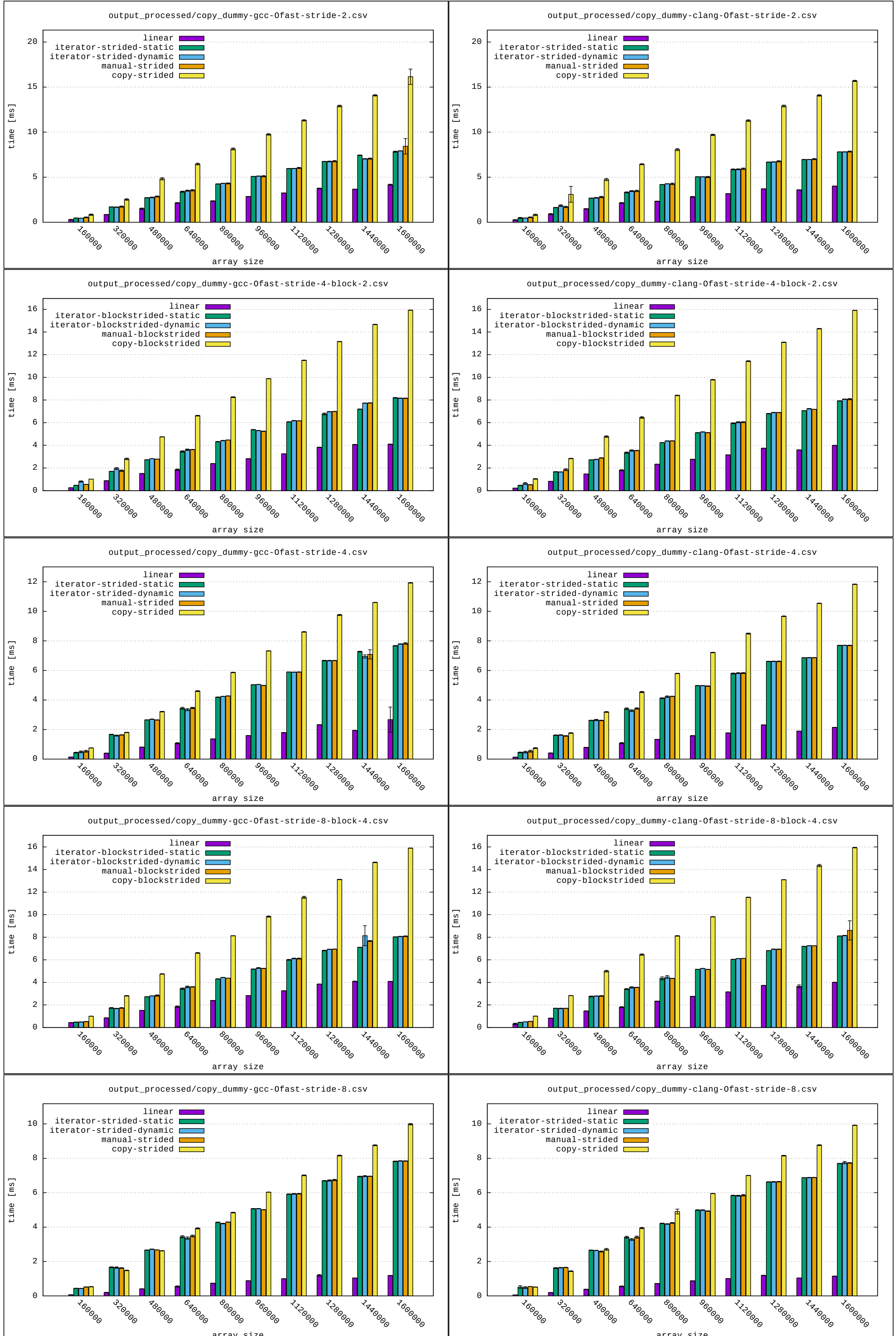
Appendix B

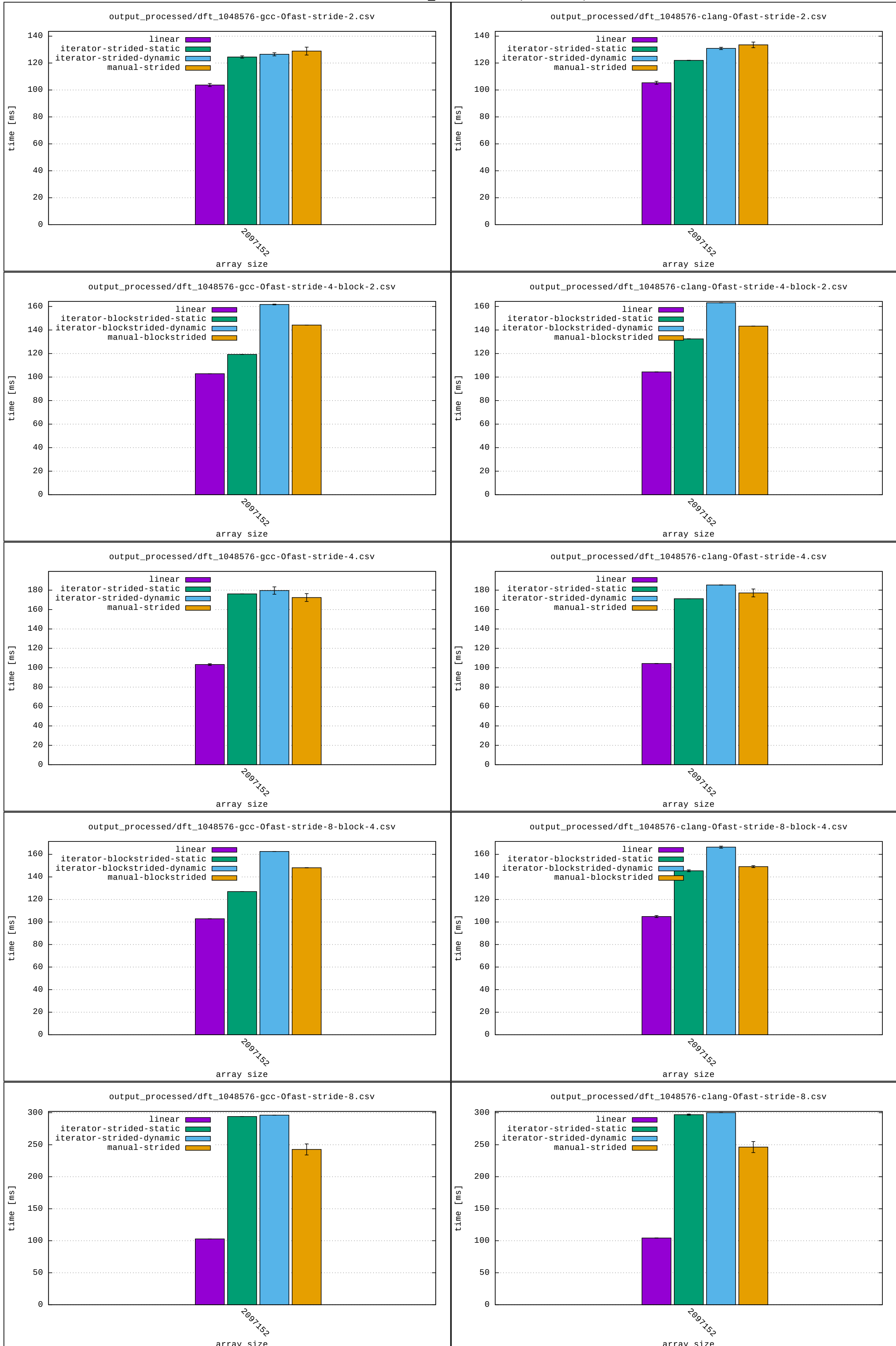
Complete Plots

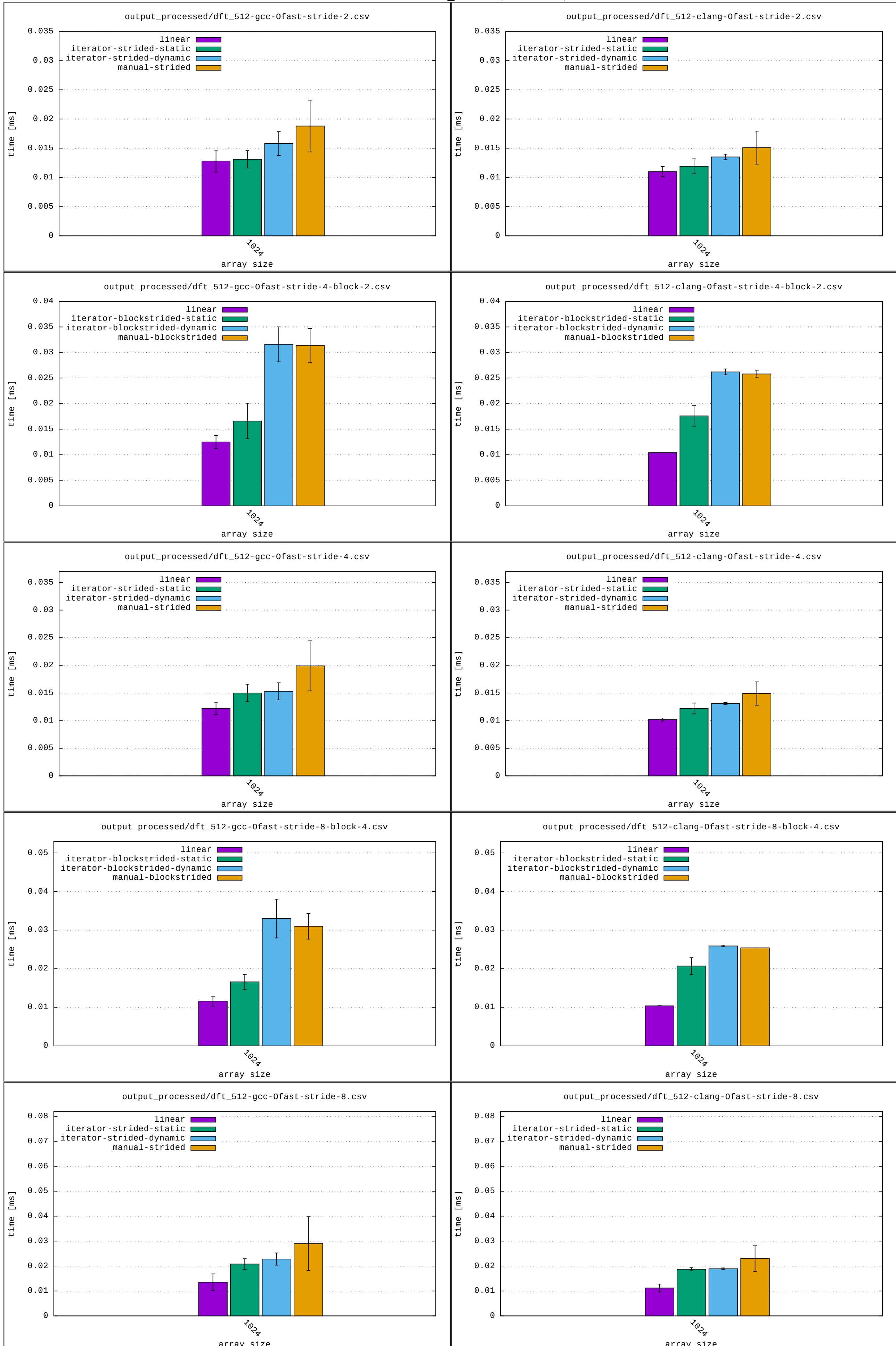
The following pages contain the full side-by-side plots for each tested function. In the left-hand column are the results of the benchmark compiled by `gcc`, the corresponding results of the binaries created by `clang` are on the right-hand side.

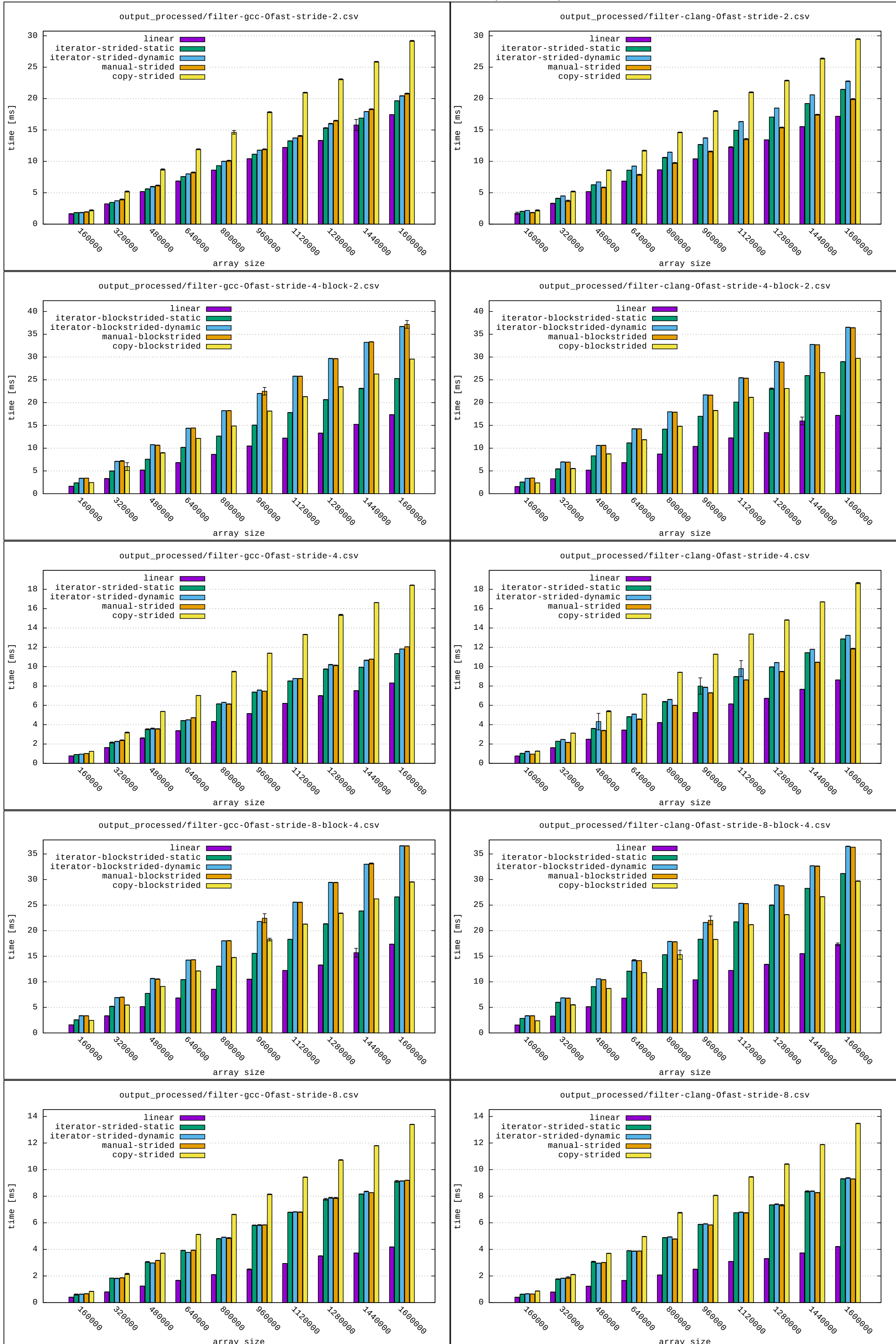

results: contrast (-Ofast)
output_processed/contrast-gcc-Ofast-stride-2.csv
output_processed/contrast-clang-Ofast-stride-2.csv
linear
iterator-strided-static
iterator-strided-dynamic
manual-strided
copy-strided
output_processed/contrast-gcc-Ofast-stride-4-block-2.csv
output_processed/contrast-clang-Ofast-stride-4-block-2.csv
linear
iterator-blockstrided-static
iterator-blockstrided-dynamic
manual-blockstrided
copy-blockstrided
output_processed/contrast-gcc-Ofast-stride-4.csv
output_processed/contrast-clang-Ofast-stride-4.csv
output_processed/contrast-gcc-Ofast-stride-8-block-4.csv
output_processed/contrast-clang-Ofast-stride-8-block-4.csv
output_processed/contrast-gcc-Ofast-stride-8.csv
output_processed/contrast-clang-Ofast-stride-8.csv
time [ms]
array size
160000
320000
480000
640000
800000
960000
1120000
1280000
1440000
1600000

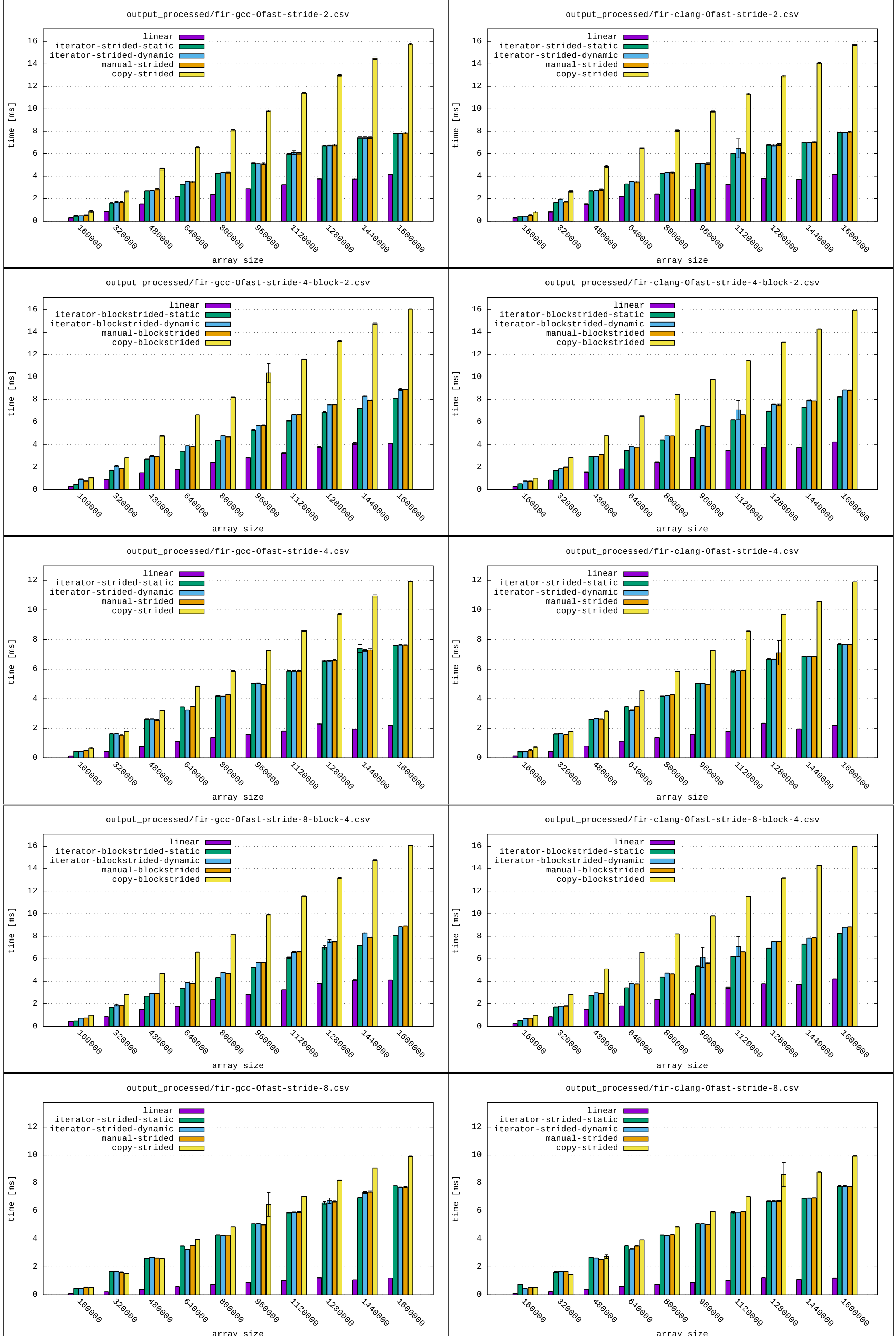

results: contrast_sse (-Ofast)
output_processed/contrast_sse-gcc-Ofast-stride-2.csv
output_processed/contrast_sse-clang-Ofast-stride-2.csv
output_processed/contrast_sse-gcc-Ofast-stride-4-block-2.csv
output_processed/contrast_sse-clang-Ofast-stride-4-block-2.csv
output_processed/contrast_sse-gcc-Ofast-stride-4.csv
output_processed/contrast_sse-clang-Ofast-stride-4.csv
output_processed/contrast_sse-gcc-Ofast-stride-8-block-4.csv
output_processed/contrast_sse-clang-Ofast-stride-8-block-4.csv
output_processed/contrast_sse-gcc-Ofast-stride-8.csv
output_processed/contrast_sse-clang-Ofast-stride-8.csv
linear
iterator-strided-static
iterator-strided-dynamic
manual-strided
copy-strided
iterator-blockstrided-static
iterator-blockstrided-dynamic
manual-blockstrided
copy-blockstrided
time [ms]
array size
160000
320000
480000
640000
800000
960000
1120000
1280000
1440000
1600000

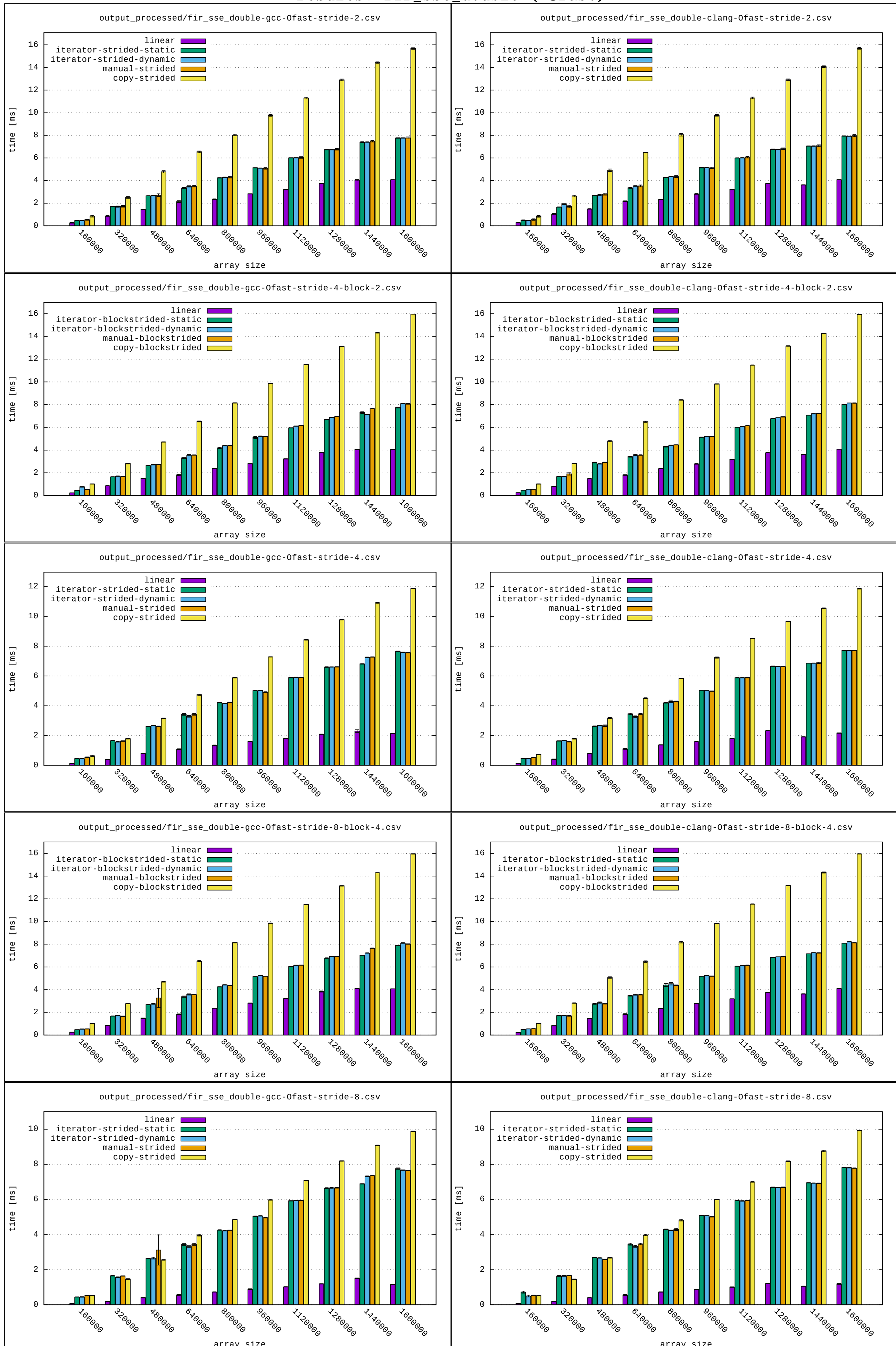

results: copy_dummy (-Ofast)
output_processed/copy_dummy-gcc-Ofast-stride-2.csv
output_processed/copy_dummy-clang-Ofast-stride-2.csv
output_processed/copy_dummy-gcc-Ofast-stride-4-block-2.csv
output_processed/copy_dummy-clang-Ofast-stride-4-block-2.csv
output_processed/copy_dummy-gcc-Ofast-stride-4.csv
output_processed/copy_dummy-clang-Ofast-stride-4.csv
output_processed/copy_dummy-gcc-Ofast-stride-8-block-4.csv
output_processed/copy_dummy-clang-Ofast-stride-8-block-4.csv
output_processed/copy_dummy-gcc-Ofast-stride-8.csv
output_processed/copy_dummy-clang-Ofast-stride-8.csv
linear
iterator-strided-static
iterator-strided-dynamic
manual-strided
copy-strided
iterator-blockstrided-static
iterator-blockstrided-dynamic
manual-blockstrided
copy-blockstrided
time [ms]
array size
160000
320000
480000
640000
800000
960000
1120000
1280000
1440000
1600000

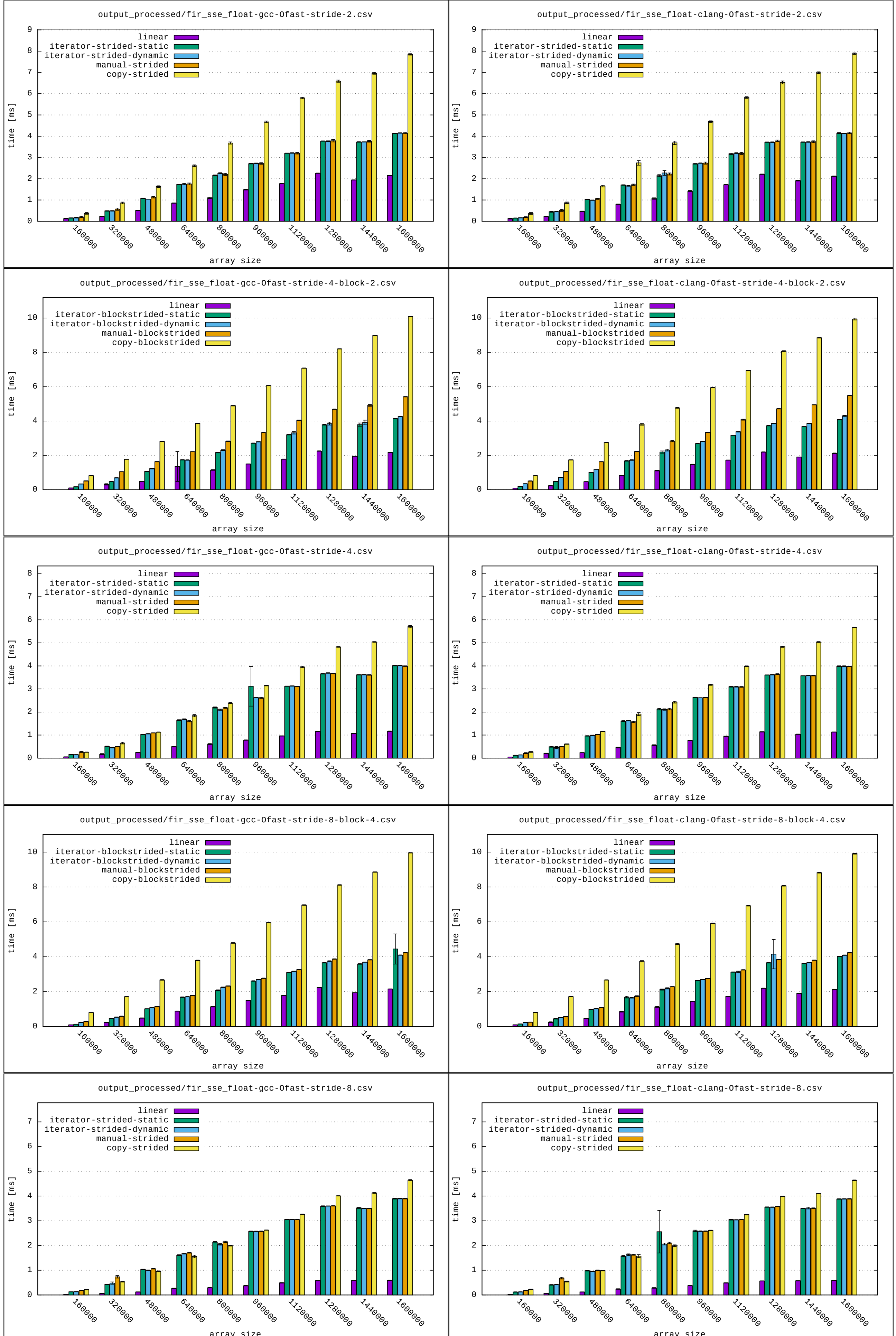

results: dft_1048576 (-Ofast)
output_processed/dft_1048576-gcc-Ofast-stride-2.csv
output_processed/dft_1048576-clang-Ofast-stride-2.csv
output_processed/dft_1048576-gcc-Ofast-stride-4-block-2.csv
output_processed/dft_1048576-clang-Ofast-stride-4-block-2.csv
output_processed/dft_1048576-gcc-Ofast-stride-4.csv
output_processed/dft_1048576-clang-Ofast-stride-4.csv
output_processed/dft_1048576-gcc-Ofast-stride-8-block-4.csv
output_processed/dft_1048576-clang-Ofast-stride-8-block-4.csv
output_processed/dft_1048576-gcc-Ofast-stride-8.csv
output_processed/dft_1048576-clang-Ofast-stride-8.csv
linear
iterator-strided-static
iterator-strided-dynamic
manual-strided
iterator-blockstrided-static
iterator-blockstrided-dynamic
manual-blockstrided
time [ms]
array size
2097152

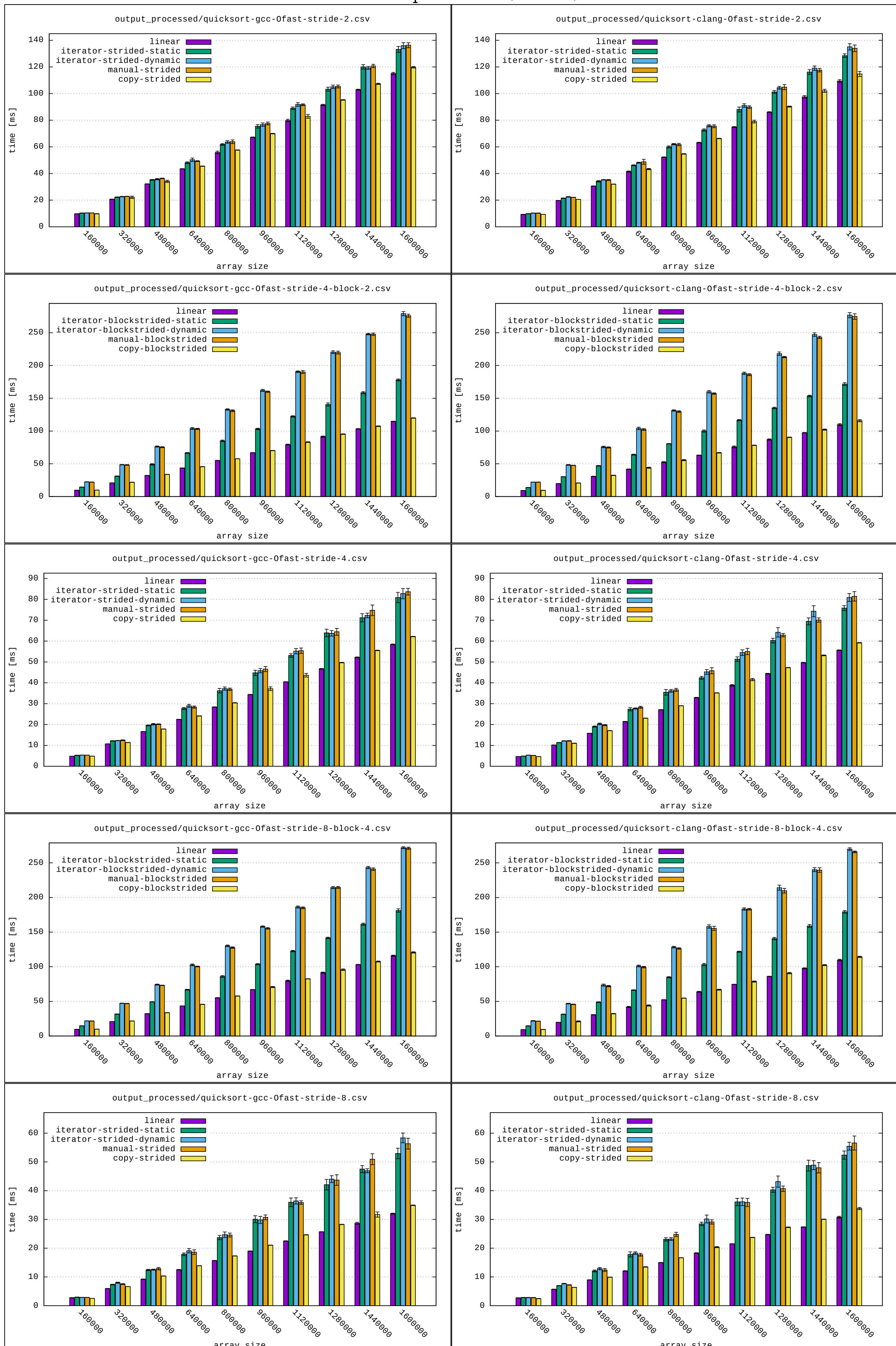

results: dft_512 (-Ofast)
output_processed/dft_512-gcc-Ofast-stride-2.csv
output_processed/dft_512-clang-Ofast-stride-2.csv
output_processed/dft_512-gcc-Ofast-stride-4-block-2.csv
output_processed/dft_512-clang-Ofast-stride-4-block-2.csv
output_processed/dft_512-gcc-Ofast-stride-4.csv
output_processed/dft_512-clang-Ofast-stride-4.csv
output_processed/dft_512-gcc-Ofast-stride-8-block-4.csv
output_processed/dft_512-clang-Ofast-stride-8-block-4.csv
output_processed/dft_512-gcc-Ofast-stride-8.csv
output_processed/dft_512-clang-Ofast-stride-8.csv
linear
iterator-strided-static
iterator-strided-dynamic
manual-strided
iterator-blockstrided-static
iterator-blockstrided-dynamic
manual-blockstrided
time [ms]
array size
1024

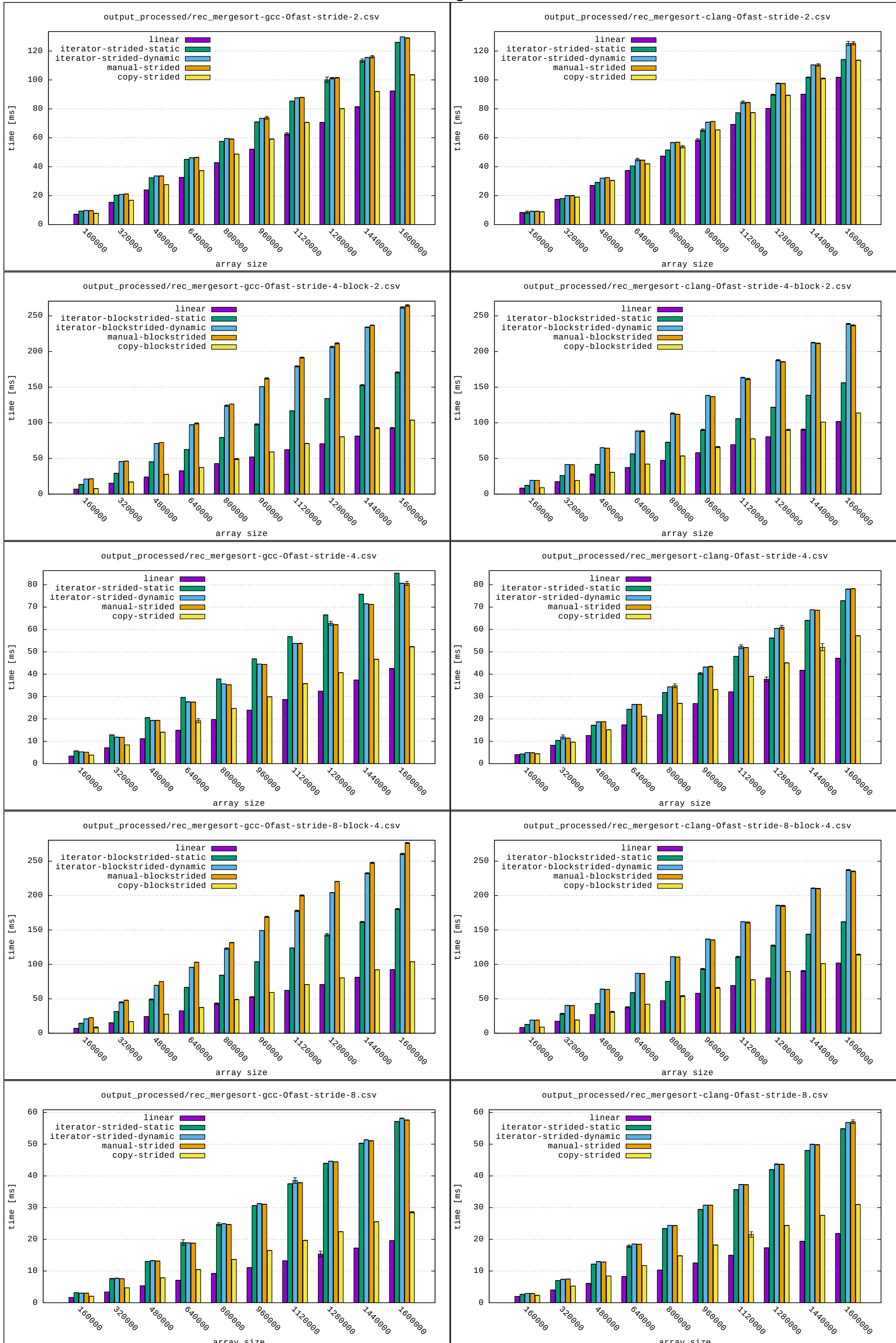

results: filter (-Ofast)
output_processed/filter-gcc-Ofast-stride-2.csv
output_processed/filter-clang-Ofast-stride-2.csv
output_processed/filter-gcc-Ofast-stride-4-block-2.csv
output_processed/filter-clang-Ofast-stride-4-block-2.csv
output_processed/filter-gcc-Ofast-stride-4.csv
output_processed/filter-clang-Ofast-stride-4.csv
output_processed/filter-gcc-Ofast-stride-8-block-4.csv
output_processed/filter-clang-Ofast-stride-8-block-4.csv
output_processed/filter-gcc-Ofast-stride-8.csv
output_processed/filter-clang-Ofast-stride-8.csv
linear
iterator-strided-static
iterator-strided-dynamic
manual-strided
copy-strided
iterator-blockstrided-static
iterator-blockstrided-dynamic
manual-blockstrided
copy-blockstrided
time [ms]
array size
160000
320000
480000
640000
800000
960000
1120000
1280000
1440000
1600000

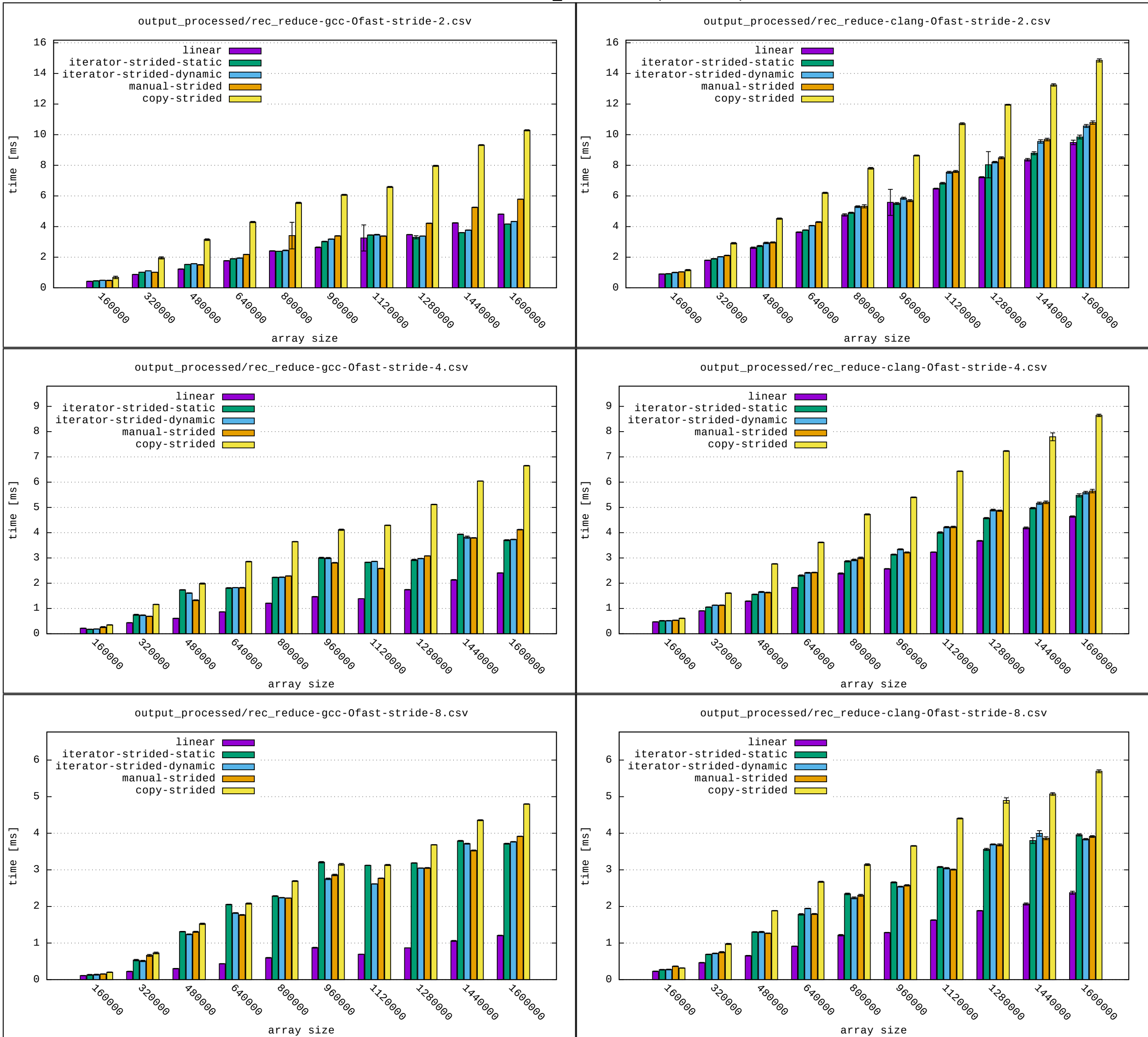

results: fir (-Ofast)
output_processed/fir-gcc-Ofast-stride-2.csv
output_processed/fir-clang-Ofast-stride-2.csv
output_processed/fir-gcc-Ofast-stride-4-block-2.csv
output_processed/fir-clang-Ofast-stride-4-block-2.csv
output_processed/fir-gcc-Ofast-stride-4.csv
output_processed/fir-clang-Ofast-stride-4.csv
output_processed/fir-gcc-Ofast-stride-8-block-4.csv
output_processed/fir-clang-Ofast-stride-8-block-4.csv
output_processed/fir-gcc-Ofast-stride-8.csv
output_processed/fir-clang-Ofast-stride-8.csv
linear
iterator-strided-static
iterator-strided-dynamic
manual-strided
copy-strided
iterator-blockstrided-static
iterator-blockstrided-dynamic
manual-blockstrided
copy-blockstrided
time [ms]
array size
160000
320000
480000
640000
800000
960000
1120000
1280000
1440000
1600000


results: fir_sse_double (-Ofast)
output_processed/fir_sse_double-gcc-Ofast-stride-2.csv
output_processed/fir_sse_double-clang-Ofast-stride-2.csv
linear
iterator-strided-static
iterator-strided-dynamic
manual-strided
copy-strided
output_processed/fir_sse_double-gcc-Ofast-stride-4-block-2.csv
output_processed/fir_sse_double-clang-Ofast-stride-4-block-2.csv
iterator-blockstrided-static
iterator-blockstrided-dynamic
manual-blockstrided
copy-blockstrided
output_processed/fir_sse_double-gcc-Ofast-stride-4.csv
output_processed/fir_sse_double-clang-Ofast-stride-4.csv
output_processed/fir_sse_double-gcc-Ofast-stride-8-block-4.csv
output_processed/fir_sse_double-clang-Ofast-stride-8-block-4.csv
output_processed/fir_sse_double-gcc-Ofast-stride-8.csv
output_processed/fir_sse_double-clang-Ofast-stride-8.csv
time [ms]
array size
160000
320000
480000
640000
800000
960000
1120000
1280000
1440000
1600000

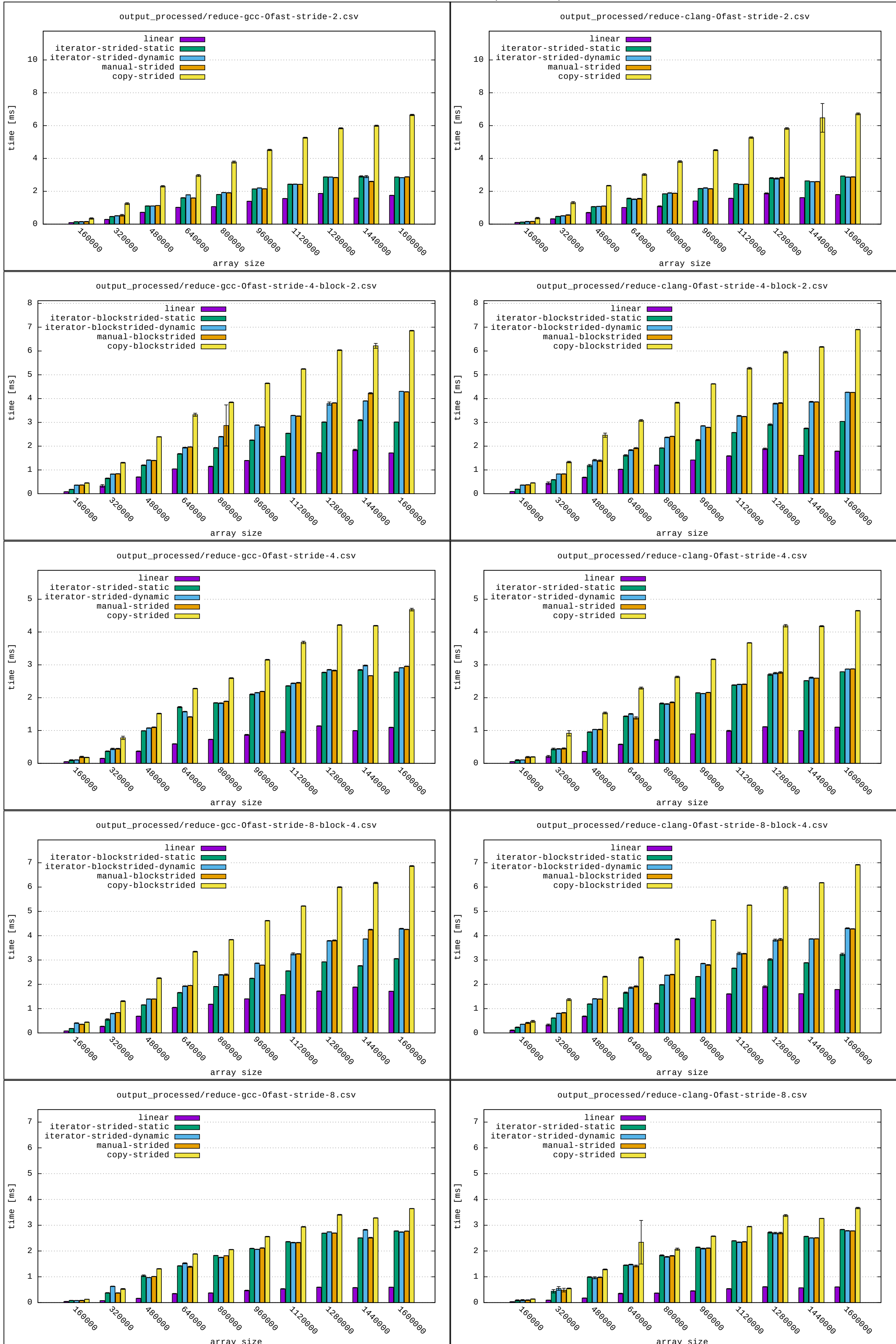

results: fir_sse_float (-Ofast)
output_processed/fir_sse_float-gcc-Ofast-stride-2.csv
output_processed/fir_sse_float-clang-Ofast-stride-2.csv
output_processed/fir_sse_float-gcc-Ofast-stride-4-block-2.csv
output_processed/fir_sse_float-clang-Ofast-stride-4-block-2.csv
output_processed/fir_sse_float-gcc-Ofast-stride-4.csv
output_processed/fir_sse_float-clang-Ofast-stride-4.csv
output_processed/fir_sse_float-gcc-Ofast-stride-8-block-4.csv
output_processed/fir_sse_float-clang-Ofast-stride-8-block-4.csv
output_processed/fir_sse_float-gcc-Ofast-stride-8.csv
output_processed/fir_sse_float-clang-Ofast-stride-8.csv
linear
iterator-strided-static
iterator-strided-dynamic
manual-strided
copy-strided
iterator-blockstrided-static
iterator-blockstrided-dynamic
manual-blockstrided
copy-blockstrided
time [ms]
array size

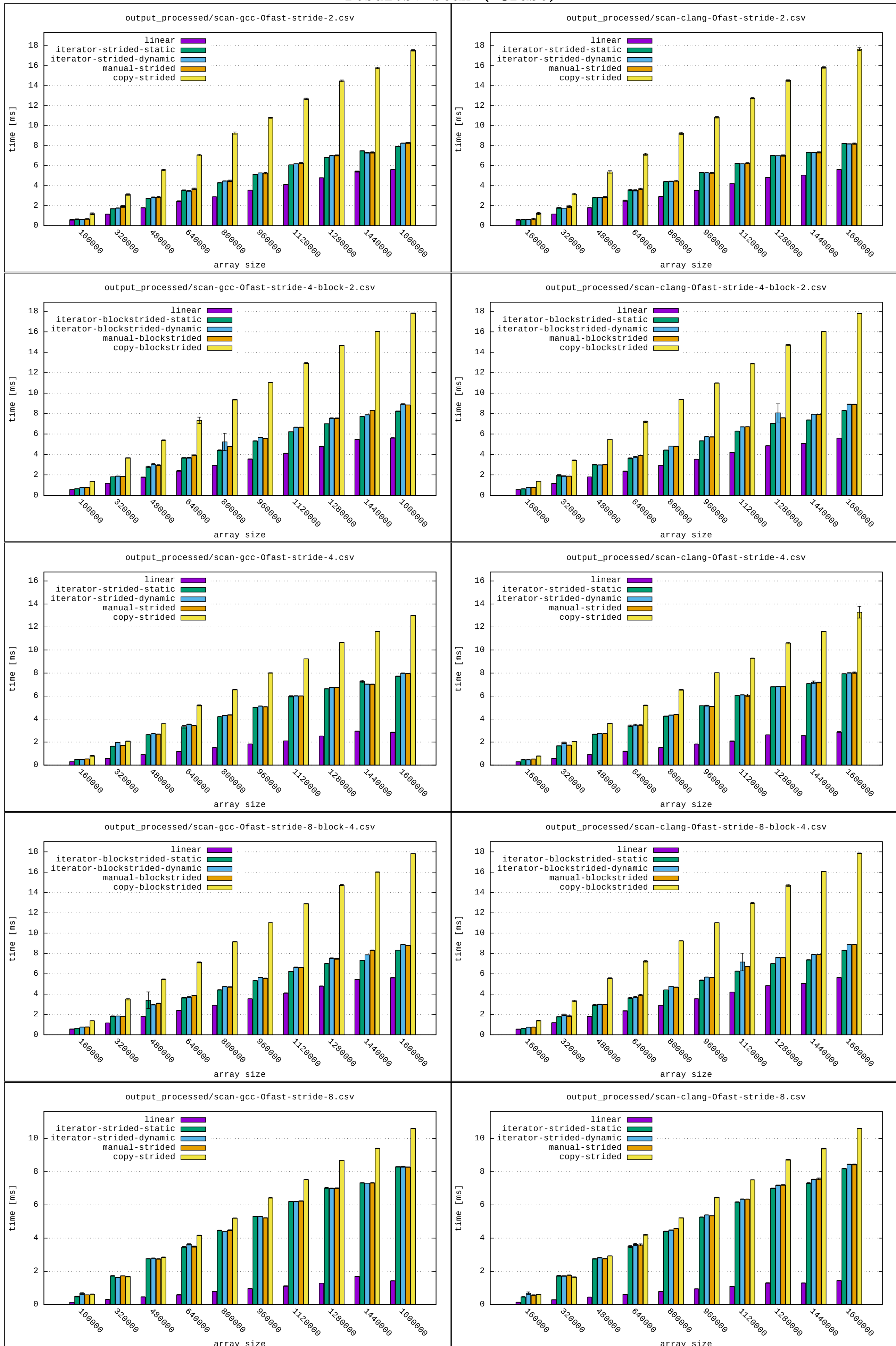

results: quicksort (-Ofast)
output_processed/quicksort-gcc-Ofast-stride-2.csv
output_processed/quicksort-clang-Ofast-stride-2.csv
output_processed/quicksort-gcc-Ofast-stride-4-block-2.csv
output_processed/quicksort-clang-Ofast-stride-4-block-2.csv
output_processed/quicksort-gcc-Ofast-stride-4.csv
output_processed/quicksort-clang-Ofast-stride-4.csv
output_processed/quicksort-gcc-Ofast-stride-8-block-4.csv
output_processed/quicksort-clang-Ofast-stride-8-block-4.csv
output_processed/quicksort-gcc-Ofast-stride-8.csv
output_processed/quicksort-clang-Ofast-stride-8.csv
linear
iterator-strided-static
iterator-strided-dynamic
manual-strided
copy-strided
iterator-blockstrided-static
iterator-blockstrided-dynamic
manual-blockstrided
copy-blockstrided
time [ms]
array size
160000
320000
480000
640000
800000
960000
1120000
1280000
1440000
1600000


results: rec_mergesort (-Ofast)
output_processed/rec_mergesort-gcc-Ofast-stride-2.csv
output_processed/rec_mergesort-clang-Ofast-stride-2.csv
output_processed/rec_mergesort-gcc-Ofast-stride-4-block-2.csv
output_processed/rec_mergesort-clang-Ofast-stride-4-block-2.csv
output_processed/rec_mergesort-gcc-Ofast-stride-4.csv
output_processed/rec_mergesort-clang-Ofast-stride-4.csv
output_processed/rec_mergesort-gcc-Ofast-stride-8-block-4.csv
output_processed/rec_mergesort-clang-Ofast-stride-8-block-4.csv
output_processed/rec_mergesort-gcc-Ofast-stride-8.csv
output_processed/rec_mergesort-clang-Ofast-stride-8.csv
linear
iterator-strided-static
iterator-strided-dynamic
manual-strided
copy-strided
iterator-blockstrided-static
iterator-blockstrided-dynamic
manual-blockstrided
copy-blockstrided
time [ms]
array size

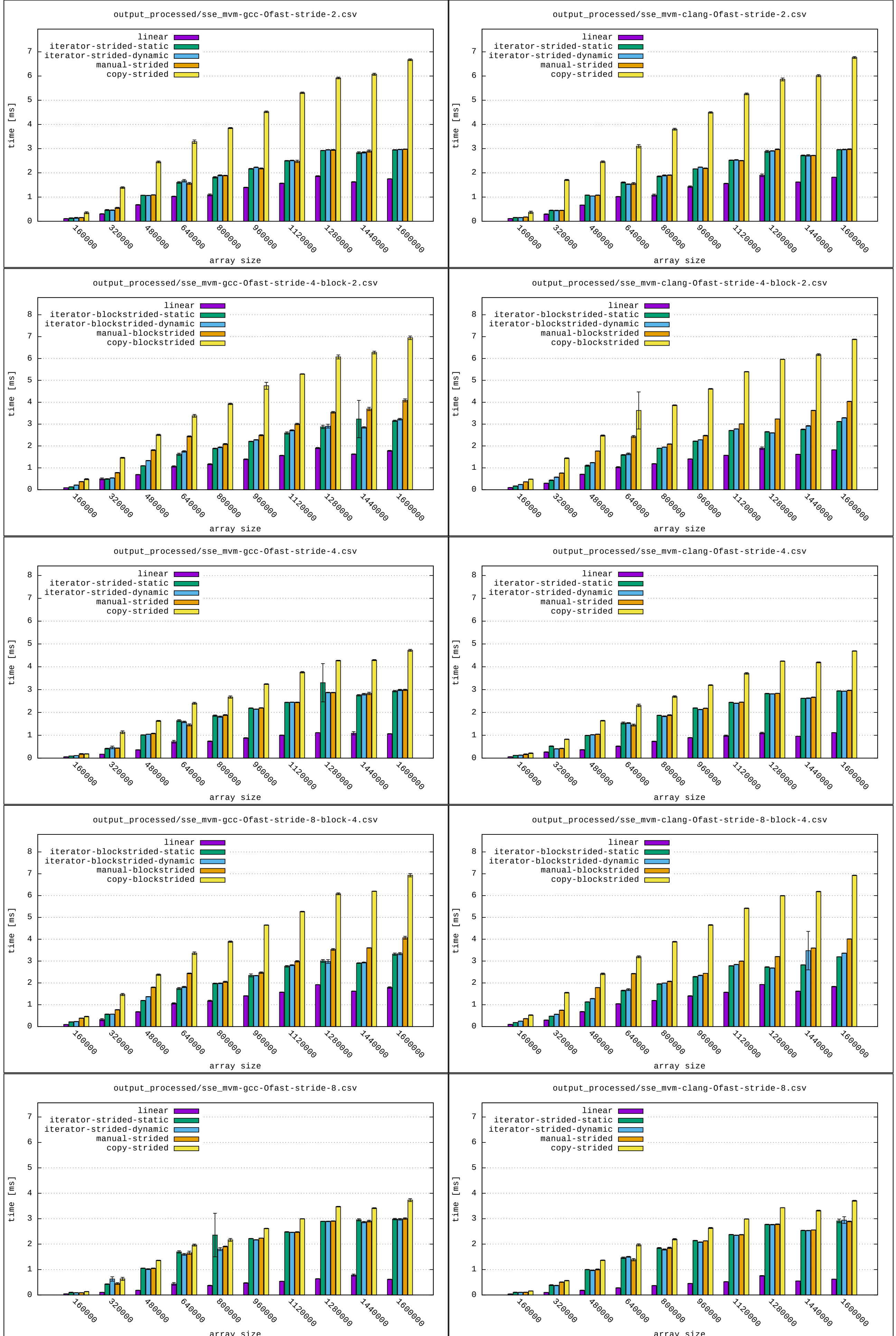

results: rec_reduce (-Ofast)
output_processed/rec_reduce-gcc-Ofast-stride-2.csv
output_processed/rec_reduce-clang-Ofast-stride-2.csv
output_processed/rec_reduce-gcc-Ofast-stride-4.csv
output_processed/rec_reduce-clang-Ofast-stride-4.csv
output_processed/rec_reduce-gcc-Ofast-stride-8.csv
output_processed/rec_reduce-clang-Ofast-stride-8.csv
linear
iterator-strided-static
iterator-strided-dynamic
manual-strided
copy-strided
time [ms]
array size
160000
320000
480000
640000
800000
960000
1120000
1280000
1440000
1600000

results: reduce (-Ofast)




results: scan (-Ofast)
output_processed/scan-gcc-Ofast-stride-2.csv
output_processed/scan-clang-Ofast-stride-2.csv
linear
iterator-strided-static
iterator-strided-dynamic
manual-strided
copy-strided
output_processed/scan-gcc-Ofast-stride-4-block-2.csv
output_processed/scan-clang-Ofast-stride-4-block-2.csv
linear
iterator-blockstrided-static
iterator-blockstrided-dynamic
manual-blockstrided
copy-blockstrided
output_processed/scan-gcc-Ofast-stride-4.csv
output_processed/scan-clang-Ofast-stride-4.csv
output_processed/scan-gcc-Ofast-stride-8-block-4.csv
output_processed/scan-clang-Ofast-stride-8-block-4.csv
output_processed/scan-gcc-Ofast-stride-8.csv
output_processed/scan-clang-Ofast-stride-8.csv
time [ms]
array size
160000
320000
480000
640000
800000
960000
1120000
1280000
1440000
1600000

results: sse_mvm (-Ofast)



List of Tables

4.1 tested functions 28
4.2 sofware and hardware used in the benchmark 29

5.1 overview of results 31

List of Figures

1.1 a function with linear access 1
1.2 applying striding to the original function via copying 1
1.3 original image 2
1.4 image with manipulated red color channel 2
1.5 including the strided access pattern in a function 3
1.6 using the transformed function 3
1.7 illustration of source-to-source translation usage 3
1.8 illustration of iterator template usage 4
1.9 enhanced function using an abstract array interface 4
1.10 selected results: summation of an array with stride distance 2 . 5
1.11 selected results: recursive mergesort 5

2.1 including the blockstrided access pattern in a function 8
2.2 a recursive summation function using pointers 8
2.3 a recursive summation function with strided access using pointers 9
2.4 a recursive summation function with strided access using pointers 9
2.5 single precision packed load instruction 10
2.6 single precision strided load instruction 11
2.7 single precision packed store instruction 11
2.8 single precision strided store instruction 11
2.9 double precision packed load instruction 11
2.10 double precision load instructions for strided access 11
2.11 double precision packed store instruction 11

2.12 double precision store instructions for strided access 11

3.1 striding iterator, with parameter defined at runtime 13
3.2 striding iterator, with parameter defined at compile time 14
3.3 blockstriding iterator, with parameters defined at runtime . . . 14
3.4 blockstriding iterator, with parameters defined at compile time . 15
3.5 strided SSE double precision load/store functions 16
3.6 strided SSE single precision load/store functions 17
3.7 aligned blockstriding SSE double precision load/store functions . 18
3.8 aligned blockstriding SSE single precision load/store functions . 19
3.9 1D alignment checks for single and double precision 20
3.10 2D alignment checks for single and double precision 21
3.11 fir: original version 22
3.12 fir: enhanced version 22
3.13 rec_reduce: original version 23
3.14 rec_reduce: enhanced version 23
3.15 fir_sse_float: original version 24
3.16 fir_sse_float: enhanced version 24
3.17 fir_sse_double: original version 25
3.18 fir_sse_double: enhanced version 25

5.1 duplicated SSE code for different iterators 34
5.2 desired combination of SSE functions 34

B.1 contrast: results 38
B.2 contrast_sse: results 39
B.3 copy_dummy: results 40
B.4 dft_1048576: results 41
B.5 dft_512: results 42
B.6 filter: results 43
B.7 fir: results 44

B.8 fir_sse_double: results 45
B.9 fir_sse_float: results 46
B.10 quicksort: results 47
B.11 mergesort: results 48
B.12 rec_reduce: results 49
B.13 rec_reduce: results 50
B.14 scan: results 51
B.15 sse_mvm: results 52

References

[1] Intel intrinsics guide, interactive website. https://software.intel.com/sites/landingpage/IntrinsicsGuide/.

[2] Spiral code generator. http://www.spiral.net/codegenerator.html.

[3] Student's t-distribution, table of selected values. https://en.wikipedia.org/w/index.php?title=Student%27s_t-distribution&oldid=682215766#Table_of_selected_values.

[4] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. In *OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.* ACM, 2007. http://users.elis.ugent.be/~leeckhou/papers/oopsla07-stat.pdf.

[5] Benjamin Hess. Automatic refactoring: Locality friendly interface enhancements for numerical functions. Master's thesis, ETH Zurich, April 2013. http://spiral.ece.cmu.edu:8080/pub-spiral/abstract.jsp?id=171.

[6] Benjamin Hess, Thomas R. Gross, and Markus Püschel. Automatic locality-friendly interface extension of numerical functions. In *GPCE 2014, Proceedings of the International Conference on Generative Programming: Concepts and Experiences*, pages 83–92. ACM, 2014. http://spiral.ece.cmu.edu:8080/pub-spiral/pubfile/paper_211.pdf.

Selbständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und nur mit den angegebenen Hilfsmitteln angefertigt habe und dass alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, durch Angaben von Quellen als Entlehnung kenntlich gemacht worden sind. Diese Bachelorarbeit wurde in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt.

Tübingen, 19-11-2015 Unterschrift