Guidelines for Fortran Programming on Heterogeneous Architectures

Kazem Ardaneh

Guidelines for Fortran Programming on Heterogeneous Architectures

Kazem Ardaneh

CMC/IPSL, Sorbonne Université - Campus Pierre et Marie Curie

May 16, 2025

Contents

1 Heterogeneous architectures

Heterogeneous architecture refers to a computing system or platform that employs different types of hardware components, such as processors and accelerator. These components are designed to work together efficiently, each contributing its strengths to achieve better overall performance, energy efficiency, or other desired characteristics. Supercomputers often utilize heterogeneous architectures to handle complex simulations, scientific calculations, and data analysis tasks. GPUs are frequently employed alongside traditional CPUs to accelerate computationally intensive workloads.

GPUs and CPUs have different strengths and are optimized for different types of tasks. CPUs are well-suited for tasks that require strong single-threaded performance and general-purpose computing, while GPUs excel at parallel processing tasks involving large datasets, such as graphics rendering, scientific simulations, and deep learning. In many applications, a combination of both CPUs and GPUs can provide a balanced approach to achieve optimal performance and efficiency.

1.1 NVIDIA Compiler

The NVIDIA compiler is a suite of compilers and tools designed for HPC and scientific computing. It is commonly used for compiling and optimizing code for parallel architectures, including CPUs and GPUs. NVIDIA compilers are known for their focus on performance optimization, parallelization, and vectorization to make the most of modern hardware capabilities. Key distinct features of the NVIDIA compiler are:

- Heterogeneous Computing: well-suited for heterogeneous computing environments where both CPUs and GPUs are utilized to accelerate computations,
- GPU Acceleration: the ability to compile CPU and GPU code within the same program, enabling seamless integration of parallel processing on GPUs,
- Directive-Based Parallelization: support OpenACC, a directive-based approach to parallel programming. Developers can annotate their code with directives to guide the compiler's parallelization and optimization efforts,
- Profiling and Debugging: tools for profiling and debugging parallel code which help to identify performance bottlenecks, analyze memory usage, and locate errors in parallel programs.

1.2 Compiler options

1.2.1 Debugging

Most modern compilers come equipped with a range of debugging options and tools that we can use to identify and resolve issues in our codes. Here are some common debugging options we might find in a compiler:

Table 1: Debuging options

Option	Description
-O0	Hinder any optimizations
-C/-Mbounds	Generate code to check array bounds
-g	Generate information for debugger + [-O0]
-Ktrap=fp	Controls on the floating-point exceptions
-Kieee	Floating-point operations with the IEEE 754 standard
-traceback	Add debug information for runtime traceback
-Mchkptr	Check for NULL pointers
-Mchkfpstk	Check the consistency of floating point stack at procedures call
-Mdepchk	Check dependence relations for vector or parallel code
-Mchkstk	Check for sufficient stack space upon subprogram entry
-Minfo	Print compiler feedback messages

1.2.2 Optimization

Optimization compiler options are settings that we can apply to our code during the compilation process to enable various levels of optimization. These options instruct the compiler to perform transformations on our code in order to improve its performance, reduce its memory usage, and make it execute more efficiently. However, optimization can sometimes introduce complexities and potential trade-offs, so it's important to understand the options available and their effects. Here are some common optimization compiler options:

Table 2: Optimization control.

Level	Description
-O0	No optimization, Compiler generates a basic block for each language statement.
-O1	Scheduling of basic blocks + Register allocation
-O	Scalar optimization + Induction recognition + Loop invariant motion. No SIMD vectorization
-O2	[-O1] + [-O] + SIMD vectorization + Cache alignment + Partial redundancy elimination
-O3	[-O1] + [-O2] + Aggressive hoisting + Scalar replacement optimizations
-O4	[-O1] + [-O2] + [-O3] + hoisting of guarded invariant floating point expressions

1.2.3 Vectorization

Vectorization is a technique used by compilers to optimize code by transforming scalar operations into vectorial operations. This takes advantage of the capabilities of modern CPUs and architectures that support Single Instruction Multiple Data (SIMD) instructions. Vectorization can significantly improve the performance of certain types of computations, especially those involving large datasets and repetitive calculations. Compiler options related to vectorization instruct the compiler to automatically identify and apply vectorization optimizations to our code. Here are some common vectorization compiler options:

Table 3: Vectorization control.

Option	Description
-Mvect=[no]altcode	Enable/disable alternate code generation for vectorization
-Mvect=cachesize:n	Assume a cache size of n when performing cache tiling
-Mvect=[no]fuse	Enable/disable loop fusion
-Mvect=partial	Generate partial vectorization
-Mvect=prefetch	Generate prefetch instructions
-Mvect=[no]sse	Enable/disable vectorize the loops with SSE/SSE2 + prefetch instructions
-Mvect=simd:n	Vectorize using SIMD either 128/256/512 bits wide
-Mvect=[no]assoc	Enable/disable associativity conversions
-Mvect=[no]tile	Enable/disable loop tiling

The -fast compiler's option is often used in various compilers to enable a combination of optimization flags that collectively aim to produce highly optimized and fast-running code. Here are some aspects to be aware of when using the -fast compiler option:

Table 4: -fast control.

Option	Description
-O2	See Table2
-Mvect=simd	See Table3
-Munroll=c:1	Unroll loops with a loop count of 1
-Mautoinline	Enable automatic function inlining
-Mflushz	Set SSE to flush-to-zero mode if a floating-point underflow occurs
-Mlre	Loop-carried redundancy elimination
-Mcache_align	Align large objects on cache-line boundaries

The Table 5 provides a detailed comparison of debug and production (**Prod**) compilation flags across three widely used Fortran compilers: GNU (`gfortran`), Intel (`ifort`), and NVIDIA (`nvfortran`). Under the **Debug** section, all compilers support the generation of debugging symbols using the `-g` flag, while GNU additionally supports verbose warning messages with `-Wall`. Profiling via Gprof is uniformly enabled across all compilers using the `-pg` flag. Bounds checking and null pointer dereferencing checks are supported in all compilers but with different syntax: GNU uses flags like `-fbounds-check` and `-fcheck=pointer`, Intel uses `-check bounds` and `-check pointers`, and NVIDIA uses `-Mbounds` and `-Mchkptr`. Stack space verification and floating-point exception trapping are also available, with compiler-specific flags such as `-fstack-check` (GNU), `-check stack` (Intel), and `-Mchkstk` (NVIDIA). IEEE 754 compliance is supported via strict floating-point flags like `-fp-model=strict` (Intel) or `-Kieee` (NVIDIA), and all compilers allow disabling optimizations with `-O0` and enable runtime error backtracing with `-fbacktrace` or `-traceback`. In the **Prod** section, optimization is generally enabled at level `-O2`. IEEE standard behavior can be enforced more thoroughly in GNU using extended flags like `-frounding-math` and `-fsignaling-nans`, whereas Intel and NVIDIA maintain strict behavior using `-fp-model=strict` and `-Kieee`, respectively. All compilers offer options to generate optimization reports (`-fopt-info-all`, `-qopt-report`, `-Minfo=all`), perform aggressive inlining of functions, generate position-independent code using `-fpic`, and apply advanced loop transformations such as fusion, tiling, and unrolling. Finally, the table includes options for aligning large objects to cache-line boundaries and setting architecture-specific tuning flags (`-march`, `-ax`, or `-tp`), making it a valuable reference for developers seeking consistent debugging and optimization strategies across different compilation environments.

Compiler	**GNU (`gfortran`)**	**Intel (`ifort`)**	**NVIDIA (`nvfortran`)**
Debug			
Debug information	`-Wall -g`	`-g`	`-g` **or** `-gopt`
Gprof profiling	`-pg`	`-pg`	`-pg`
Bounds check	`-fbounds-check` **or** `-fcheck=bounds` **or all**	`-check bounds` **or** `-check all`	`-C` **or** `-Mbounds`
Check for unintended dereferencing of NULL pointers	`-fcheck=pointer` **or** `-fcheck-pointer-bounds`	`-check pointers` **or** `-check all`	`-Mchkptr`
Check the stack for available space upon entry	`-fstack-check` **or** `-fcheck=mem`	`-check stack` `-fp-stack-check`	`-Mchkstk`
Trap floating-point exceptions	`-ffpe-trap=invalid,zero,overflow`	`-fpe0 -ftrapuv`	`-Ktrap=fp (inv,divz,ovf)`
IEEE 754 floating-point standard	`-fprotect-parens` `-fno-unsafe-math-optimizations`	`-fp-model=strict`	`-Kieee`
No optimizations	`-O0`	`-O0`	`-O0`
Backtrace on errors	`-fbacktrace`	`-traceback`	`-traceback`
Prod			
Optimization	`-O2`	`-O2`	`-O2`
IEEE 754 floating-point standard	`-fprotect-parens` `-fno-unsafe-math-optimizations` `-frounding-math` `-fsignaling-nans`	`-fp-model=strict`	`-Kieee`
Optimization report	`-fopt-info-all`	`-qopt-report=2 or 3`	`-Minfo=all`
Inline all functions	`-finline` **or** `-finline-functions` **or** `-fwhole-program`	`-finline-functions -ip -ipo`	`-Minline -Mextract` or `-Mautoinline`
Generate position-independent code	`-fpic`	`-fpic`	`-fpic/-KPIC/-Kpic`
Enable loop fusion	`-ftree-loop-optimize`	Included in `-O2`	`-Mvect=fuse`
Enable loop tiling	`-ftree-loop-distribution`	Included in `-O2`	`-Mvect=tile`
Align large objects on cache-line boundaries	`-falign-commons`	`-qopt-mem-layout-trans=2`	`-Mcache_align`
Enable loop unrolling	`-funroll-loops`	`-unroll` **or** `-funroll-loops`	`-Munroll`
Sets the target architecture	`-mtune={arch} -march={arch}` `-m{arch}`	`-ax{arch} or -x{arch} or -m{arch}`	`-tp={arch}`

Table 5: Compiler Options Comparison

2 Optimization

2.1 Register spilling

Register spilling refers to the situation in which a compiler cannot allocate all the variables and values that need to be stored in the limited number of available CPU registers. As a result, some of these variables or values need to be "spilled" or stored in memory locations (usually on the stack) instead of being kept in registers.

To reduce register spilling, minimize the scope of variables and the distance between usage.

```
! - -
x = a + b * c + d / e
! --> Many lines that do not use x
y = x + other_stuff
! + +
! --> Many lines that do not use x
x = a + b * c + d / e
y = x + other_stuff
```

Register spilling

Minimize the scope of variables and the gap between their utilization.

2.2 Factorization

The speed of program execution for numerical solutions is predominantly influenced by the quantity of function (subroutine) calls and arithmetic operations conducted within the program. Consequently, preference is given to algorithms demanding fewer function calls and arithmetic operations. For instance, when tasked with evaluating the value of a polynomial, such as:

$$p_4(x) = a_1x^4 + a_2x^3 + a_3x^2 + a_4x + a_5$$

It is more efficient to utilize a factorized structure (as shown below) than to employ the original form:

$$p_{4f}(x) = (((a_1x + a_2)x + a_3)x + a_4)x + a_5$$

It's worth noting that the factorized multiplication in $p_{4f}(x)$ involves 4 multiplications, whereas the original form $p_4(x)$ requires 4 + 3 + 2 + 1 = 9 multiplications. This concept is illustrated through the following example, where a polynomial $\sum_{i=0}^{N-1} a_i x^i$ is computed using both methods. The performance difference is evident, with the factorized multiplication approach being two times faster.

```
SUBROUTINE nonestedm()
      PARAMETER (n = 10**8 + 1, x = 1.)
      DIMENSION a(n), xi(0:n-1)
      DO k = 1, n
        a(k) = k
      ENDDO
      DO k = n-1, 0, -1
        xi(k) = x**k
      ENDDO
      p = SUM(a(:)*xi(:))
      WRITE(*,11) p
11    FORMAT('SUM =',2X,ES19.12)
ENDSUBROUTINE nonestedm
SUBROUTINE onnestedm()
      PARAMETER (n = 10**8 + 1, x = 1.)
      DIMENSION a(n), xi(0:n-1)
```

```
      DO k = 1, n
        a(k) = k
      ENDDO
      p = a(1)
      DO k = 2, n
        p = p * x + a(k)
      ENDDO
      WRITE(*,11) p
11    FORMAT('SUM =',2X,ES19.12)
ENDSUBROUTINE onnestedm
```

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 71.03      0.17     0.17        1   170.48   170.48  errors_nonestedm_
 29.25      0.24     0.07        1    70.20    70.20  errors_onnestedm_
  0.00      0.24     0.00        1     0.00   240.67  MAIN_
```

Factorization

Efficient computation is achieved by consistently factoring common expressions.

2.3 Loop ordering

Loop order pertains to the spatial locality within memory. Spatial locality implies that instructions located near the recently executed instruction have a higher probability of being executed. It involves utilizing data instructions that are closely situated to storage locations.

For instance, consider the following Fortran code. The arrangement of array elements in memory follows the order in which the left-hand side dimension grows. Consequently, $x(i+1,j)$, $y(i+1,j)$, and $z(i+1,j)$ are stored next to $x(i,j)$, $y(i,j)$, and $z(i,j)$, respectively. This arrangement enables the compiler to anticipate their usage in the subsequent loop iteration and prefetch them from memory to the cache.

In the second part of the code, we have reversed the sequence of loops. In this scenario, between two consecutive iterations of the inner loops, the variables $x(i,j+1)$, $y(i,j+1)$, and $z(i,j+1)$ are spaced apart by n variables from $x(i,j)$, $y(i,j)$, and $z(i,j)$. Attempting to prefetch data from these arrays is more likely to result in cache misses.

In the given example, the version with the correct loop ordering exhibits performance that is nearly ten times faster than the version with the reversed loop order. This highlights the significant impact that loop ordering can have on computational efficiency.

```
24  SUBROUTINE conti(x, y, z, n, m)
25  DIMENSION z(n,m), y(n,m), x(n,m)
26  #IFDEF _OpenACC
27    #IF defined LOO
28      !$ACC PARALLEL LOOP COLLAPSE(2) DEFAULT(PRESENT)
29    #ELIF defined KER
30      !$ACC KERNELS DEFAULT(PRESENT)
31    #ENDIF
32  #ENDIF
33  DO j = 1, m
34    DO i = 1, n
35      x(i,j) = x(i,j) + y(i,j) * z(i,j)
36    ENDDO
37  ENDDO
38  #IFDEF _OpenACC
39    #IF defined KER
40      !$ACC END KERNELS
```

```
41     #ENDIF
42   #ENDIF
43   ENDSUBROUTINE conti
44
45   SUBROUTINE nonconti(x, y, z, n, m)
46   DIMENSION z(n,m), y(n,m), x(n,m)
47   #IFDEF _OpenACC
48     #IF defined LOO
49       !$ACC PARALLEL LOOP COLLAPSE(2) DEFAULT(PRESENT)
50     #ELIF defined KER
51       !$ACC KERNELS DEFAULT(PRESENT)
52     #ENDIF
53   #ENDIF
54   DO i = 1, n
55     DO j = 1, m
56       x(i,j) = x(i,j) + y(i,j) * z(i,j)
57     ENDDO
58   ENDDO
59   #IFDEF _OpenACC
60     #IF defined KER
61       !$ACC END KERNELS
62     #ENDIF
63   #ENDIF
64   ENDSUBROUTINE nonconti
```

Running the code without applying any optimization options results in a significant tenfold decrease in the performance of the loop.

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 68.72      6.68     6.68        1     6.68     6.68  locality_nonconti_
 28.05      9.41     2.73        1     2.73     9.78  locality_locality_
  3.83      9.78     0.37        1     0.37     0.37  locality_conti_
  0.00      9.78     0.00        1     0.00     9.78  MAIN_
```

Upon incorporating the compiler's optimization option, the compiler generates the subsequent message concerning the nonconti routine. It appears that the compiler has inverted the sequence of loops, resulting in nearly identical execution times for the two loops.

```
nonconti:
     54, Loop interchange produces reordered loop nest: 55,54

  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 51.38      0.26     0.26        1   262.03   262.03  locality_nonconti_
 49.40      0.51     0.25        1   251.95   251.95  locality_conti_
  0.00      0.51     0.00        1     0.00     0.00  MAIN_
  0.00      0.51     0.00        1     0.00   251.95  locality_locality_
```

Is cache alignment always rectifiable by the compiler? To examine this scenario, we will assess execution on a GPU utilizing the OpenACC protocol. By employing the "PARALLEL LOOP" construct in conjunction with the "COLLAPSE" clause, which combines the two nested loops into a singular loop, we will observe the subsequent message from the compiler:

```
conti:
     28, Generating Tesla code
         33, !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
         34,   ! blockidx%x threadidx%x collapsed
nonconti:
     49, Generating Tesla code
```

```
          54, !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
          55,   ! blockidx%x threadidx%x collapsed


   Time(%)  Total Time (ns)                      Name
0  -------  ---------------  ------------------------
1     92.8       78,561,967  locality_nonconti_49_gpu
2      7.2        6,116,692     locality_conti_28_gpu
```

An alternative approach could involve using the "KERNELS" construct in lieu of the "PARALLEL" construct. The "KERNELS" construct affords the compiler maximum flexibility to parallelize and optimize the code according to the characteristics of the target accelerator. However, it also places a significant reliance on the compiler's inherent capability to autonomously parallelize the code. Consequently, variations might arise in terms of what different compilers can parallelize and their chosen methodologies for doing so. On the other hand, the "PARALLEL LOOP" directive serves as an explicit indication by the programmer that parallelizing the specific loop is both safe and advantageous. Meanwhile, adopting the "KERNELS" construct yields the ensuing output:

```
conti:
     30, Generating default present(x(:n,:m),z(:n,:m),y(:n,:m))
     33, Loop is parallelizable
     34, Loop is parallelizable
         Generating Tesla code
         33, !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
         34,   ! blockidx%x threadidx%x auto-collapsed
nonconti:
     51, Generating default present(x(:n,:m),z(:n,:m),y(:n,:m))
     54, Loop is parallelizable
     55, Loop is parallelizable
         Generating Tesla code
         54, !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
             collapsed-innermost
         55,   ! blockidx%x threadidx%x auto-collapsed

   Time(%)  Total Time (ns)                      Name
0  -------  ---------------  ------------------------
1     50.0        6,116,883     locality_conti_34_gpu
2     50.0        6,115,570  locality_nonconti_55_gpu
```

As evident from the output, it appears that the "KERNELS" construct potentially facilitates cache alignment, a feature that may not be as pronounced when employing the "PARALLEL" construct.

Loop ordering

Placing the loop over the leftmost dimension as the innermost loop is consistently a safe practice. Furthermore, due to the faster growth of the innermost loop, it is imperative that the largest dimension of an array corresponds to the leftmost dimension.

2.3.1 Array reductions

In certain scenarios, we compute the average of a multidimensional array to generate another array with reduced dimensions. On the CPU, when the loop ordering does not align with the spatial locality of the larger array, the compiler may reorganize the loops. However, the question arises: is this reordering beneficial in this context? Let's examine the subsequent example. Consider a 2D array where we compute an average along the second dimension. For the purpose of highlighting potential issues with compiler optimization, we assume that the second dimension is larger than the first dimension.

```
 81  SUBROUTINE redu01(x, s, n, m)
 82  DIMENSION x(n,m), s(n)
 83  #IFDEF _OPENACC
 84    #IF defined LOO
 85      !$ACC PARALLEL DEFAULT(PRESENT)
 86    #ELIF defined KER
 87      !$ACC KERNELS DEFAULT(PRESENT)
 88    #ENDIF
 89  #ENDIF
 90  !$ACC LOOP SEQ
 91  DO j = 1, m
 92    !$ACC LOOP
 93    DO i = 1, n
 94      s(i) = s(i) + x(i,j)
 95    ENDDO
 96  ENDDO
 97  #IFDEF _OPENACC
 98    #IF defined LOO
 99      !$ACC END PARALLEL
100    #ELIF defined KER
101      !$ACC END KERNELS
102    #ENDIF
103  #ENDIF
104  ENDSUBROUTINE redu01
105
106  SUBROUTINE redu02(x, s, n, m)
107  DIMENSION x(n,m), s(n)
108  #IFDEF _OPENACC
109    #IF defined LOO
110      !$ACC PARALLEL DEFAULT(PRESENT)
111    #ELIF defined KER
112      !$ACC KERNELS DEFAULT(PRESENT)
113    #ENDIF
114  #ENDIF
115  !$ACC LOOP
116  DO i = 1, n
117    !$ACC LOOP SEQ
118    DO j = 1, m
119      s(i) = s(i) + x(i,j)
120    ENDDO
121  ENDDO
122  #IFDEF _OPENACC
123    #IF defined LOO
124      !$ACC END PARALLEL
125    #ELIF defined KER
126      !$ACC END KERNELS
127    #ENDIF
128  #ENDIF
129  ENDSUBROUTINE redu02
```

Upon compiling the code on the CPU, the following messages are displayed: In the first version, there exists a perfect data locality concerning the x array. Consequently, the compiler produces a vector for the loop denoted by "i". In the subsequent version, we modify the loop order with the intent of observing whether the compiler generates a vector for the larger dimension, represented by the loop marked "j". However, the compiler's optimization process results in the reversal of the loop. Consequently, both versions exhibit identical performance outcomes.

```
redu01:
     93, Generated vector simd code for the loop
redu02:
    116, Loop interchange produces reordered loop nest: 118,116
```

```
         Generated vector simd code for the loop

  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 50.00      0.02     0.02        1    20.11    20.11  locality_redu01_
 50.00      0.04     0.02        1    20.11    20.11  locality_redu02_
```

The issue becomes evident when considering the GPU. The loop labeled "j" cannot be parallelized due to the exposed use of the s array. In such a scenario, optimal performance can be attained by enclosing the sequential loop within the parallel loop, similar to the approach in the second version. Notably, the second version demonstrates a twofold increase in speed compared to the first version.

```
redu01:
     85, Generating Tesla code
         91, !$acc loop seq
         93, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
     85, Generating default present(s(:n),x(:n,:m))
     91, Loop carried dependence due to exposed use of s(:n) prevents parallelization
redu02:
    110, Generating Tesla code
        116, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
        118, !$acc loop seq
    110, Generating default present(x(:n,:m),s(:n))

   Time(%)  Total Time (ns)                    Name
0  -------  ---------------  ----------------------
1     68.8        4,241,792   locality_redu01_85_gpu
2     31.2        1,922,450  locality_redu02_110_gpu
```

Loop ordering: Array reductions

When reducing multidimensional arrays to fewer dimensions, organizing the loops over the reduced array as the outermost loops results in improved performance.

2.4 Loop expressing

The loop expression pertains to temporal locality within memory. Temporal locality means that an instruction that is recently executed has a high chance of execution again. So the instruction is kept in cache memory such that it can be fetched easily and takes no time to search for the same instruction. Let's explore various methods of loop expression through the following example.

```
45   SUBROUTINE expl(x, z, s, t, n, m)
46   DIMENSION y(n,m), z(n,m), x(n,m), s(n,m), t(n,m)
47   #IFDEF _OPENACC
48     #IF defined LOO
49       !$ACC PARALLEL LOOP COLLAPSE(2) PRESENT(x,z,s,t) CREATE(y)
50     #ELIF defined KER
51       !$ACC KERNELS PRESENT(x,z,s,t) CREATE(y)
52     #ENDIF
53   #ENDIF
54   DO j = 1, m
55     DO i = 1, n
56      x(i,j) = x(i,j) * t(i,j)
57      y(i,j) = x(i,j)**3.
58      z(i,j) = z(i,j) + y(i,j)
59      s(i,j) = s(i,j)/3. + z(i,j)
60      t(i,j) = t(i,j) * s(i,j)
```

```
61      ENDDO
62    ENDDO
63    #IFDEF _OPENACC
64      #IF defined KER
65        !$ACC END KERNELS
66      #ENDIF
67    #ENDIF
68    ENDSUBROUTINE expl
69
70    SUBROUTINE impl(x, z, s, t)
71    DIMENSION z(:,:), x(:,:), s(:,:), t(:,:)
72    DIMENSION y(SIZE(x, DIM=1),SIZE(x, DIM=2))
73    #IFDEF _OPENACC
74      !$ACC KERNELS PRESENT(x,z,s,t) CREATE(y)
75    #ENDIF
76    x(:,:) = x(:,:) * t(:,:)
77    y(:,:) = x(:,:)**3.
78    z(:,:) = z(:,:) + y(:,:)
79    s(:,:) = s(:,:)/3. + z(:,:)
80    t(:,:) = t(:,:) * s(:,:)
81    #IFDEF _OPENACC
82      !$ACC END KERNELS
83    #ENDIF
84    ENDSUBROUTINE impl
85
86    SUBROUTINE miximex(x, z, s, t, m)
87    DIMENSION z(:,:), x(:,:), s(:,:), t(:,:)
88    DIMENSION y(SIZE(x, DIM=1),SIZE(x, DIM=2))
89    #IFDEF _OPENACC
90      #IF defined LOO
91        !$ACC PARALLEL LOOP PRESENT(x,z,s,t) CREATE(y)
92      #ELIF defined KER
93        !$ACC KERNELS PRESENT(x,z,s,t) CREATE(y)
94      #ENDIF
95    #ENDIF
96    DO j = 1, m
97     x(:,j) = x(:,j) * t(:,j)
98     y(:,j) = x(:,j)**3.
99     z(:,j) = z(:,j) + y(:,j)
100    s(:,j) = s(:,j)/3. + z(:,j)
101    t(:,j) = t(:,j) * s(:,j)
102   ENDDO
103   #IFDEF _OPENACC
104     #IF defined KER
105       !$ACC END KERNELS
106     #ENDIF
107   #ENDIF
108   ENDSUBROUTINE miximex
```

In the provided example, the code reuses the variable $t(i, j)$ after three lines of instructions. The management of the cache in this context is optimized by considering the temporal locality of variables, prompting the compiler to retain these variables within the cache. Conversely, in the loop constructed using array notation, there exist $(4n - 1) \times (4m - 1)$ lines of instructions before the reuse of $t(i, j)$. For a substantial loop, the array t may be evicted from the cache and subsequently reloaded. Consequently, the loop's execution might decelerate due to the overhead associated with data access. We also employed a combination of the two approaches. In terms of performance, the explicit loop expression proves to be the fastest, followed by the mixed approach, with the least optimal performance observed in the case involving array notation.

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 49.42      9.49     9.49        1     9.49    19.24  loops_tempo_
 20.82     13.49     4.00        1     4.00     4.00  loops_impl_
 20.61     17.44     3.96        1     3.96     3.96  loops_miximex_
  9.34     19.24     1.79        1     1.79     1.79  loops_expl_
  0.00     19.24     0.00        1     0.00    19.24  MAIN_
```

Upon recompiling the code with the optimizer options enabled, the compiler generates the following messages:

```
expl:
     54, Loop not fused: function call before adjacent loop
     55, Generated vector simd code for the loop
impl:
     76, Loop not fused: different loop trip count
         Generated vector simd code for the loop
     77, Loop not fused: different loop trip count
         Generated vector simd code for the loop
     78, Loop not fused: different loop trip count
         Generated vector simd code for the loop
     79, Loop not fused: different loop trip count
         Generated vector simd code for the loop
     80, Loop not fused: function call before adjacent loop
         Generated vector simd code for the loop
miximex:
     96, Loop not fused: function call before adjacent loop
     97, Loop not fused: different loop trip count
         Generated vector simd code for the loop
     98, Loop not fused: different loop trip count
         Generated vector simd code for the loop
     99, Loop not fused: different loop trip count
         Generated vector simd code for the loop
    100, Loop not fused: different loop trip count
         Generated vector simd code for the loop
    101, Generated vector simd code for the loop
```

As indicated in the compiler report, when employing the explicit loop construct wherein all calculations are consolidated within a single loop by the developer, the compiler engages vectorization for the innermost loop. In contrast, for both the array notation and mixed versions, while the compiler continues to vectorize the inner loops of each instruction, it encounters challenges in merging the instructions. The performance report underscores a notable distinction: the explicit loop outperforms the others by nearly twofold.

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 43.17      2.53     2.53        1     2.53     2.53  loops_impl_
 37.88      4.76     2.22        1     2.22     2.22  loops_miximex_
 19.11      5.88     1.12        1     1.12     1.12  loops_expl_
  0.00      5.88     0.00        1     0.00     0.00  MAIN_
  0.00      5.88     0.00        1     0.00     5.88  loops_tempo_
```

When compiling the code for GPU utilization via OpenACC, the explicit loop version can be executed on the GPU using both the "KERNELS" and "PARALLEL" constructs. However, the "KERNELS" construct exclusively applies to the array notation. By employing the "KERNELS" construct for all three versions, insights from the compiler report emerge: The compiler launches a single kernel for the explicit loop version, collapsing the two loops into one while for the next two versions, the compiler initiates individual kernels for each instruction. In terms of performance, the anticipated outcome is evident: The explicit loop demonstrates the highest speed (30.8%). The mixed version follows suit as the second fastest (a cumulative 33.1%, comprising 24.3% and four instances of 2.2%). The array notation, representing the implicit loop, shows the slowest performance.

```
expl:
     51, Generating present(t(:,:),x(:,:),z(:,:),s(:,:))
         Generating create(y(:,:)) [if not already present]
     54, Loop is parallelizable
     55, Loop is parallelizable
         Generating Tesla code
         54, !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
         55,   ! blockidx%x threadidx%x auto-collapsed
impl:
     74, Generating present(t(:,:),x(:,:),z(:,:),s(:,:))
         Generating create(y(:,:)) [if not already present]
     76, Loop is parallelizable
         Generating Tesla code
         76,   ! blockidx%x threadidx%x auto-collapsed
             !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
     77, Loop is parallelizable
         Generating Tesla code
         77,   ! blockidx%x threadidx%x auto-collapsed
             !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
     78, Loop is parallelizable
         Generating Tesla code
         78,   ! blockidx%x threadidx%x auto-collapsed
             !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
     79, Loop is parallelizable
         Generating Tesla code
         79,   ! blockidx%x threadidx%x auto-collapsed
             !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
     80, Loop is parallelizable
         Generating Tesla code
         80,   ! blockidx%x threadidx%x auto-collapsed
             !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
miximex:
     93, Generating present(t(:,:),x(:,:),z(:,:),s(:,:))
         Generating create(y(:,:)) [if not already present]
     96, Loop is parallelizable
     97, Loop is parallelizable
         Generating Tesla code
         96, !$acc loop gang, vector(4) ! blockidx%y threadidx%y
         97, !$acc loop gang, vector(32) ! blockidx%x threadidx%x
     98, Loop is parallelizable
         Generating Tesla code
         96, !$acc loop gang, vector(4) ! blockidx%y threadidx%y
         98, !$acc loop gang, vector(32) ! blockidx%x threadidx%x
     99, Loop is parallelizable
         Generating Tesla code
         96, !$acc loop gang, vector(4) ! blockidx%y threadidx%y
         99, !$acc loop gang, vector(32) ! blockidx%x threadidx%x
    100, Loop is parallelizable
         Generating Tesla code
         96, !$acc loop gang, vector(4) ! blockidx%y threadidx%y
        100, !$acc loop gang, vector(32) ! blockidx%x threadidx%x
    101, Loop is parallelizable
         Generating Tesla code
         96, !$acc loop gang, vector(4) ! blockidx%y threadidx%y
        101, !$acc loop gang, vector(32) ! blockidx%x threadidx%x
```

The performance report on the GPU is given as

```
    Time(%)  Total Time (ns)                  Name
0   -------  --------------  --------------------
1      30.8     135,331,997      loops_expl_55_gpu
2      27.5     121,123,545      loops_impl_77_gpu
3      24.3     106,930,294   loops_miximex_98_gpu
4       2.2       9,696,291      loops_impl_79_gpu
5       2.2       9,578,369   loops_miximex_99_gpu
6       2.2       9,562,945  loops_miximex_101_gpu
7       2.2       9,560,259  loops_miximex_100_gpu
8       2.2       9,557,506   loops_miximex_97_gpu
9       2.2       9,540,833      loops_impl_80_gpu
10      2.2       9,530,785      loops_impl_78_gpu
11      2.2       9,514,882      loops_impl_76_gpu
```

Employing the "PARALLEL" construct with "COLLAPSE" for the explicit loop, utilizing "KERNELS" for the array notation (as "PARALLEL" is not permitted), and employing "PARALLEL" for the mixed version yields the following observation: the explicit loop remains the fastest, while the performance of the mixed version experiences a decline. This reduction in performance for the mixed version is attributed to the compiler's challenge in determining the optimal distribution of instructions across gangs and vectors.

Loop expressing

During the computation phase, the utilization of the explicit loop construct consistently results in better performance, whether executed on a CPU or GPU.

2.4.1 Initializing/Copying Arrays

In cases where a loop does not involve data reuse, as seen during the initialization phase of calculations, optimizing efforts should be directed toward enhancing spatial data locality. To cater to this goal, specialized functions are available for initialization and copying, denoted as msetN and mcopyN, where N corresponds to the calculation precision. It's important to emphasize that these functions are not native Fortran intrinsic functions. Consequently, the code should be structured in a manner that prompts the compiler to utilize them (if automatic utilization is not feasible).

Let's revisit the three approaches to expressing the loop construct: explicit, array, and mixed, as illustrated in the following example.

```
21  SUBROUTINE loop(x, y, z, n, m)
22  DIMENSION x(n,m), y(n,m), z(n,m)
23  #IFDEF _OPENACC
24    #IF defined LOO
25      !$ACC PARALLEL LOOP COLLAPSE(2) PRESENT(x,y,z)
26    #ELIF defined KER
27      !$ACC KERNELS PRESENT(x,y,z)
28    #ENDIF
29  #ENDIF
30  DO j = 1, m
31    DO i = 1, n
32     y(i,j) = 8.0
33     z(i,j) = x(i,j)
34    ENDDO
35  ENDDO
36  #IFDEF _OPENACC
37    #IF defined KER
38      !$ACC END KERNELS
39    #ENDIF
40  #ENDIF
```

```
41   ENDSUBROUTINE loop
42
43   SUBROUTINE array(x, y, z)
44   DIMENSION x(:,:), y(:,:), z(:,:)
45   #IFDEF _OPENACC
46   !$ACC KERNELS PRESENT(x,y,z)
47   #ENDIF
48   y(:,:) = 8.0
49   z(:,:) = x(:,:)
50   #IFDEF _OPENACC
51   !$ACC END KERNELS
52   #ENDIF
53   ENDSUBROUTINE array
54
55   SUBROUTINE mix(x, y, z, m)
56   DIMENSION x(:,:), y(:,:), z(:,:)
57   #IFDEF _OPENACC
58     #IF defined LOO
59       !$ACC PARALLEL LOOP PRESENT(x,y,z)
60     #ELIF defined KER
61       !$ACC KERNELS PRESENT(x,y,z)
62     #ENDIF
63   #ENDIF
64   DO j = 1, m
65    y(:,j) = 8.0
66    z(:,j) = x(:,j)
67   ENDDO
68   #IFDEF _OPENACC
69     #IF defined KER
70       !$ACC END KERNELS
71     #ENDIF
72   #ENDIF
73   ENDSUBROUTINE mix
```

Upon compiling the code with optimization options, the ensuing outcomes are evident: The explicit loop is vectorized by the compiler. For the array and mixed versions, the compiler employs optimized memory functions, mset and mcopy, resulting in enhanced performance. It's worth noting a potential drawback of the array notation: if the cache becomes saturated, it can lead to latencies that adversely affect overall performance. The mixed construct strategically confines the application of mset or mcopy to the first dimension of arrays, aligning with an optimal cache blocking strategy.

```
loop:
     31, Generated vector simd code for the loop
array:
     48, Memory set idiom, loop replaced by call to __c_mset8
     49, Memory copy idiom, loop replaced by call to __c_mcopy8
mix:
     65, Memory set idiom, loop replaced by call to __c_mset8
     66, Memory copy idiom, loop replaced by call to __c_mcopy8

  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
100.50      3.96     3.96        1     3.96     3.96  for_loop_
  0.00      3.96     0.00        1     0.00     0.00  MAIN_
  0.00      3.96     0.00        1     0.00     0.00  for_array_
  0.00      3.96     0.00        1     0.00     3.96  for_initial_
  0.00      3.96     0.00        1     0.00     0.00  for_mix_
```

When utilizing the "KERNELS" construct on the GPU, the explicit loop structure and array notation yield equivalent outcomes. In both instances, the compiler performs an auto-collapsed operation as illustrated below. Nevertheless, in the mixed version, the compiler treats the outer and inner loops

distinctly, employing diverse gang, worker, and vector sets. Consequently, this approach results in inferior performance compared to the two preceding cases.

```
loop:
     27, Generating present(x(:,:),z(:,:),y(:,:))
     30, Loop is parallelizable
     31, Loop is parallelizable
         Generating Tesla code
         30, !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
         31,   ! blockidx%x threadidx%x auto-collapsed
array:
     46, Generating present(x(:,:),z(:,:),y(:,:))
     48, Loop is parallelizable
         Generating Tesla code
         48,   ! blockidx%x threadidx%x auto-collapsed
             !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
     49, Loop is parallelizable
         Generating Tesla code
         49,   ! blockidx%x threadidx%x auto-collapsed
             !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
mix:
     61, Generating present(x(:,:),z(:,:),y(:,:))
     64, Loop is parallelizable
     65, Loop is parallelizable
         Generating Tesla code
         64, !$acc loop gang, vector(4) ! blockidx%y threadidx%y
         65, !$acc loop gang, vector(32) ! blockidx%x threadidx%x
     66, Loop is parallelizable
         Generating Tesla code
         64, !$acc loop gang, vector(4) ! blockidx%y threadidx%y
         66, !$acc loop gang, vector(32) ! blockidx%x threadidx%x


   Time(%)  Total Time (ns)              Name
0  -------  ---------------  ----------------
1     33.5        9,465,615   for_loop_31_gpu
2     22.7        6,395,744    for_mix_66_gpu
3     22.4        6,327,135  for_array_49_gpu
4     10.7        3,015,919    for_mix_65_gpu
5     10.7        3,009,870  for_array_48_gpu
```

Initializing/Copying Arrays

For execution on the CPU, opting for a mixed loop and array approach consistently results in better performance. However, on the GPU, the explicit loop consistently ensures better performance.

2.4.2 Loop fusion

Loop fusion involves the merging of neighboring or closely situated loops into a singular loop. The advantages of loop fusion mirror those of loop unrolling. Moreover, when the two pre-optimized loops access shared data, loop fusion enhances cache locality, furnishing the compiler with greater prospects for harnessing instruction-level parallelism. Let's examine the subsequent example to delve deeper into this concept.

```
23  SUBROUTINE nofused(x, y, z, s, v)
24  DIMENSION x(:,:), y(:,:), z(:,:), s(:,:), v(:,:)
25  DIMENSION t(SIZE(x, DIM=1),SIZE(x, DIM=2))
26  DIMENSION u(SIZE(x, DIM=1),SIZE(x, DIM=2))
27  #IFDEF _OPENACC
28    #IF defined KER
29      !$ACC KERNELS PRESENT(x,y,z,s,v) CREATE(t,u)
30    #ENDIF
31  #ENDIF
32  t(:,:) = x(:,:) * y(:,:)
33  u(:,:) = z(:,:) + s(:,:)
34  v(:,:) = t(:,:) / u(:,:)
35  #IFDEF _OPENACC
36    #IF defined KER
37      !$ACC END KERNELS
38    #ENDIF
39  #ENDIF
40  ENDSUBROUTINE nofused
41
42  SUBROUTINE fuseda(x, y, z, s, v, n, m)
43  DIMENSION x(n,m), y(n,m), z(n,m), s(n,m), t(n,m), u(n,m), v(n,m)
44  #IFDEF _OPENACC
45    #IF defined KER
46      !$ACC KERNELS PRESENT(x,y,z,s,v) CREATE(t,u)
47    #ENDIF
48  #ENDIF
49  t(:,:) = x(:,:) * y(:,:)
50  u(:,:) = z(:,:) + s(:,:)
51  v(:,:) = t(:,:) / u(:,:)
52  #IFDEF _OPENACC
53    #IF defined KER
54      !$ACC END KERNELS
55    #ENDIF
56  #ENDIF
57  ENDSUBROUTINE fuseda
58
59  SUBROUTINE fuseddo(x, y, z, s, v, n, m)
60  DIMENSION x(n,m), y(n,m), z(n,m), s(n,m), v(n,m)
61  #IFDEF _OPENACC
62    #IF defined LOO
63      !$ACC PARALLEL LOOP COLLAPSE(2) DEFAULT(PRESENT)
64    #ELIF defined KER
65      !$ACC KERNELS DEFAULT(PRESENT)
66    #ENDIF
67  #ENDIF
68  DO j = 1, m
69    DO i = 1, n
70      t = x(i,j) * y(i,j)
71      u = z(i,j) + s(i,j)
72      v(i,j) = t / u
73    ENDDO
74  ENDDO
75  #IFDEF _OPENACC
76    #IF defined KER
77      !$ACC END KERNELS
78    #ENDIF
79  #ENDIF
80  ENDSUBROUTINE fuseddo
```

In this example, the dummy arguments are established through implicit shape declaration. The compiler notification indicates that three loops remain unfused. This outcome arises from the com-

piler's lack of knowledge regarding the number of iterations (size) associated with each dummy argument.

```
nofused:
     32, Loop not fused: different loop trip count
         Generated vector simd code for the loop
     33, Loop not fused: different loop trip count
         Generated vector simd code for the loop
     34, Loop not fused: function call before adjacent loop
         Generated vector simd code for the loop
```

We have the option to furnish the compiler with information regarding the size of the dummy argument. In such instances, the compiler gains the understanding that the subsequent three loops share the same number of iterations. Consequently, when employing the optimization option, the compiler is empowered to fuse these loops. Consequently, the compiler successfully merges loops 55, 56, and 57, as depicted. The process of loop fusion, in this case, contributes to enhanced performance.

```
fuseda:
     49, Array assignment / Forall at line 50,51 fused
         Loop not fused: function call before adjacent loop
         Generated vector simd code for the loop
```

As previously emphasized, utilizing an explicit loop is preferable over employing an array notation. Furthermore, in the provided example, the arrays t and u are local variables that could potentially be substituted with scalars. Adopting scalars in place of arrays has yielded an improvement in performance.

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 45.04      1.46     1.46        1     1.46     1.46  fuse_nofused_
 42.26      2.84     1.37        1     1.37     1.37  fuse_fuseda_
 12.96      3.26     0.42        1     0.42     0.42  fuse_fuseddo_
  0.00      3.26     0.00        1     0.00     0.00  MAIN_
  0.00      3.26     0.00        1     0.00     3.26  fuse_fusion_
```

Upon compiling the code for GPU execution, a comparable pattern emerges: loop fusion is achievable when we explicitly declare the shape of the arrays. However, the explicit loop maintains an advantage by employing scalars within the loop.

```
nofused:
     29, Generating present(v(:,:),x(:,:),y(:,:),z(:,:),s(:,:))
         Generating create(u(:,:),t(:,:)) [if not already present]
     32, Loop is parallelizable
         Generating Tesla code
         32,   ! blockidx%x threadidx%x auto-collapsed
             !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
     33, Loop is parallelizable
         Generating Tesla code
         33,   ! blockidx%x threadidx%x auto-collapsed
             !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
     34, Loop is parallelizable
         Generating Tesla code
         34,   ! blockidx%x threadidx%x auto-collapsed
             !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
fuseda:
     46, Generating present(v(:,:),x(:,:),y(:,:),z(:,:),s(:,:))
         Generating create(u(:,:),t(:,:)) [if not already present]
     49, Array assignment / Forall at line 51,50 fused
         Loop is parallelizable
         Generating Tesla code
```

```
          49,   ! blockidx%x threadidx%x auto-collapsed
              !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
fuseddo:
     65, Generating default present(s(:n,:m),z(:n,:m),y(:n,:m),v(:n,:m),x(:n,:m))
     68, Loop is parallelizable
     69, Loop is parallelizable
         Generating Tesla code
         68, !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
         69,   ! blockidx%x threadidx%x auto-collapsed
```

Looking at the performance on GPU reveals the following: when utilizing an implicit array, the compiler initiates three KERNELS, contributing to approximately 43.2% of all calculations. In contrast, the explicit loop with scalars demonstrates a significantly reduced workload, accounting for only 23% of the total calculations.

```
   Time(%)  Total Time (ns)                 Name
0  -------  ---------------  --------------------
1     33.7       11,165,665   fuse_fuseda_49_gpu
2     23.2        7,698,966  fuse_fuseddo_69_gpu
3     14.4        4,779,150  fuse_nofused_33_gpu
4     14.4        4,770,798  fuse_nofused_34_gpu
5     14.3        4,746,222  fuse_nofused_32_gpu
```

Loop fusion

Merging identical loops into a singular loop enhances vectorization on the CPU and improves KERNEL occupancy on the GPU. This yields improved performance in both cases.

2.5 Loop tiling

To mitigate cache misses and paging activity, it is effective to divide extensive matrices into smaller rectangular blocks. This partitioning is achieved by segmenting the "iteration space" into blocks. An illustrative case is the multiplication of square matrices.

```
24  SUBROUTINE mmnotailing (x, y, z, n)
25  DIMENSION x(n,n), y(n,n), z(n,n)
26  #IFDEF _OPENACC
27    #IF defined KER
28      !$ACC KERNELS DEFAULT(PRESENT)
29    #ENDIF
30  #ENDIF
31  x(:,:) = 0.
32  DO j = 1, n
33    DO k = 1, n
34      DO i = 1, n
35        x(i,j) = x(i,j) + y(i,k) * z(k,j)
36      END DO
37    END DO
38  END DO
39  #IFDEF _OPENACC
40    #IF defined KER
41      !$ACC END KERNELS
42    #ENDIF
43  #ENDIF
44  ENDSUBROUTINE mmnotailing
```

According to its definition, every entry in the product matrix x necessitates the inclusion of an entire row and column from the matrices undergoing multiplication. When implemented in a straightforward manner, as demonstrated above, this implies that one of the matrices will always be accessed along its less-efficient direction (such as row-wise in Fortran). Furthermore, given that each row of y and each column of z are used N times, there could potentially be as many as N repeated fetches of both matrices in their entirety, as seen in the conventional matrix multiplication approach. This poses a significant performance drawback and prompts an exploration for an improved approach to define the memory layout for this operation.

A viable strategy involves tailing a conventional loop, wherein an outer loop iterates over the tails, while the inner loop traverses each tail sequentially. The selection of the tile size should be made carefully to ensure it fits within the cache. This optimization technique aims to mitigate the mentioned performance challenges.

```
46  SUBROUTINE mmontailing (x, y, z, n, is, js, ks)
47  DIMENSION x(n,n), y(n,n), z(n,n)
48  #IFDEF _OPENACC
49    #IF defined KER
50      !$ACC KERNELS DEFAULT(PRESENT)
51    #ENDIF
52  #ENDIF
53  x(:,:) = 0.
54  DO it = 1, n, is
55    DO jt = 1, n, js
56      DO kt = 1, n, ks
57        DO j = jt, min(n, jt+js-1)
58          DO k = kt, min(n, kt+ks-1)
59            DO i = it, min(n, it+is-1)
60              x(i,j) = x(i,j) + y(i,k) * z(k,j)
61            ENDDO
62          ENDDO
63        ENDDO
64      ENDDO
65    ENDDO
66  ENDDO
67  #IFDEF _OPENACC
68    #IF defined KER
69      !$ACC END KERNELS
70    #ENDIF
71  #ENDIF
72  ENDSUBROUTINE mmontailing
```

This complex loop structure incurs additional overhead in terms of loop initialization and control. However, it offers the advantage of minimizing paging activity and maximizing the reuse of data already present in the cache. When running the code on the CPU, the loop-tailing version exhibits better performance.

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 72.72      5.46     5.46        1     5.46     5.46  caching_mmnotailing_
 27.67      7.54     2.08        1     2.08     2.08  caching_mmontailing_
  0.00      7.54     0.00        1     0.00     0.00  MAIN_
  0.00      7.54     0.00        1     0.00     7.54  caching_cachemanager_
```

The scenario on the GPU presents a contrast. In the GPU context, the compiler exclusively parallelizes loops 62 and 63, leaving the rest of the loops as sequential. Consequently, the manual loop tailing approach is anticipated to perform significantly slower on the GPU compared to the conventional implementation.

```
     50, Generating default present(z(:,:),y(:,:),x(:,:))
     53, Loop is parallelizable
         Generating Tesla code
         53,   ! blockidx%x threadidx%x auto-collapsed
             !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
     54, Loop carried dependence of x prevents parallelization
         Loop carried backward dependence of x prevents vectorization
         Generating Tesla code
         54, !$acc loop seq
         55, !$acc loop seq
         56, !$acc loop seq
         57, !$acc loop seq
         58, !$acc loop vector(128) ! threadidx%x
         59, !$acc loop seq
     55, Loop carried dependence of x prevents parallelization
         Loop carried backward dependence of x prevents vectorization
     56, Loop carried dependence due to exposed use of x(:,:) prevents parallelization
     57, Loop is parallelizable
     58, Loop is parallelizable
     59, Complex loop carried dependence of x prevents parallelization
         Loop carried dependence of x prevents parallelization
         Loop carried backward dependence of x prevents vectorization
         Inner sequential loop scheduled on accelerator
```

Nonetheless, there exists a means to direct the GPU compiler towards implementing loop tailing. This is achieved through the utilization of the TILE clause, demonstrated as follows:

```
 79   SUBROUTINE mmontailingGPU (x, y, z, n)
 80   DIMENSION x(n,n), y(n,n), z(n,n)
 81   #IFDEF _OpenACC
 82     #IF defined LOO
 83       !$ACC PARALLEL LOOP COLLAPSE(2) DEFAULT(PRESENT)
 84     #ELIF defined KER
 85       !$ACC KERNELS DEFAULT(PRESENT)
 86     #ENDIF
 87   #ENDIF
 88   x(:,:) = 0.
 89   !$ACC LOOP TILE(64,64)
 90   DO i = 1, n
 91     DO j = 1, n
 92       DO k = 1, n
 93         x(i,j) = x(i,j) + y(i,k) * z(k,j)
 94       END DO
 95     END DO
 96   END DO
 97   #IFDEF _OpenACC
 98     #IF defined KER
 99       !$ACC END KERNELS
100     #ENDIF
101   #ENDIF
102   ENDSUBROUTINE mmontailingGPU
```

The profile report indicates that compiler-induced tailing enhances GPU performance, while manual tailing results in inferior performance on the GPU.

```
   Time(%)  Total Time (ns)                          Name
0  -------  ---------------  ------------------------------
1     98.4    6,312,379,748     caching_mmontailing_54_gpu
2      1.5       94,337,665     caching_mmnotailing_34_gpu
3      0.1        8,996,496  caching_mmontailinggpu_85_gpu
4      0.0           60,767     caching_mmnotailing_31_gpu
```

```
5       0.0            60,512      caching_mmontailing_53_gpu
6       0.0            58,559   caching_mmontailinggpu_81_gpu
```

Loop tiling

Loop tailing enhances performance on both the CPU and GPU. On the CPU, this optimization needs to be manually implemented by the developer (with directives available in the Cray compiler). Conversely, on the GPU, loop tailing is achieved using the "TILE" clause within the loop construct.

2.6 Repeated array accesses

There are cases inside a loop where multiple accesses are made to the same index of an array. Let's revisit the matrix multiplication example discussed in the preceding section:

```
32   DO j = 1, n
33     DO i = 1, n
34       x(i,j) = 0.
35       DO k = 1, n
36         x(i,j) = x(i,j) + y(i,k) * z(k,j)
37       END DO
38     END DO
39   END DO
```

When dealing with $x(i, j)$ in each iteration, the compiler is required to access cache memory repeatedly, resulting in inefficiency. A potential remedy for addressing repeated array accesses involves substituting them with temporary scalar variables, as demonstrated below:

```
32   DO j = 1, n
33     DO i = 1, n
34       tem = 0.
35       DO k = 1, n
36         tem = tem + y(i,k) * z(k,j)
37       END DO
38       x(i,j) = tem
39     END DO
40   END DO
```

This approach prompts the compiler to allocate the variable tem within a register, ensuring rapid accessibility for the floating-point unit's operations.

Repeated array accesses

Substitute the repeated array accesses with temporary scalar variables.

3 Vectorization/Parallelization

Vectorization involves the process of converting an algorithm that initially operates on individual scalar values, processing one pair of operands at a time, into a vector operation. In a vector operation, a single instruction can operate on a group of data elements simultaneously, known as SIMD. Examples of operations that can be vectorized include tasks such as Linear Algebra, Fast Fourier Transforms, and Vector Math. Various approaches can be employed to vectorize code, including automatic vectorization by the compiler, using SIMD constructs, utilizing Array notation, and more. When

dealing with nested loops, it's important to note that the compiler typically vectorizes the inner loop. Compiler-generated vectorization reports contain valuable insights. These reports highlight which loops were vectorized and, importantly, provide explanations for loops that were not vectorized. The lack of successful vectorization can stem from various facts.

3.1 I/O

Vectorization of a loop can be broken by I/O operations, such as leaving a debugging "WRITE" statement. In the following example, a debugging break point is situated within the loops.

```
18  SUBROUTINE iononv(x, y, z, n, m)
19  DIMENSION z(n,m), y(n,m), x(n,m)
20  DO j = 1, m
21    DO i = 1, n
22     x(i,j) = 2.*x(i,j)*z(i,j) + x(i,j)*x(i,j) + x(i,j)*x(i,j)
23     y(i,j) = 1.-EXP(-x(i,j))
24     IF (y(i,j).GT.0.5) THEN
25       WRITE(*,10)i, j, y(i,j)
26     ENDIF
27    ENDDO
28  ENDDO
29  10    FORMAT('Error at',2X,'i=',I5,2X,'j=',I5,2X,'Where',2X,'y=',F19.12)
30  ENDSUBROUTINE iononv
```

Upon examining the compiler message, it is evident that the loop is not subject to vectorization owing to an external function call.

```
21, Loop not vectorized/parallelized: contains call
```

To benefit from vectorization, the above code can be adjusted by relocating all debugging options outside of the calculation loop. It's important to mention that the conditional statement could also be eliminated using a reduction construct. The revised code is as follows:

```
38  SUBROUTINE iov(x, y, z, n, m)
39  DIMENSION z(n,m), y(n,m), x(n,m)
40  #IFDEF _OpenACC
41    #IF defined LOO
42      !$ACC PARALLEL LOOP COLLAPSE(2) PRESENT(x,y,z)
43    #ELIF defined KER
44      !$ACC KERNELS PRESENT(x,y,z)
45    #ENDIF
46  #ENDIF
47  DO j = 1, m
48    DO i = 1, n
49     x(i,j) = 2.*x(i,j)*z(i,j) + x(i,j)*x(i,j) + x(i,j)*x(i,j)
50     y(i,j) = 1.-EXP(-x(i,j))
51    ENDDO
52  ENDDO
53  cond = MAXVAL(y)
54  #IFDEF _OpenACC
55    #IF defined LOO
56      !$ACC END PARALLEL LOOP
57    #ELIF defined KER
58      !$ACC END KERNELS
59    #ENDIF
60  #ENDIF
61  IF (cond.GT.0.5)THEN
62    WRITE(*,10) cond
63  ENDIF
64  10    FORMAT('There is an error!',2x,'Max is=',F19.12)
65  ENDSUBROUTINE iov
```

Now, the compiler generates a vectorized version of the inner loop, along with vectorizing the required reduction for the termination condition.

```
iov:
     47, Loop not fused: different loop trip count
     48, Generated vector simd code for the loop
     53, maxval reduction inlined
         Loop not fused: function call before adjacent loop
         Generated vector simd code for the loop containing reductions
```

The primary issue arising from incorporating debugging options within the loop becomes more pronounced when compiling the code for GPU execution. On the GPU, calling the compiler runtime function is unsupported. Hence, all instances of "PRINT" and "WRITE" must be positioned outside the computational kernel. Conversely, in the second version, the "KERNELS" construct effectively collapses and parallelizes the loops, as indicated below:

```
iov:
     44, Generating present(x(:,:),z(:,:),y(:,:))
     47, Loop is parallelizable
     48, Loop is parallelizable
         Generating Tesla code
         47, !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
         48,   ! blockidx%x threadidx%x auto-collapsed
     53, maxval reduction inlined
         Loop is parallelizable
         Generating Tesla code
         53,   ! blockidx%x threadidx%x auto-collapsed
             !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
             Generating implicit reduction(max:y$r)
```

3.2 Procedures

Calling a procedure (function or subroutine) within a loop hinders the compiler's ability to vectorize the loop. An example is provided below, demonstrating a situation where the "IOIPSL" routine is called within a loop. If a certain condition is met, the "IOIPSL" routine will terminate the code.

```
70   SUBROUTINE pronov(x, y, z, n, m)
71   DIMENSION z(n,m), y(n,m), x(n,m)
72   DO j = 1, m
73     DO i = 1, n
74      x(i,j) = 2.*x(i,j)*z(i,j) + x(i,j)*x(i,j) + z(i,j)*z(i,j)
75      y(i,j) = 1.-EXP(-x(i,j))
76      IF (y(i,j).GE.0.5) THEN
77        CALL ipslerr_p(3,'Oops -> An erros is found', &
78             'All values must be smaller than 0.5', &
79             'Check the calculations', '')
80      ENDIF
81     ENDDO
82   ENDDO
83   ENDSUBROUTINE pronov
```

The compiler report indicates:

```
73, Loop not vectorized/parallelized: contains call
```

We can employ a similar approach as previously mentioned to facilitate loop vectorization. The call can be positioned after the loop (if feasible).

```
 88   SUBROUTINE proyv(x, y, z, n, m)
 89   DIMENSION z(n,m), y(n,m), x(n,m)
 90   #IFDEF _OpenACC
 91     #IF defined LOO
 92       !$ACC PARALLEL LOOP COLLAPSE(2) PRESENT(x,y,z)
 93     #ELIF defined KER
 94       !$ACC KERNELS PRESENT(x,y,z)
 95     #ENDIF
 96   #ENDIF
 97   DO j = 1, m
 98     DO i = 1, n
 99      x(i,j) = 2.*x(i,j)*z(i,j) + x(i,j)*x(i,j) + z(i,j)*z(i,j)
100      y(i,j) = 1.-EXP(-x(i,j))
101     ENDDO
102   ENDDO
103   cond = MAXVAL(y)
104   #IFDEF _OpenACC
105     #IF defined LOO
106       !$ACC END PARALLEL LOOP
107     #ELIF defined KER
108       !$ACC END KERNELS
109     #ENDIF
110   #ENDIF
111   IF (cond.GE.0.5) THEN
112     CALL ipslerr_p(3,'Oops -> An erros is found', &
113           'All values must be smaller than 0.5', &
114           'Check the calculations', '')
115   ENDIF
116   ENDSUBROUTINE proyv
```

In this scenario, the compiler performs subroutine inlining and successfully vectorizes the loop, as indicated by the compiler message.

```
     23, proyv inlined, size=25, file Vectorization.f90 (88)
          97, Loop not fused: different loop trip count
          98, Generated vector simd code for the loop
              FMA (fused multiply-add) instruction(s) generated
          99, Loop not fused: function call before adjacent loop
              Generated vector simd code for the loop containing reductions
```

By implementing this adjustment, the code becomes well-suited for GPU execution as well. The loop is executed on the GPU, and a scalar variable can be conveniently copied from the GPU to the CPU. The stop condition is examined, and if the condition evaluates to true, the code halts.

```
proyv:
     94, Generating present(x(:,:),z(:,:),y(:,:))
     97, Loop is parallelizable
     98, Loop is parallelizable
         Generating Tesla code
         97, !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
         98,   ! blockidx%x threadidx%x auto-collapsed
    103, maxval reduction inlined
         Loop is parallelizable
         Generating Tesla code
        103,   ! blockidx%x threadidx%x auto-collapsed
             !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
             Generating implicit reduction(max:y$r)
```

```
iov:
     44, Generating present(x(:,:),z(:,:),y(:,:))
     47, Loop is parallelizable
     48, Loop is parallelizable
         Generating Tesla code
         47, !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
         48,   ! blockidx%x threadidx%x auto-collapsed
     53, maxval reduction inlined
         Loop is parallelizable
         Generating Tesla code
         53,   ! blockidx%x threadidx%x auto-collapsed
             !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
             Generating implicit reduction(max:y$r)
```

I/O

To optimize vectorization when running on the CPU and parallelization when on the GPU, it is advisable to refrain from embedding any I/O procedures and debugging within the loops.

3.3 Function

Functions can be called within loops while maintaining loop vectorization, employing three approaches: (i) calling an external function with enforced inlining, (ii) encapsulating the function in an external file and including it within the main program, and (iii) defining the function as an internal entity within the program. These techniques are illustrated in the following example:

```
134  SUBROUTINE profyv1(x, y, z, n, m)
135  DIMENSION z(n,m), y(n,m), x(n,m)
136  #IFDEF _OPENACC
137    #IF defined LOO
138      !$ACC PARALLEL LOOP COLLAPSE(2) DEFAULT(PRESENT)
139    #ELIF defined KER
140      !$ACC KERNELS DEFAULT(PRESENT)
141    #ENDIF
142  #ENDIF
143  DO j = 1, m
144    DO i = 1, n
145     a = 2.*x(i,j)*z(i,j) + x(i,j)*x(i,j) + z(i,j)*z(i,j)
146     y(i,j) = funv(a)
147    ENDDO
148  ENDDO
149  #IFDEF _OPENACC
150    #IF defined KER
151      !$ACC END KERNELS
152    #ENDIF
153  #ENDIF
154  ENDSUBROUTINE profyv1
155
156  SUBROUTINE profyv2(x, y, z, n, m)
157  DIMENSION z(n,m), y(n,m), x(n,m)
158  #INCLUDE "func.h"
159  #IFDEF _OPENACC
160    #IF defined LOO
161      !$ACC PARALLEL LOOP COLLAPSE(2) DEFAULT(PRESENT)
162    #ELIF defined KER
163      !$ACC KERNELS DEFAULT(PRESENT)
164    #ENDIF
```

```
165  #ENDIF
166  DO j = 1, m
167    DO i = 1, n
168     a = 2.*x(i,j)*z(i,j) + x(i,j)*x(i,j) + z(i,j)*z(i,j)
169     y(i,j) = fu(a)
170    ENDDO
171  ENDDO
172  #IFDEF _OPENACC
173    #IF defined KER
174      !$ACC END KERNELS
175    #ENDIF
176  #ENDIF
177  ENDSUBROUTINE profyv2
178
179  SUBROUTINE profyv3(x, y, z, n, m)
180  DIMENSION z(n,m), y(n,m), x(n,m)
181  fu(t) = 1.-EXP(-t)
182  #IFDEF _OPENACC
183    #IF defined LOO
184      !$ACC PARALLEL LOOP COLLAPSE(2) DEFAULT(PRESENT)
185    #ELIF defined KER
186      !$ACC KERNELS DEFAULT(PRESENT)
187    #ENDIF
188  #ENDIF
189  DO j = 1, m
190    DO i = 1, n
191     a = 2.*x(i,j)*z(i,j) + x(i,j)*x(i,j) + z(i,j)*z(i,j)
192     y(i,j) = fu(a)
193    ENDDO
194  ENDDO
195  #IFDEF _OPENACC
196    #IF defined KER
197      !$ACC END KERNELS
198    #ENDIF
199  #ENDIF
200  ENDSUBROUTINE profyv3
201
202  SUBROUTINE profyv4(x, y, z, n, m)
203  !$ACC ROUTINE(funva) vector
204  DIMENSION z(n,m), y(n,m), x(n,m), a(n)
205  #IFDEF _OPENACC
206    #IF defined LOO
207      !$ACC PARALLEL LOOP GANG PRESENT(x,y,z) CREATE(a)
208    #ELIF defined KER
209      !$ACC KERNELS PRESENT(x,y,z) CREATE(a)
210    #ENDIF
211  #ENDIF
212  DO j = 1, m
213    a(:) = 2.*x(:,j)*z(:,j) + x(:,j)*x(:,j) + z(:,j)*z(:,j)
214    y(:,j) = funva(a(:),n)
215  ENDDO
216  #IFDEF _OPENACC
217    #IF defined KER
218      !$ACC END KERNELS
219    #ENDIF
220  #ENDIF
221  ENDSUBROUTINE profyv4
```

Upon compiling the code for CPU execution, even when employing the aggressive inline option, the first version of the code yields the following message:

```
profyv1:
    144, Loop not vectorized/parallelized: contains call
    145, FMA (fused multiply-add) instruction(s) generated
profyv2:
    166, FMA (fused multiply-add) instruction(s) generated
    167, Generated vector simd code for the loop
         FMA (fused multiply-add) instruction(s) generated
profyv3:
    189, FMA (fused multiply-add) instruction(s) generated
    190, Generated vector simd code for the loop
         FMA (fused multiply-add) instruction(s) generated
profyv4:
    212, Loop not vectorized/parallelized: contains call
         FMA (fused multiply-add) instruction(s) generated
    213, Loop not fused: function call before adjacent loop
         Generated vector simd code for the loop
         FMA (fused multiply-add) instruction(s) generated
```

As evident from the results, the compiler does not vectorize loop 144. The function is called n times, corresponding to the number of iterations in the loops. The cumulative execution time is also displayed in the performance profile. Now, let's explore the compiler's behavior when we incorporate the function into the program.

There are distinct advantages to employing the "INCLUDE" approach. Firstly, the compiler vectorizes the most interior loop (167), resulting in an execution time that is nearly one-third of the previous case. However, an even more optimized situation arises when defining the function as an internal function within the code. Here, the compiler vectorizes the most inner loop, leading to a reduced execution time compared to the "INCLUDE" approach.

Furthermore, we should consider the function call scenario when the function returns an array instead of a scalar. In this scenario, the number of function calls is reduced, potentially leading to improved performance. Remarkably, the compiler manages to vectorize the inner loop in this case, and the operations within the function are also optimized.

In terms of performance, the most favorable outcome is achieved by explicitly incorporating the function within the routine. The second-best approach is employing the "INCLUDE" method. Notably, calling an external function in the vectorized case (profyv4) demonstrates enhanced performance compared to calling an external scalar function.

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 35.57      0.27     0.27 134217728     0.00     0.00  functions_funv_
 22.40      0.44     0.17        1   170.22   440.57  vectorial_profyv1_
 10.54      0.52     0.08     4096     0.02     0.02  functions_funva_
 10.54      0.60     0.08        1    80.10    80.10  vectorial_profyv2_
  9.22      0.67     0.07        1    70.09   150.19  vectorial_profyv4_
  6.59      0.72     0.05        1    50.06    50.06  vectorial_profyv3_
  5.27      0.76     0.04                             functions_
  0.00      0.76     0.00        1     0.00     0.00  MAIN_
  0.00      0.76     0.00        1     0.00   720.93  vectorial_vec_
```

When compiling the code for GPU execution, the benefits of employing "INCLUDE" or an internal function over the CALL become even more evident. Calling a procedure within a GPU kernel necessitates the explicit definition of the level of parallelism for the procedure. This entails specifying how the loops are to be distributed across the GPU threads. In the provided example, the two loops are parallelized on the GPU using the "GANG" and "VECTOR" levels of parallelism (representing two layers of parallel execution on the GPU). Therefore, we guide the compiler to utilize a sequential version of the function, as demonstrated below:

```
1 MODULE functions
2   CONTAINS
3
```

```
 4   FUNCTION funv(x)
 5   #IFDEF _OpenACC
 6     !$ACC ROUTINE SEQ
 7   #ENDIF
 8   funv = 1.-EXP(-x)
 9   ENDFUNCTION funv
10
11   FUNCTION funva(x,n)
12   #IFDEF _OpenACC
13     !$ACC ROUTINE vector
14   #ENDIF
15   DIMENSION x(n), funva(n)
16   #IFDEF _OpenACC
17     !$ACC LOOP
18   #ENDIF
19   DO i= 1, n
20     funva(i) = 1.-EXP(-x(i))
21   ENDDO
22   ENDFUNCTION funva
23
24 ENDMODULE functions
```

By providing this clarification to the compiler, the execution times for the first three versions will converge to nearly identical values when employing either the PARALLEL or KERNELS constructs. The compiler treats all three versions similarly due to the specified adjustments:

```
profyv1:
    138, Generating Tesla code
        143, !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
        144,   ! blockidx%x threadidx%x collapsed
    138, Generating default present(x(:n,:m),z(:n,:m),y(:n,:m))
profyv2:
    161, Generating Tesla code
        166, !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
        167,   ! blockidx%x threadidx%x collapsed
    161, Generating default present(x(:n,:m),z(:n,:m),y(:n,:m))
profyv3:
    184, Generating Tesla code
        189, !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
        190,   ! blockidx%x threadidx%x collapsed
    184, Generating default present(x(:n,:m),z(:n,:m),y(:n,:m))
profyv4:
    207, Generating present(x(:,:),z(:,:),y(:,:))
         Generating create(a(:)) [if not already present]
         Generating Tesla code
        212, !$acc loop gang ! blockidx%x
        213, !$acc loop seq
    213, Loop is parallelizable
```

When executing on a GPU, it's necessary to explicitly denote the vectorial operation within the function. Despite this declaration, the performance doesn't match that of the previous cases.

```
   Time(%)  Total Time (ns)                      Name
0  -------  ---------------  -------------------------
1     91.4      432,876,147  vectorial_profyv4_207_gpu
2      2.9       13,518,577  vectorial_profyv1_138_gpu
3      2.9       13,496,114  vectorial_profyv2_161_gpu
4      2.9       13,495,954  vectorial_profyv3_184_gpu
```

Function

To enhance vectorization during CPU execution and parallelization on a GPU, it is advisable to refrain from calling external functions within loops. Instead, consider utilizing "INCLUDE" or internal functions.

3.4 Indirect addressing

Indirect addressing involves using a variable or pointer that holds the memory address. This scenario arises when dealing with an unstructured mesh or sorting an array, where memory access tends to be non-sequential. This poses a challenge to the compiler since it cannot predict the memory addresses to be accessed due to potential randomness, striding, proximity, or distance. Such access patterns often lead to suboptimal performance. To address this issue, one potential solution involves incorporating a contiguous dimension within the indirect addressing. By doing so, a significant portion of memory accesses can remain contiguous, thereby improving performance. The following example illustrates this concept:

```
268   SUBROUTINE indirect1(x, y, z, m)
269   DIMENSION z(m), y(m), x(m), yy(m)
270   DIMENSION inx(m)
271
272   inx = Shuffle(m)
273   DO j = 1, m
274     y(inx(j)) = 2.*x(j)*z(inx(j)) + x(j)*x(j) + z(inx(j))*z(inx(j))
275   ENDDO
276   ENDSUBROUTINE indirect1
277
278   SUBROUTINE indirect2(x, y, z, n, m)
279   DIMENSION z(n,m), y(n,m), x(n,m), yy(n,m)
280   DIMENSION inx(m)
281
282   inx = Shuffle(m)
283   DO j = 1, m
284     DO i = 1, n
285       y(i,inx(j)) = 2.*x(i,j)*z(i,inx(j)) + x(i,j)*x(i,j) + z(i,inx(j))*z(i,inx(j))
286     ENDDO
287   ENDDO
288   ENDSUBROUTINE indirect2
```

In the provided example, memory access occurs randomly due to the shuffled indexes. In the first version, the loop does not get vectorized by the compiler because the array reference does not have a stride-1 pattern, which implies that threads will not access contiguous data from the arrays. However, introducing an extra dimension as the leftmost dimension in the array preserves memory contiguity, allowing the compiler to utilize vectorization.

```
indirect1:
    272, Memory copy idiom, loop replaced by call to __c_mcopy8
    273, Loop not vectorized: non-stride-1 array reference
         Loop unrolled 2 times
         FMA (fused multiply-add) instruction(s) generated
indirect2:
    282, Memory copy idiom, loop replaced by call to __c_mcopy8
    283, Loop not fused: function call before adjacent loop
         FMA (fused multiply-add) instruction(s) generated
    284, Generated vector simd code for the loop
         FMA (fused multiply-add) instruction(s) generated
```

Indirect addressing

Reduce indirect addressing by introducing an extra dimension that ensures contiguous memory access.

3.5 IFs

Straightforward computations within simple "DO" loops can often be efficiently mapped to vector instructions. However, the presence of branching and conditionals, such as an "IF" statement within a loop, can hinder vectorization. This is because vector instructions struggle to represent branching operations. Nevertheless, compilers can sometimes circumvent this limitation by employing masked assignments for basic conditionals. In this approach, the compiled code employs vector instructions to execute both branches of the conditional ("THEN" and "ELSE" clauses). The branching condition itself is also evaluated using vectorized operations. Ultimately, a mask is applied during the final assignment to determine which results to retain from each branch. Despite this masking technique, loops containing "IF" statements tend to have longer execution times. Let's explore this with an example:

```
215  SUBROUTINE ifsindo1(x, y, z, n, m)
216  DIMENSION z(n,m), y(n,m), x(n,m)
217  #IFDEF _OPENACC
218    #IF defined LOO
219      !$ACC PARALLEL LOOP COLLAPSE(2) PRESENT(x,y,z)
220    #ELIF defined KER
221      !$ACC KERNELS PRESENT(x,y,z)
222    #ENDIF
223  #ENDIF
224  DO j = 1, m
225    DO i = 1, n
226     x(i,j) = x(i,j) + z(i,j)
227     y(i,j) = 0.5 - x(i,j)
228     IF (y(i,j).LT.0.) THEN
229       y(i,j) = EXP(y(i,j))
230     ELSE
231       y(i,j) = EXP(-y(i,j))
232     ENDIF
233    ENDDO
234  ENDDO
235  #IFDEF _OPENACC
236    #IF defined KER
237      !$ACC END KERNELS
238    #ENDIF
239  #ENDIF
240  ENDSUBROUTINE ifsindo1
241
242  SUBROUTINE ifsindo2(x, y, z, n, m)
243  DIMENSION z(n,m), y(n,m), x(n,m)
244  #IFDEF _OPENACC
245    #IF defined LOO
246      !$ACC PARALLEL LOOP COLLAPSE(2) PRESENT(x,y,z)
247    #ELIF defined KER
248      !$ACC KERNELS PRESENT(x,y,z)
249    #ENDIF
250  #ENDIF
251  DO j = 1, m
252    DO i = 1, n
253     x(i,j) = x(i,j) + z(i,j)
254     y(i,j) = 0.5 - x(i,j)
255     y(i,j) = EXP(-ABS(y(i,j)))
256    ENDDO
```

```
257   ENDDO
258   #IFDEF _OPENACC
259     #IF defined KER
260       !$ACC END KERNELS
261     #ENDIF
262   #ENDIF
263   ENDSUBROUTINE ifsindo2
```

In the first scenario, the compiler managed to vectorize loop 251 even in the presence of the "IF"' condition. However, it's often possible for developers to circumvent the need for conditional statements through the use of mathematical expressions. In the provided example, the inclusion of the conditional statement led to a nearly twofold slowdown in the code's performance.

```
ifsindo1:
    225, Generated vector simd code for the loop
ifsindo2:
    252, Generated vector simd code for the loop

  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 79.27      0.19     0.19        1   190.25   190.25  vectorial_ifsindo1_
 20.86      0.24     0.05        1    50.07    50.07  vectorial_ifsindo2_
  0.00      0.24     0.00        1     0.00     0.00  MAIN_
  0.00      0.24     0.00        1     0.00   240.31  vectorial_vec_
```

Similar behavior is observed when running the code on a GPU. Despite the compiler's efforts to parallelize both versions, the version without branching achieves faster execution on the GPU. The compiler report and execution times for the two versions are provided below:

```
ifsindo1:
    219, Generating present(x(:,:),z(:,:),y(:,:))
         Generating Tesla code
        224, !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
        225,   ! blockidx%x threadidx%x collapsed
ifsindo2:
    246, Generating present(x(:,:),z(:,:),y(:,:))
         Generating Tesla code
        251, !$acc loop gang, vector(128) collapse(2) ! blockidx%x threadidx%x
        252,   ! blockidx%x threadidx%x collapsed

   Time(%)  Total Time (ns)                        Name
0  -------  ---------------  --------------------------
1     65.4       22,132,737  vectorial_ifsindo1_219_gpu
2     34.6       11,716,530  vectorial_ifsindo2_246_gpu
```

IFs

Avoid introducing branches within loops whenever possible. Aim to maintain a single, coherent control flow within the loop.

3.6 Dependence

Loop-carried dependence refers to a situation where the execution of iterations in a loop is influenced by the values computed in previous iterations, e.g., in time/space marching. It arises when the correct order of execution is necessary to obtain accurate results, and changing the order could lead to incorrect or unpredictable outcomes. One example is given below:

```
293   DO j = 1, m
294     y(j) = y(j-1) + h * fu(x(j-1))
295   ENDDO
```

The compiler message indicates:

```
    293, Loop not vectorized: data dependency
         Loop unrolled 2 times
         FMA (fused multiply-add) instruction(s) generated
```

A potential solution for this issue involves precomputing all major calculations into a temporary array using SIMD processing. Then, isolate the sequential operations to the minimum workload feasible, as depicted below:

```
301   DO j = 1, m
302     t(j) = fu(x(j-1)) * h
303   ENDDO
304   DO j = 1, m
305     y(j) = y(j-1) + t(j)
306   ENDDO
```

In which the compiler message indicates:

```
    301, Loop not fused: dummy arguments read/write - possible conflict; try
        -Mvect=fuse
         Loop not fused: unsafe variable with target attribute
         Generated vector simd code for the loop
    304, Loop not fused: function call before adjacent loop
         Loop not vectorized: data dependency
         Loop unrolled 4 times
```

Dependence

Minimize the sequential workload to the smallest workload feasible.

4 Management of memory

Fortran provides several mechanisms for managing memory, especially through intrinsic functions, array declarations, and dynamic memory allocation. Here are some key aspects of memory management in Fortran.

4.1 Dynamic memory allocation

Dynamic memory allocation in Fortran allows for precise control over the creation and destruction of workspaces. This can be achieved by declaring allocatable arrays, which are essentially placeholders without assigned memory until they are specifically allocated within the program or subprogram. Once their purpose is fulfilled, these arrays can be deallocated to free up memory resources.

```
REAL(KIND=dp_t), DIMENSION(:,:), ALLOCATABLE :: array
!!
ALLOCATE(array(n1,m1))
!!
DEALLOCATE(array)
!!
ALLOCATE(array(n2,m2))
```

```
!!
DEALLOCATE(array)
```

4.2 Pointers

Pointer variables provide a higher level of control, offering even more flexibility than allocatable arrays. Similar to allocatable arrays, pointers can be allocated memory as needed. However, pointers offer an added advantage – they can also serve as references to existing targets that are named separately. This allows pointers to act as dynamic aliases for other variables and arrays, enhancing their versatility.

Consider a practical scenario involving the "Diffusive" problem in a one-dimensional context. In this context, a tridiagonal system of equations can be efficiently solved. However, the complexity increases when dealing with implicit problems in two-dimensional or three-dimensional settings, where the coefficient matrix takes on a Pentadiagonal or Heptadiagonal form. To address this challenge, an effective strategy is to employ the alternating direction implicit (ADI) method. This approach transforms the original problem into a sequence of tridiagonal systems for each direction: x, y, and z. Nonetheless, when utilizing ADI, it's necessary to update the coefficient matrix for each direction. To facilitate this process, a promising approach is to utilize pointer variables to represent the coefficients. This allows for greater flexibility in handling the evolving coefficient matrices in different directions.

```
REAL(KIND=dp_t), POINTER, DIMENSION (:) :: a => NULL(), b => NULL(), c => NULL(), g
    => NULL(), x => NULL()
```

Then we can update the coefficients as follows. The coefficients are in principle different in shape.

```
TYPE(gridvar_t), INTENT(INOUT), TARGET :: gridvar
!!
     ! X Sweep
     DO j=2,grid%n2
       a => gridvar%data(2:grid%n1,j,1 )
       b => gridvar%data(2:grid%n1,j,2 )
       c => gridvar%data(2:grid%n1,j,3 )
       x => gridvar%data(2:grid%n1,j,12)
       g => gridvar%data(2:grid%n1,j,7 )
       CALL trdiag(grid%n1-1,a,b,c,x,g)
       gridvar%data(2:grid%n1,j,12) = x(:)
     ENDDO
     ! Y Sweep
     DO i=2,grid%n1
       a => gridvar%data(i,2:grid%n2,4 )
       b => gridvar%data(i,2:grid%n2,5 )
       c => gridvar%data(i,2:grid%n2,6 )
       x => gridvar%data(i,2:grid%n2,11)
       g => gridvar%data(i,2:grid%n2,8 )
       CALL trdiag (grid%n2-1,a,b,c,x,g)
       gridvar%data(i,2:grid%n2,11) = x(:)
     ENDDO
```

Management of memory

Opt for pointer variables instead of array copying. Pointers provide the most effective approach in Fortran for array manipulation.

5 Errors in floating-point computations

Errors in floating-point computations can arise due to the limitations of representing real numbers with finite precision in binary. Here are some common types of errors:

- Truncation error: when a continuous mathematical process is approximated by a discrete process, leading to a difference between the exact mathematical solution and the approximate numerical solution
- Rounding Errors: Floating-point numbers can't represent all real numbers exactly, leading to rounding errors when converting between decimal and binary representations.
- Overflow and Underflow Errors: When the magnitude of a number exceeds the range that can be represented, overflow occurs (resulting in infinity), or underflow occurs (leading to zero or denormalized numbers).
- Cancellation Errors: Subtracting two nearly equal numbers can lead to a significant loss of precision, as the significant digits "cancel out."
- Precision Errors: Repeated arithmetic operations can lead to loss of precision. The more operations you perform, the more the result might deviate from the actual value.
- Representation Errors: Some decimal numbers can't be represented exactly in binary, resulting in small discrepancies between the actual value and its floating-point representation.
- Propagation of Errors: Errors can propagate through calculations, potentially magnifying inaccuracies in complex computations or iterative algorithms.
- Loss of Associativity: Floating-point arithmetic doesn't always follow the associative property due to rounding. Changing the order of operations might yield slightly different results.
- Numerical Instability: Certain algorithms can amplify small errors, causing inaccurate results. This is particularly relevant in iterative methods and simulations.

Here are some tips to avoid errors in floating-point computations:

- Choose the Right Data Type: Select the appropriate floating-point data type based on the required precision and range. For example, consider using REAL for faster computation and REAL(8) or DOUBLE PRECISION for higher precision.
- Understand the Algorithms: Be aware of the numerical characteristics of the algorithms you're using. Some algorithms are more prone to numerical instability or cancellation than others.
- Minimize Accumulated Errors: Avoid excessive accumulation of rounding errors by reordering operations or using compensated algorithms to reduce the loss of precision.
- Utilize established libraries and programming languages that provide robust implementations of floating-point arithmetic.
- Normalize Inputs: When possible, normalize inputs to a range that minimizes precision loss. This can help prevent underflow or overflow.
- Avoid Subtraction of Nearly Equal Numbers: In situations where subtracting nearly equal numbers is necessary, consider using alternative methods that preserve precision.
- Perform Error Analysis: Analyze the error propagation in your computations. This can help you anticipate where errors might accumulate and take measures to mitigate them.
- Use Robust Algorithms: Some algorithms are designed to be more robust in the presence of floating-point errors. Research and choose algorithms that are well-suited for your specific problem.
- Implement Error Handling: Handle exceptional cases like overflow, underflow, and NaN values gracefully in your code. Make sure your program doesn't crash due to these scenarios.

- Avoid Unnecessary Conversions: Minimize conversions between floating-point and integer types, as they can introduce rounding errors.
- Keep Scale in Mind: Scaling your problem to a range that works well with the floating-point format can help maintain accuracy.

References

[1] NVIDIA HPC Compilers Reference Guide; 2023.

[2] Openacc programming and best practices guide; 2021.

[3] Brainerd WS. Guide to Fortran 2003 programming. Berlin: Springer; 2009 Jun 11.

[4] Press WH, Teukolsky SA, Vetterling WT, Flannery BP. Numerical recipes in Fortran 90 the art of parallel scientific computing. Cambridge university press; 1996 Sep 1.

[5] Dongarra J, Foster I, Fox G, Gropp W, Kennedy K, Torczon L, White A. Sourcebook of parallel computing. San Francisco: Morgan Kaufmann Publishers; 2003 Jan 1.

[6] User Notes On Fortran Programming, An Open Cooperative Practical Guide; 1998.

[7] Ryad El Khatib. General Fortran Optimizations Guide. Meteo-france - Cnrm/gmap; 2019.

[8] Piper C. An Introduction to Vectorization with the Intel Fortran Compiler; 2012.

[9] Pilla LL. Basics of Vectorization for Fortran Applications (Doctoral dissertation, Inria Grenoble Rhone-Alpes); 2018.