Codee Training Series

April 25-26, 2023

Shift Left Performance

Automated Code inspection for Performance

April 24th, 2023

ZPIC: The code “em2d”

Website: https://zpic-plasma.github.io/ [last checked: Apr 2023]
Github: https://github.com/zpic-plasma/zpic [last checked: Apr 2023]

- Particle-in-Cell (PIC) codes are used in almost all areas of plasma physics, such as fusion energy research, plasma accelerators, space physics, ion propulsion, and plasma processing, and many other areas.
- ZPIC is a suite of 1D/2D fully relativistic electromagnetic PIC codes, as well as 1D electrostatic.
- Code written in the C programming language.
- The directory structure is organized as follows:
  - em1d - 1D electromagnetic (finite difference)
  - em1ds - 1D electromagnetic (spectral)
  - em2d - 2D electromagnetic (finite difference)
  - em2ds - 2D electromagnetic (spectral)
  - es1d - 1D electrostatic
  - mods - Modified versions of the base codes
  - python - Python interface to ZPIC codes

Benchmark “ZPIC/em2d”

14 files
193 functions
141 loops
4035 lines of code (SLOC)

Build, run & verify ZPIC with Weibel workload

```
$ cd em2d-0-serial
$ make clean
$ make CC=gcc CFLAGS="-Ofast"
$ ./zpic
Starting simulation ...

n = 0, t = 0.000000
n = 1, t = 0.070000
n = 2, t = 0.140000
n = 3, t = 0.210000
...
n = 497, t = 34.790001
n = 498, t = 34.860001
n = 499, t = 34.930000
n = 500, t = 35.000000

Simulation ended.

Time for spec. advance = 10.339094 s
Time for emf   advance = 0.066140 s
Total simulation time  = 10.431528 s

Particle advance [nsec/part] = 78.723580
Particle advance [Mpart/sec] = 12.702674
```

Experimental setup of ZPIC:
- Weibel
- Grid 128x128 cells
- 500 time-steps
- 8 particles per cell
- uniform distribution

```
$ cat -n input/simulation_test.c
     1  /**
     2   * ZPIC - em2d
     3   *
     4   * Weibel instability
     5   */
     6
     7  #include <stdlib.h>
     8  #include "../simulation.h"
     9
    10  void sim_init( t_simulation* sim ){
    11
    12              // Time step
    13              float dt = 0.07;
    14              float tmax = 35.0;
    15
    16              // Simulation box
    17              int   nx[2]  = { 128, 128 };
    18              float box[2] = { 12.8, 12.8 };
    19
    20              // Diagnostic frequency
    21              int ndump = 10;
    22
    23          // Initialize particles
    24              const int n_species = 2;
    25              t_species* species = (t_species *) malloc( n_species * sizeof( t_species ));
    26
    27              // Use 4x2 particles per cell
    28              int ppc[] = {4,2};
    29
    30              // Initial fluid and thermal velocities
    31              t_part_data ufl[] = { 0.0, 0.0, 0.6 };
    32              t_part_data uth[] = { 0.1, 0.1, 0.1 };
    33
    34              spec_new( &species[0], "electrons", -1.0, ppc, ufl, uth, nx, box, dt, NULL );
    35
    36              ufl[2] = -ufl[2];
    37              spec_new( &species[1], "positrons", +1.0, ppc, ufl, uth, nx, box, dt, NULL );
    38
    39              // Initialize Simulation data
    40              sim_new( sim, nx, box, dt, tmax, ndump, species, n_species );
    41
    42  }
    43
    44  void sim_report( t_simulation* sim ){
    45
    46              // All magnetic field components
    47              emf_report( &sim->emf, BFLD, 0 );
    48              emf_report( &sim->emf, BFLD, 1 );
    49              emf_report( &sim->emf, BFLD, 2 );
    50
    51  }
```

Profiling: Find the hotspots of ZPIC/em2d

```
// Build and add profiling
// flag -pg to the ZPIC code
$ make CC=gcc CFLAGS="-Ofast -pg" clean zpic

// Run the ZPIC code
$ ./zpic

// Output gprof information
$ gprof zpic
```

Gprof flat profile

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 55.80      7.96     7.96     1002     7.95    14.12  spec_advance
 22.43     11.16     3.20 131334144     0.00     0.00  interpolate_fld
 19.00     13.87     2.71 131334144     0.00     0.00  dep_current_zamb
  1.89     14.14     0.27       62     4.36     4.36  spec_sort
```

Gprof call graph for function *spec_advance()*

```
index % time    self  children    called     name
                7.96    6.18    1002/1002        sim_iter [2]
[1]     99.1    7.96    6.18    1002         spec_advance [1]
                3.20    0.00 131334144/131334144     interpolate_fld [3]
                2.71    0.00 131334144/131334144     dep_current_zamb [4]
                0.27    0.00      62/62          spec_sort [5]
                0.00    0.00    2004/3006        timer_ticks [13]
                0.00    0.00    1002/1504        timer_interval_seconds [15]
```

Profiling: Find the hotspots of ZPIC/em2d (cont'd)

- The profiler *gprof* reports hotspots at the function-level
  - ie. functions are ranked by % time consumed.
- We **need more information to select the target hotspots**.
- We **need to analyze the function/procedure calls and the loops altogether** (at least).

particle.c:889:spec_advance() **[55.80%]**
-> contains 1+ loops with function calls

→ **This loop in *spec_advance()* is a good parallelization opportunity (coarse-grain parallelism).**

-> calls particle.c:840:interpolate_fld() **[22.43%]**
-> does not contain any loops

→ **This hotspot *interpolate_fld()* does not contain loops, so it is not a (loop-level) parallelization opportunity.**

-> calls particle.c:605:dep_current_zamb() **[19.00%]**
-> contains 1 loop without function calls

→ **This loop in *dep_current_zamb()* is a good parallelization opportunity (fine-grain parallelism).**

- The **selected hotspot is loop at line 677 in function *spec_advance()*** **[up to 97.23% of the runtime]**
  - Note it includes runtime of the other two hotspots as well.
  - Ignore *interpolate_fld()* because it does not contain any loop.
  - Try to parallelize *dep_current_zamb()*, but you will slow down ZPIC (65M calls at ~0 seconds/call).

Parallel ZPIC/em2d: from serial to CPU & GPU...

EM2D #7a (Basic GPU+OpenACC Atomic)

EM2D #6a (Basic GPU+OpenMP Atomic)

EM2D #5a (Basic CPU+OpenMP Atomic)

EM2D #4 (loop fission)

EM2D #3 (change Array of Structs by separate arrays)

EM2D #2 (inlining deb_current_zamb)

EM2D #1 (outlining, gather-scatter data)

EM2D #0 (serial version)

EM2D #0 (serial version)

- Start by running *pwreport --screening* to **get an assessment of the hotspot function**.
- Reported a total of 3 checks, but none of the related to parallelism (Vector, Multi and Offload).

```
$ cd em2d-0-serial/

$ pwreport --screening --include-tags all particles.c:spec_advance --brief -- -Ofast -lm
[C] target compiler: <none> (Compiler Agnostic Mode)

SCREENING REPORT

Target                  Lines of code Optimizable lines Analysis time # checks Effort Cost   Profiling
----------------------- ------------- ----------------- ------------- -------- ------ ------ ---------
particles.c:spec_advance 749           6                 131 ms        3        9 h    294€   n/a
----------------------- ------------- ----------------- ------------- -------- ------ ------ ---------
Total                   749           6                 131 ms        3        9 h    294€   n/a

CHECKS PER STAGE OF THE PERFORMANCE OPTIMIZATION ROADMAP
Target                  Scalar Control Memory Vector Multi Offload
----------------------- ------ ------- ------ ------ ----- -------
particles.c:spec_advance 1      0       2      0      0     0
----------------------- ------ ------- ------ ------ ----- -------
Total                   1      0       2      0      0     0

TOTAL NUMBER OF LOOPS FOR CODEE AUTO AND GUIDED MODES
                    ---------------- # checks ----------------
Codee mode # Loops Scalar Control Memory Vector Multi Offload
---------- ------- ------ ------- ------ ------ ----- -------
Auto       0       0      0       0      0      0     0
Guided     1       0      0       1      0      0     0

1 file successfully analyzed and 0 failures in 131 ms
```

EM2D #0 (serial version)

- Let's run *pwreport --non-analyzable* to get a report of the challenges identified by Codee in the code.
- We need to remove the unsupported C programming language feature from the source code.
  - Usage of structs
  - Calls to functions/procedures

```
$ cd em2d-0-serial/

$ pwreport --non-analyzable particles.c:spec_advance -- -Ofast -lm
Compiler flags: -Ofast -lm

[C] target compiler: <none> (Compiler Agnostic Mode)

Count Type
----- ----------------------------------------------------------------
1     Unsupported usage of different fields of a struct type variable
1     Potential aliasing in call to a function
1     Unsupported usage of variable inside unstructured code

Count : Number of occurrences of the unsupported feature
Type : Type of unsupported feature

1 file successfully analyzed and 0 failures in 112 ms
```

Parallel ZPIC/em2d: from serial to CPU & GPU...

EM2D #7a (Basic GPU+OpenACC Atomic)

EM2D #6a (Basic GPU+OpenMP Atomic)

EM2D #5a (Basic CPU+OpenMP Atomic)

EM2D #4 (loop fission)

EM2D #3 (change Array of Structs by separate arrays)

EM2D #2 (inlining deb_current_zamb)

EM2D #1 (outlining, gather-scatter)

EM2D #0 (serial version)

EM2D #1 (outlining, gather-scatter)

- Addressing the issues reported by Codee with respect to pointers to structs and function/procedure calls requires significant changes in the source code of ZPIC. It might even require to rethink the data types used across the entire project.
- So **let's adopt an incremental approach that is less intrusive and less time-consuming.**
- We will do the following actions:
  - **First, use the outlining technique** to isolate the hotspot loop at *em2d/particles.c:spec_advance:677* into a separate function that receives all the input data as function parameters (i.e., no global variables are used in the function body).
  - **Second, use the gather-scatter technique** to copy input data and output data into temporary arrays with a contiguous data layout in memory. This copies input data dispersed across memory into consecutive arrays (*gather*), makes the hotspot loop at *em2d/particles.c:spec_advance:677* operate on those temporary consecutive arrays, and finally copies the results into the output data dispersed across memory (*scatter*).
- **Note that these techniques will introduce an overhead that increases the runtime of the sequential ZPIC code and that must be amortized by exploiting parallelism in the hotspot loop** at *em2d/particles.c:spec_advance:677*.

EM2D #1 (outlining, gather-scatter) (cont'd)

- In computing, **outlining** is a manual or compiler optimization that **replaces a code section by a function call site and creates a new function with the code section as function body**.
- Benefits of outlining for the programmer:
  - Improve the readability and the structure of the source code.
  - Enable code transformations that otherwise would not be applicable.
- **In ZPIC we use it to isolate the hotspot loop into a separate function promoting data hiding (i.e., all input data and output data passed as explicit function parameters)**, and increase the code coverage of Codee tools.

Code example

The following code might be obscure for static code analysis tools due to the dynamic allocation of a pointer-based struct data type in the C programming language:

```c
#include <stdlib.h>

typedef struct
{
   double *A;
   double *B;
} data;

void foo()
{
   data *S = (data *)malloc(sizeof(data));
   S->A = (double *)malloc(sizeof(double) * 1000);
   S->B = (double *)malloc(sizeof(double) * 1000);

   for (size_t i = 0; i < 1000; i++)
   {
      S->A[i] = 0;
   }

   free(S->B);
   free(S->A);
   free(S);
}
```

By outlining the loop to a dedicated function (see *bar()* in the code below) the loop can be easily analyzed in isolation because the resulting code is free of uses of the pointer-based struct data type. The resulting code is as follows:

```c
#include <stdlib.h>

typedef struct
{
   double *A;
   double *B;
} data;

void bar(double *A)
{
   for (size_t i = 0; i < 1000; i++)
   {
      A[i] = 0;
   }
}

void foo()
{
   data *S = (data *)malloc(sizeof(data));
   S->A = (double *)malloc(sizeof(double) * 1000);
   S->B = (double *)malloc(sizeof(double) * 1000);

   bar(S->A);

   free(S->B);
   free(S->A);
   free(S);
}
```

EM2D #1 (outlining, gather-scatter) (cont'd)

- Let's **apply outlining to the hotpost loop** *em2d/particles.c:spec_advance:677* of function

```
void spec_advance( t_species* spec, t_emf* emf, t_current* current )
```

And create **a separate function named *spec_advance_outlining()***.

```
void spec_advance_outlining(
   // Spec data
   const int spec_np, const t_part_data spec_q, const t_part_data qnx, const t_part_data qny, const t_part_data tem,
   const t_part_data dt_dx, const t_part_data dt_dy, int *restrict spec_ix, int *restrict spec_iy, t_part_data *restrict spec_x,
   t_part_data *restrict spec_y, t_part_data *restrict spec_ux, t_part_data *restrict spec_uy, t_part_data *restrict spec_uz,
   // Emf data
   const int emf_nrow, t_fld *restrict Ex, t_fld *restrict Ey, t_fld *restrict Ez, t_fld *restrict Bx, t_fld *restrict By,
   t_fld *restrict Bz,
   // Current data
   const int current_nrow, t_fld *restrict current_Jx, t_fld *restrict current_Jy, t_fld *restrict current_Jz)
```

- **This is time-consuming so we provide you the new source code to make better use of your time.**

EM2D #1 (outlining, gather-scatter) (cont'd)

- In computing, **gather-scatter** is a manual optimization that **changes memory addressing enabling to replace indexed read/write operations by consecutive memory accesses**.
- Benefits for the programmer:
  - It is often used in sparse linear algebra operations to gather indexed reads and scatter indexed writes.
  - It is also used to improve locality by enabling loops to benefit from consecutive memory accesses.
  - It is also important in SIMD/vector programming, as SIMD/vector CPUs often provide hardware support for gather-scatter operations.
- **In ZPIC we use it**
  - To "pack" data spread across memory through pointer-to-structs into consecutive arrays
  - And increase the code coverage of Codee tools by promoting data hiding (removing pointer-to-structs).

The code has a vector *y* holding *n* elements stored at memory locations given by *index*, so they are not guaranteed to be at consecutive memory locations. The **gather** copies all those array elements into a consecutive array *x*.

```
for(int i=0; i<n; ++i) {
   x[i] = y[index[i]];
}
```

The code now computes the results using array *x*. At the end, the **scatter** copies the consecutive array entries of *x* into the corresponding entries of *y*.

```
for(int i=0; i<n; i++) {
   y[index[i]] = x[i];
}
```

EM2D #1 (cont'd)

- Let's **apply gather-scatter to the hotpost loop**
  em2d/particles.c:spec_advance:677
- The following pointers-to-struct are used in the source code:

  t_species (/ particles.h */)*
  t_emf (/ emf.h */)*
  t_current (/ current.h */)*

- In ZPIC there are multiple parameters of type pointer to Array-of-Structs (AoS) that must be "packed":

  *t_part * t_species.part*
  *t_vfld * t_current.J_buf*

```
/*
 *  particles.h
 *  zpic
 *
 *  Created by Ricardo Fonseca on 11/8/10.
 *  Copyright 2010 Centro de Física dos Plasmas. All rights reserved.
 *
 */
typedef struct {

        char name[MAX_SPNAME_LEN];

        // Particle data buffer
        t_part *part;
        int np;
        int np_max;

        // mass over charge ratio
        t_part_data m_q;

        // total kinetic energy
        double energy;

        // charge of individual particle
        t_part_data q;

        // Number of particles per cell
        int ppc[2];

        // Density profile to inject
        t_density density;

        // Initial momentum of particles
        t_part_data ufl[3];
        t_part_data uth[3];

        // Simulation box info
        int nx[2];
        t_part_data dx[2];
        t_part_data box[2];

        // Time step
        float dt;

        // Iteration number
        int iter;

        // Moving window
        int moving_window;
        int n_move;

} t_species;
```

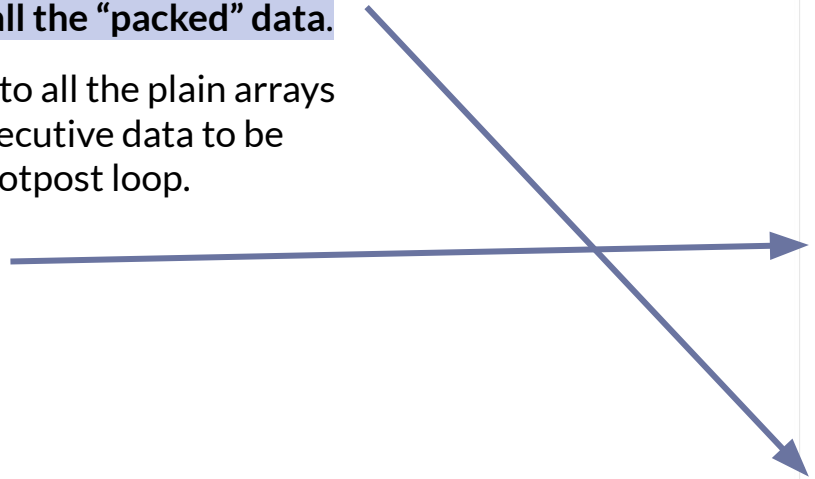
```
/*
 *  current.h
 *  zpic
 *
 *  Created by Ricardo Fonseca on 12/8/10.
 *  Copyright 2010 Centro de Física dos Plasmas. All rights reserved.
 *
 */

#ifndef __CURRENT__
#define __CURRENT__

#include "zpic.h"

enum smooth_type { NONE, BINOMIAL, COMPENSATED };

typedef struct {
        enum smooth_type xtype, ytype;
        int xlevel, ylevel;
} t_smooth;

typedef struct {

        t_vfld *J;

        t_vfld *J_buf;

        // Grid parameters
        int nx[2];
        int nrow;
        int gc[2][2];

        // Box size
        t_fld box[2];

        // Cell size
        t_fld dx[2];

        // Current smoothing
        t_smooth smooth;

        // Time step
        float dt;

        // Iteration number
        int iter;

        // Moving window
        int moving_window;

} t_current;
```

EM2D #1 (cont'd)

- Let's take a look at how we have implemented the gather-scatter in ZPIC.
- First, we have defined **a new *struct* *t_all_data* to hold all the "packed" data.**
- It provides access to all the plain arrays that hold the consecutive data to be processed in the hotpost loop.

  *t_fld *Jx_buf*
  *t_fld *Jy_buf*
  *t_fld *Jz_buf*
  *t_fld *Jx*
  *t_fld *Jy*
  *t_fld *Jz*
  ...

- With *t_fld* being a *float* data type.
  typedef float t_fld;



```c
#ifndef __ONEMALLOC__
#define __ONEMALLOC__

#include "current.h"
#include "emf.h"
#include <stdint.h>

typedef struct {
  int size;
  int offset;

  t_fld *Ex_buf;
  t_fld *Ey_buf;
  t_fld *Ez_buf;
  t_fld *Bx_buf;
  t_fld *By_buf;
  t_fld *Bz_buf;

  t_fld *Ex;
  t_fld *Ey;
  t_fld *Ez;
  t_fld *Bx;
  t_fld *By;
  t_fld *Bz;
} t_emf_data;

typedef struct {
  int size;
  int offset;

  t_fld *Jx_buf;
  t_fld *Jy_buf;
  t_fld *Jz_buf;

  t_fld *Jx;
  t_fld *Jy;
  t_fld *Jz;
} t_current_data;

typedef struct {
  t_current_data current_data;
  t_emf_data emf_data;
} t_all_data;

void init_all_data(t_all_data *data, t_current *current, t_emf *emf);
void delete_all_data(t_all_data *data);

void copy_in_all_data(t_all_data *data, t_current *current, t_emf *emf);
void copy_out_all_data(t_all_data *data, t_current *current, t_emf *emf);

#endif //__ONEMALLOC__
```

EM2D #1 (outlining, gather-scatter) (cont'd)

Allocate (malloc) memory

```
5   void init_all_data(t_all_data *data, t_current *current, t_emf *emf) {
6
7     // t_current buffers
8     data->current_data.size = current->size;
9     data->current_data.offset = current->offset;
10
11    data->current_data.Jx_buf = malloc(current->size * sizeof(t_fld));
12    data->current_data.Jy_buf = malloc(current->size * sizeof(t_fld));
13    data->current_data.Jz_buf = malloc(current->size * sizeof(t_fld));
14
15    assert(data->current_data.Jx_buf);
16    assert(data->current_data.Jy_buf);
17    assert(data->current_data.Jz_buf);
18
19    // Move pointers to [0][0]
20    data->current_data.Jx = data->current_data.Jx_buf + data->current_data.offset;
21    data->current_data.Jy = data->current_data.Jy_buf + data->current_data.offset;
22    data->current_data.Jz = data->current_data.Jz_buf + data->current_data.offset;
23
24    // t_emf buffers
25    data->emf_data.size = emf->size;
26    data->emf_data.offset = emf->offset;
27
28    data->emf_data.Ex_buf = malloc(emf->size * sizeof(t_fld));
29    data->emf_data.Ey_buf = malloc(emf->size * sizeof(t_fld));
30    data->emf_data.Ez_buf = malloc(emf->size * sizeof(t_fld));
31    data->emf_data.Bx_buf = malloc(emf->size * sizeof(t_fld));
32    data->emf_data.By_buf = malloc(emf->size * sizeof(t_fld));
33    data->emf_data.Bz_buf = malloc(emf->size * sizeof(t_fld));
34
35    assert(data->emf_data.Ex_buf);
36    assert(data->emf_data.Ey_buf);
37    assert(data->emf_data.Ez_buf);
38    assert(data->emf_data.Bx_buf);
39    assert(data->emf_data.By_buf);
40    assert(data->emf_data.Bz_buf);
41
42    // Move pointers to [0][0]
43    data->emf_data.Ex = data->emf_data.Ex_buf + data->emf_data.offset;
44    data->emf_data.Ey = data->emf_data.Ey_buf + data->emf_data.offset;
45    data->emf_data.Ez = data->emf_data.Ez_buf + data->emf_data.offset;
46
47    data->emf_data.Bx = data->emf_data.Bx_buf + data->emf_data.offset;
48    data->emf_data.By = data->emf_data.By_buf + data->emf_data.offset;
49    data->emf_data.Bz = data->emf_data.Bz_buf + data->emf_data.offset;
50  }
```

Copy data to/from 1D arrays

```
76  void copy_in_all_data(t_all_data *data, t_current *current, t_emf *emf) {
77
78    for (int i = 0; i < data->emf_data.size; ++i) {
79      data->emf_data.Ex_buf[i] = emf->E_buf[i].x;
80      data->emf_data.Ey_buf[i] = emf->E_buf[i].y;
81      data->emf_data.Ez_buf[i] = emf->E_buf[i].z;
82
83      data->emf_data.Bx_buf[i] = emf->B_buf[i].x;
84      data->emf_data.By_buf[i] = emf->B_buf[i].y;
85      data->emf_data.Bz_buf[i] = emf->B_buf[i].z;
86    }
87
88    for (int i = 0; i < data->current_data.size; ++i) {
89      data->current_data.Jx_buf[i] = current->J_buf[i].x;
90      data->current_data.Jy_buf[i] = current->J_buf[i].y;
91      data->current_data.Jz_buf[i] = current->J_buf[i].z;
92    }
93  }
94
95  void copy_out_all_data(t_all_data *data, t_current *current, t_emf *emf) {
96
97    for (int i = 0; i < data->emf_data.size; ++i) {
98      emf->E_buf[i].x = data->emf_data.Ex_buf[i];
99      emf->E_buf[i].y = data->emf_data.Ey_buf[i];
100     emf->E_buf[i].z = data->emf_data.Ez_buf[i];
101
102     emf->B_buf[i].x = data->emf_data.Bx_buf[i];
103     emf->B_buf[i].y = data->emf_data.By_buf[i];
104     emf->B_buf[i].z = data->emf_data.Bz_buf[i];
105   }
106
107   for (int i = 0; i < data->current_data.size; ++i) {
108     current->J_buf[i].x = data->current_data.Jx_buf[i];
109     current->J_buf[i].y = data->current_data.Jy_buf[i];
110     current->J_buf[i].z = data->current_data.Jz_buf[i];
111   }
112 }
```

Deallocate (free) memory

```
52  void delete_all_data(t_all_data *data) {
53    free(data->current_data.Jx_buf);
54    free(data->current_data.Jy_buf);
55    free(data->current_data.Jz_buf);
56
57    data->current_data.Jx_buf = NULL;
58    data->current_data.Jy_buf = NULL;
59    data->current_data.Jz_buf = NULL;
60
61    free(data->emf_data.Ex_buf);
62    free(data->emf_data.Ey_buf);
63    free(data->emf_data.Ez_buf);
64    free(data->emf_data.Bx_buf);
65    free(data->emf_data.By_buf);
66    free(data->emf_data.Bz_buf);
67
68    data->emf_data.Ex_buf = NULL;
69    data->emf_data.Ey_buf = NULL;
70    data->emf_data.Ez_buf = NULL;
71    data->emf_data.Bx_buf = NULL;
72    data->emf_data.By_buf = NULL;
73    data->emf_data.Bz_buf = NULL;
74  }
```

EM2D #1 (outlining, gather-scatter) (cont'd)

- The ZPIC code also adapts the size of the data structures to the number of particles interacting at each simulation step. This leads to additional computations to keep data consistent in ZPIC.

```
783  void pack_spec(const int size, const t_part *part, int **ix, int **iy,
784                 t_part_data **x, t_part_data **y, t_part_data **ux,
785                 t_part_data **uy, t_part_data **uz) {
786
787    *ix = malloc(size * sizeof(int));
788    *iy = malloc(size * sizeof(int));
789
790    *x = malloc(size * sizeof(t_part_data));
791    *y = malloc(size * sizeof(t_part_data));
792    *ux = malloc(size * sizeof(t_part_data));
793    *uy = malloc(size * sizeof(t_part_data));
794    *uz = malloc(size * sizeof(t_part_data));
795
796    assert(*ix);
797    assert(*iy);
798    assert(*x);
799    assert(*y);
800    assert(*ux);
801    assert(*uy);
802    assert(*uz);
803
804    for (int i = 0; i < size; ++i) {
805      (*ix)[i] = part[i].ix;
806      (*iy)[i] = part[i].iy;
807
808      (*x)[i] = part[i].x;
809      (*y)[i] = part[i].y;
810      (*ux)[i] = part[i].ux;
811      (*uy)[i] = part[i].uy;
812      (*uz)[i] = part[i].uz;
813    }
814  }
```

```
816  void unpack_spec(const int size, t_part *part, int **ix, int **iy,
817                   t_part_data **x, t_part_data **y, t_part_data **ux,
818                   t_part_data **uy, t_part_data **uz) {
819
820    for (int i = 0; i < size; ++i) {
821      part[i].ix = (*ix)[i];
822      part[i].iy = (*iy)[i];
823
824      part[i].x = (*x)[i];
825      part[i].y = (*y)[i];
826      part[i].ux = (*ux)[i];
827      part[i].uy = (*uy)[i];
828      part[i].uz = (*uz)[i];
829    }
830
831    free(*ix);
832    free(*iy);
833    free(*x);
834    free(*y);
835    free(*ux);
836    free(*uy);
837    free(*uz);
838
839    *ix = *iy = NULL;
840    *x = *y = *ux = *uy = *uz = NULL;
841  }
```

EM2D #1 (cont'd)

- How do we glue all these pieces of code?
- The original function *spec_advance()* has the same interface with the rest of the code.
- Reallocation of memory is needed before running the hotspot loop.
- Also needed to gather the relevant data into consecutive arrays.
- Next step is to compute the hotspot loop by invoking the outlined function.
- Once the output of the simulation step is ready, it is scattered to the corresponding memory locations.
- And finally the memory re-allocation is done after running the hotspot loop.

```
844 void spec_advance(t_species *spec, t_emf *emf, t_current *current,
845                    t_all_data *data) {
846   int i;
847   t_part_data qnx, qny, qvz;
848
849   uint64_t t0;
850   t0 = timer_ticks();
851
852   const t_part_data tem = 0.5 * spec->dt / spec->m_q;
853   const t_part_data dt_dx = spec->dt / spec->dx[0];
854   const t_part_data dt_dy = spec->dt / spec->dx[1];
855
856   // Auxiliary values for current deposition
857   qnx = spec->q * spec->dx[0] / spec->dt;
858   qny = spec->q * spec->dx[1] / spec->dt;
859
860   const int nx0 = spec->nx[0];
861   const int nx1 = spec->nx[1];
862
863   ////////////////////
864   // Pack spec->part //
865   ////////////////////
866   int *spec_ix, *spec_iy;
867   t_part_data *spec_x, *spec_y, *spec_ux, *spec_uy, *spec_uz;
868
869   pack_spec(spec->np_max, spec->part, &spec_ix, &spec_iy, &spec_x, &spec_y,
870             &spec_ux, &spec_uy, &spec_uz);
871
872   copy_in_all_data(data, current, emf);
873
874   spec_advance_outlining( // spec data
875       spec->part, spec->np, spec->q, qnx, qny, tem, dt_dx, dt_dy, spec_ix,
876       spec_iy, spec_x, spec_y, spec_ux, spec_uy, spec_uz,
877       // emf data
878       emf->nrow, data->emf_data.Ex, data->emf_data.Ey, data->emf_data.Ez,
879       data->emf_data.Bx, data->emf_data.By, data->emf_data.Bz,
880       // current data
881       current, data->current_data.Jx, data->current_data.Jy,
882       data->current_data.Jz);
883
884   copy_out_all_data(data, current, emf);
885
886   ////////////////////
887   //   Unpack spec   //
888   ////////////////////
889   unpack_spec(spec->np_max, spec->part, &spec_ix, &spec_iy, &spec_x, &spec_y,
890               &spec_ux, &spec_uy, &spec_uz);
891
892   // Advance internal iteration number
893   spec->iter += 1;
894   _spec_npush += spec->np;
```

EM2D #1 (outlining, gather-scatter) (cont'd)

- Let's see changes in Codee Screening Report by narrowing it down to the outlined function.
- Reported 0 checks, so nothing helps to exploit parallelism (Vector, Multi and Offload).

```
$ cd em2d-1-outlining/

$ pwreport --screening --include-tags all particles.c:spec_advance_outlining --brief -- -Ofast -lm
Compiler flags: -Ofast -lm

[C] target compiler: <none> (Compiler Agnostic Mode)

SCREENING REPORT

Target                            Lines of code Optimizable lines Analysis time # checks Effort Cost Profiling
--------------------------------- ------------- ----------------- ------------- -------- ------ ---- ---------
particles.c:spec_advance_outlining 837           0                 128 ms        0        0 h    0€   n/a
--------------------------------- ------------- ----------------- ------------- -------- ------ ---- ---------
Total                             837           0                 128 ms        0        0 h    0€   n/a

CHECKS PER STAGE OF THE PERFORMANCE OPTIMIZATION ROADMAP
Target                            Scalar Control Memory Vector Multi Offload
--------------------------------- ------ ------- ------ ------ ----- -------
particles.c:spec_advance_outlining 0      0       0      0      0     0
--------------------------------- ------ ------- ------ ------ ----- -------
Total                             0      0       0      0      0     0

TOTAL NUMBER OF LOOPS FOR CODEE AUTO AND GUIDED MODES
                  ---------------- # checks ----------------
Codee mode # Loops Scalar Control Memory Vector Multi Offload
---------- ------- ------ ------- ------ ------ ----- -------
Auto       0       0      0       0      0      0     0
Guided     0       0      0       0      0      0     0

1 file successfully analyzed and 0 failures in 129 ms
```

EM2D #1 (outlining, gather-scatter data) (cont'd)

- Let's investigate the remaining challenges in the outlined function through *pwreport --non-analyzable*.
- Codee output message "*Unsupported interprocedural dependency analysis for a function*" pinpoints the call to the function "*dep_current_zamp()*" as the next issue to overcome.

```
$ pwreport --non-analyzable particles.c:spec_advance_outlining
[C] target compiler: <none> (Compiler Agnostic Mode)

Count Type
----- ----------------------------------------------------------------
1     Unsupported interprocedural dependency analysis for a function

Count : Number of occurrences of the unsupported feature
Type : Type of unsupported feature

1 file successfully analyzed and 0 failures in 96 ms
```

Parallel ZPIC/em2d: from serial to CPU & GPU...

EM2D #7a (Basic GPU+OpenACC Atomic)

EM2D #6a (Basic GPU+OpenMP Atomic)

EM2D #5a (Basic CPU+OpenMP Atomic)

EM2D #4 (loop fission)

EM2D #3 (change Array of Structs by separate arrays)

EM2D #2 (inlining deb_current_zamb)

EM2D #1 (outlining, gather-scatter data)

EM2D #0 (serial version)

EM2D #2 (inlining deb_current_zamb)

- In computing, **inline expansion**, or inlining, is a manual or compiler optimization that **replaces a function call site with the body of the called function**.
- Benefits for the programmer:
  - Eliminates overhead from calls to functions/procedures.
  - Enable code transformations that otherwise would not be applicable.
- In ZPIC we use inlining to enable the detection of the pattern “sparse reduction” by Codee (it is work-in-progress to do detection across procedure calls).

The code has a function that is called in other code sections:

```
void foo(int *x, int i) {
   x[i] = 0;
}
```

Before inlining (e.g., assume *foo* is called within a loop):

```
for(int i=0; i<n; ++i) {
   foo(x, i);
}
```

After inlining:

```
for(int i=0; i<n; ++i) {
   x[i] = 0;
}
```

EM2D #2 (inlining deb_current_zamb)

Before inlining

```
    // Store new momenta
    spec_ux[i] = ux;
    spec_uy[i] = uy;
    spec_uz[i] = uz;

    // push particle
    rg = 1.0f / sqrtf(1.0f + ux * ux + uy * uy + uz * uz);

    dx = dt_dx * rg * ux;
    dy = dt_dy * rg * uy;

    x1 = spec_x[i] + dx;
    y1 = spec_y[i] + dy;

    di = ltrim(x1);
    dj = ltrim(y1);

    x1 -= di;
    y1 -= dj;

    qvz = spec_q * uz * rg;

    dep_current_zamb(spec_ix[i], spec_iy[i], di, dj, spec_x[i], spec_y[i], dx,
                     dy, qnx, qny, qvz, current_nrow, current_Jx, current_Jy,
                     current_Jz);

    // Store results
    spec_x[i] = x1;
    spec_y[i] = y1;
    spec_ix[i] += di;
    spec_iy[i] += dj;
  }
}
// clang-format off

void pack_spec(const int size, const t_part *part, int **ix, int **iy,
               t_part_data **x, t_part_data **y, t_part_data **ux,
               t_part_data **uy, t_part_data **uz) {

  *ix = malloc(size * sizeof(int));
  *iy = malloc(size * sizeof(int));

  *x = malloc(size * sizeof(t_part_data));
  *y = malloc(size * sizeof(t_part_data));
  *ux = malloc(size * sizeof(t_part_data));
  *uy = malloc(size * sizeof(t_part_data));
```

After inlining

```
    // dep_current_zamb(spec_ix[i], spec_iy[i], di, dj, spec_x[i], spec_y[i],
    //                  dx, dy, qnx, qny, qvz, current_nrow, current_Jx,
    //                  current_Jy, current_Jz);

    ////////////////////////////////////
    //   dep_current_zamb inlined START //
    ////////////////////////////////////

    const float ix = spec_ix[i];
    const float iy = spec_iy[i];
    const float x0 = spec_x[i];

    ...

    wp1[0] = 0.5f * (S0y[0] + S1y[0]);
    wp1[1] = 0.5f * (S0y[1] + S1y[1]);

    wp2[0] = 0.5f * (S0x[0] + S1x[0]);
    wp2[1] = 0.5f * (S0x[1] + S1x[1]);

    current_Jx[vp[k].ix + nrow * vp[k].iy] += wl1 * wp1[0];
    current_Jx[vp[k].ix + nrow * (vp[k].iy + 1)] += wl1 * wp1[1];

    current_Jy[vp[k].ix + nrow * vp[k].iy] += wl2 * wp2[0];
    current_Jy[vp[k].ix + 1 + nrow * vp[k].iy] += wl2 * wp2[1];

    current_Jz[vp[k].ix + nrow * vp[k].iy] +=
        vp[k].qvz * (S0x[0] * S0y[0] + S1x[0] * S1y[0] +
                     (S0x[0] * S1y[0] - S1x[0] * S0y[0]) / 2.0f);
    current_Jz[vp[k].ix + 1 + nrow * vp[k].iy] +=
        vp[k].qvz * (S0x[1] * S0y[0] + S1x[1] * S1y[0] +
                     (S0x[1] * S1y[0] - S1x[1] * S0y[0]) / 2.0f);
    current_Jz[vp[k].ix + nrow * (vp[k].iy + 1)] +=
        vp[k].qvz * (S0x[0] * S0y[1] + S1x[0] * S1y[1] +
                     (S0x[0] * S1y[1] - S1x[0] * S0y[1]) / 2.0f);
    current_Jz[vp[k].ix + 1 + nrow * (vp[k].iy + 1)] +=
        vp[k].qvz * (S0x[1] * S0y[1] + S1x[1] * S1y[1] +
                     (S0x[1] * S1y[1] - S1x[1] * S0y[1]) / 2.0f);
  }

  ////////////////////////////////////
  //   dep_current_zamb inlined  END //
  ////////////////////////////////////
```

The inlining exposes the computation of *Jx*, *Jy* and *Jz* in the target hotspot loop. As a result, the sparse reduction pattern in the code.

EM2D #2 (inlining deb_current_zamb) (cont'd)

- After applying inlining, Codee reports a total of 135 checks in the hotspot function.
- It also reports 2 loops in the function, but 0 opportunities related to parallelism (Multi, Offload).

```
$ cd em2d-2-inlining/

$ pwreport --screening --include-tags all particles.c:spec_advance_outlining --brief -- -Ofast -lm
Compiler flags: -Ofast -lm

[C] target compiler: <none> (Compiler Agnostic Mode)

SCREENING REPORT

Target                              Lines of code Optimizable lines Analysis time # checks Effort Cost     Profiling
----------------------------------- ------------- ----------------- ------------- -------- ------ -------- ---------
particles.c:spec_advance_outlining 936           172               183 ms        135      513 h  16787€   n/a
----------------------------------- ------------- ----------------- ------------- -------- ------ -------- ---------
Total                               936           172               183 ms        135      513 h  16787€   n/a

CHECKS PER STAGE OF THE PERFORMANCE OPTIMIZATION ROADMAP
Target                              Scalar Control Memory Vector Multi Offload
----------------------------------- ------ ------- ------ ------ ----- -------
particles.c:spec_advance_outlining 11     1       121    2      0     0
----------------------------------- ------ ------- ------ ------ ----- -------
Total                               11     1       121    2      0     0

TOTAL NUMBER OF LOOPS FOR CODEE AUTO AND GUIDED MODES
                    ---------- # checks ----------
Codee mode # Loops Scalar Control Memory Vector Multi Offload
---------- ------- ------ ------- ------ ------ ----- -------
Auto       0       0      0       0      0      0     0
Guided     2       5      1       121    2      0     0

1 file successfully analyzed and 0 failures in 183 ms
```

EM2D #2 (inlining deb_current_zamb) (cont'd)

- Let's see **the detailed list of checkers reported by Codee**, and we can identify the next recommendations:
  - Avoid the usage of Array of Structs (AoS) [PWR016]
  - Avoid the usage of non-consecutive and indirect memory accesses [PWR010, PWR035, PWR036]
- These recommendations are reported for the loops at lines 690 and 891.

```
$ pwreport --checks --include-tags all particles.c:spec_advance_outlining --brief -- -Ofast -lm
Compiler flags: -Ofast -lm

[C] target compiler: <none> (Compiler Agnostic Mode)

CHECKS REPORT

particles.c:823:22 [PWR048]: Replace multiplication/addition combo with an explicit call to fused multiply-add
...
particles.c:795:5 [PWR016]: using separated plain arrays instead of Array-of-Structs (AoS) 'vp' is recommended to maximize data locality
...
particles.c:687:15 [PWR002]: 'qvz' not declared in the innermost scope possible
particles.c:690:3 [PWR010]: 'vp' multi-dimensional array not accessed in row-major order
particles.c:690:3 [PWR021]: extract temporary computations to a vectorizable loop
particles.c:690:3 [PWR035]: avoid non-consecutive array access for variable 'vp' to improve performance
particles.c:690:3 [PWR036]: avoid indirect array access for variables 'current_Jz', 'current_Jy' and 'current_Jx' to improve performance
particles.c:804:17 [PWR045]: replace division with a multiplication with a reciprocal
particles.c:896:23 [PWR016]: using separated plain arrays instead of Array-of-Structs (AoS) 'vp' is recommended to maximize data locality
...
particles.c:891:5 [PWR010]: 'vp' multi-dimensional array not accessed in row-major order
particles.c:891:5 [PWR021]: extract temporary computations to a vectorizable loop
particles.c:891:5 [PWR035]: avoid non-consecutive array access for variable 'vp' to improve performance
particles.c:891:5 [PWR036]: avoid indirect array access for variables 'current_Jz', 'current_Jy' and 'current_Jx' to improve performance
particles.c:925:24 [PWR045]: replace division with a multiplication with a reciprocal
...

1 file successfully analyzed and 0 failures in 209 ms
```

Parallel ZPIC/em2d: from serial to CPU & GPU...

EM2D #7a (Basic GPU+OpenACC Atomic)

EM2D #6a (Basic GPU+OpenMP Atomic)

EM2D #5a (Basic CPU+OpenMP Atomic)

EM2D #4 (loop fission)

EM2D #3 (change Array of Structs by separate arrays)

✓ EM2D #2 (inlining deb_current_zamb)

✓ EM2D #1 (outlining, gather-scatter data)

✓ EM2D #0 (serial version)

EM2D #3 (change Array of Structs by separate arrays)

- In computing, **Array of Structures (AoS)**, Structure of Arrays (SoA) and Array of Structures of Arrays (AoSoA) refer to contrasting ways to arrange a sequence of records in memory, with regard to interleaving, and are of interest in SIMD programming, for example.
- In ZPIC we use it to enable the detection of the pattern “sparse reduction”, which cannot be detected because the loop body contains uses of AoS (that we must first replace by separate arrays).

Code example

The following example shows a loop processing the *x* and *y* coordinates for an array of points:

```
typedef struct {
    int x;
    int y;
    int z;
} point;

void foo() {
    point points[1000];
    for (int i = 0; i < 1000; i ++) {
        points[i].x = 1;
        points[i].y = 1;
    }
}
```

This could seem like an example where using an Array-of-Structs is justifiable. However, since the *z* coordinate is never accessed along the other two, there may be cache misses that could be avoided by creating one array for each coordinate:

```
void foo() {
    int points_x[1000];
    int points_y[1000];
    int points_z[1000];
    for (int i = 0; i < 1000; i ++) {
        points_x[i] = 1;
        points_y[i] = 1;
    }
}
```

[Source: PWR016: Use separate arrays instead of an Array-of-Structs (https://www.codee.com/knowledge/pwr016/)]

EM2D #3 (change Array of Structs by separate arrays)

Before AoS into separate arrays

```
for (k = 0; k < vnp; k++) {
  float S0x[2], S1x[2], S0y[2], S1y[2];
  float wl1, wl2;
  float wp1[2], wp2[2];

  S0x[0] = 1.0f - vp[k].x0;
  S0x[1] = vp[k].x0;

  S1x[0] = 1.0f - vp[k].x1;
  S1x[1] = vp[k].x1;

  S0y[0] = 1.0f - vp[k].y0;
  S0y[1] = vp[k].y0;

  S1y[0] = 1.0f - vp[k].y1;
  S1y[1] = vp[k].y1;

  wl1 = qnx * vp[k].dx;
  wl2 = qny * vp[k].dy;

  wp1[0] = 0.5f * (S0y[0] + S1y[0]);
  wp1[1] = 0.5f * (S0y[1] + S1y[1]);

  wp2[0] = 0.5f * (S0x[0] + S1x[0]);
  wp2[1] = 0.5f * (S0x[1] + S1x[1]);

  current_Jx[vp[k].ix + nrow * vp[k].iy] += wl1 * wp1[0];
  current_Jx[vp[k].ix + nrow * (vp[k].iy + 1)] += wl1 * wp1[1];

  current_Jy[vp[k].ix + nrow * vp[k].iy] += wl2 * wp2[0];
  current_Jy[vp[k].ix + 1 + nrow * vp[k].iy] += wl2 * wp2[1];

  current_Jz[vp[k].ix + nrow * vp[k].iy] +=
      vp[k].qvz * (S0x[0] * S0y[0] + S1x[0] * S1y[0] +
                   (S0x[0] * S1y[0] - S1x[0] * S0y[0]) / 2.0f);
  current_Jz[vp[k].ix + 1 + nrow * vp[k].iy] +=
      vp[k].qvz * (S0x[1] * S0y[0] + S1x[1] * S1y[0] +
                   (S0x[1] * S1y[0] - S1x[1] * S0y[0]) / 2.0f);
  current_Jz[vp[k].ix + nrow * (vp[k].iy + 1)] +=
      vp[k].qvz * (S0x[0] * S0y[1] + S1x[0] * S1y[1] +
                   (S0x[0] * S1y[1] - S1x[0] * S0y[1]) / 2.0f);
  current_Jz[vp[k].ix + 1 + nrow * (vp[k].iy + 1)] +=
      vp[k].qvz * (S0x[1] * S0y[1] + S1x[1] * S1y[1] +
                   (S0x[1] * S1y[1] - S1x[1] * S0y[1]) / 2.0f);
}
```

After AoS into separate arrays

```
float S0x[2], S1x[2], S0y[2], S1y[2];
float wl1, wl2;
float wp1[2], wp2[2];

S0x[0] = 1.0f - vp_x0[k];
S0x[1] = vp_x0[k];

S1x[0] = 1.0f - vp_x1[k];
S1x[1] = vp_x1[k];

S0y[0] = 1.0f - vp_y0[k];
S0y[1] = vp_y0[k];

S1y[0] = 1.0f - vp_y1[k];
S1y[1] = vp_y1[k];

wl1 = qnx * vp_dx[k];
wl2 = qny * vp_dy[k];

wp1[0] = 0.5f * (S0y[0] + S1y[0]);
wp1[1] = 0.5f * (S0y[1] + S1y[1]);

wp2[0] = 0.5f * (S0x[0] + S1x[0]);
wp2[1] = 0.5f * (S0x[1] + S1x[1]);

current_Jx[vp_ix[k] + nrow * vp_iy[k]] += wl1 * wp1[0];
current_Jx[vp_ix[k] + nrow * (vp_iy[k] + 1)] += wl1 * wp1[1];

current_Jy[vp_ix[k] + nrow * vp_iy[k]] += wl2 * wp2[0];
current_Jy[vp_ix[k] + 1 + nrow * vp_iy[k]] += wl2 * wp2[1];

current_Jz[vp_ix[k] + nrow * vp_iy[k]] +=
    vp_qvz[k] * (S0x[0] * S0y[0] + S1x[0] * S1y[0] +
                 (S0x[0] * S1y[0] - S1x[0] * S0y[0]) / 2.0f);
current_Jz[vp_ix[k] + 1 + nrow * vp_iy[k]] +=
    vp_qvz[k] * (S0x[1] * S0y[0] + S1x[1] * S1y[0] +
                 (S0x[1] * S1y[0] - S1x[1] * S0y[0]) / 2.0f);
current_Jz[vp_ix[k] + nrow * (vp_iy[k] + 1)] +=
    vp_qvz[k] * (S0x[0] * S0y[1] + S1x[0] * S1y[1] +
                 (S0x[0] * S1y[1] - S1x[0] * S0y[1]) / 2.0f);
current_Jz[vp_ix[k] + 1 + nrow * (vp_iy[k] + 1)] +=
    vp_qvz[k] * (S0x[1] * S0y[1] + S1x[1] * S1y[1] +
                 (S0x[1] * S1y[1] - S1x[1] * S0y[1]) / 2.0f);
}

////////////////////////////////////
//  dep current_zamb_inlined  END //
```

Applying convert AoS into separate arrays enables to detect the sparse reduction pattern associated with *Jx*, *Jy* and *Jz* in the target hotspot loop.

Example: Replace ‘vp[k].x0’ by ‘vp_x0[k]’.

EM2D #3 (change Array of Structs by separate arrays) (cont'd)

- Now that the AOS issues have been overcome, Codee still reports the non-consecutive memory access issues [PWR036] in both loops at lines 690 and 887.

```
$ cd em2d-3-aos/

$ pwreport --checks particles.c:spec_advance_outlining --brief -- -Ofast -lm
[C] target compiler: <none> (Compiler Agnostic Mode)

CHECKS REPORT

particles.c:819:22 [PWR048]: Replace multiplication/addition combo with an explicit call to fused multiply-add
particles.c:856:22 [PWR048]: Replace multiplication/addition combo with an explicit call to fused multiply-add
particles.c:920:24 [PWR048]: Replace multiplication/addition combo with an explicit call to fused multiply-add
particles.c:923:24 [PWR048]: Replace multiplication/addition combo with an explicit call to fused multiply-add
particles.c:926:24 [PWR048]: Replace multiplication/addition combo with an explicit call to fused multiply-add
particles.c:929:24 [PWR048]: Replace multiplication/addition combo with an explicit call to fused multiply-add
particles.c:690:3 [PWR021]: extract temporary computations to a vectorizable loop
particles.c:690:3 [PWR036]: avoid indirect array access for variables 'current_Jz', 'current_Jy' and 'current_Jx' to improve performance
particles.c:800:17 [PWR045]: replace division with a multiplication with a reciprocal
particles.c:887:5 [PWR021]: extract temporary computations to a vectorizable loop
particles.c:887:5 [PWR036]: avoid indirect array access for variables 'current_Jz', 'current_Jy' and 'current_Jx' to improve performance
particles.c:921:24 [PWR045]: replace division with a multiplication with a reciprocal
particles.c:924:24 [PWR045]: replace division with a multiplication with a reciprocal
particles.c:927:24 [PWR045]: replace division with a multiplication with a reciprocal
particles.c:930:24 [PWR045]: replace division with a multiplication with a reciprocal

1 file successfully analyzed and 0 failures in 184 ms
```

Parallel ZPIC/em2d: from serial to CPU & GPU...

EM2D #7a (Basic GPU+OpenACC Atomic)

EM2D #6a (Basic GPU+OpenMP Atomic)

EM2D #5a (Basic CPU+OpenMP Atomic)

EM2D #4 (loop fission)

✓ EM2D #3 (change Array of Structs by separate arrays)

✓ EM2D #2 (inlining deb_current_zamb)

✓ EM2D #1 (outlining, gather-scatter data)

✓ EM2D #0 (serial version)

EM2D #4 (loop fission)

- In computer science, **loop fission** (or loop distribution) is a compiler optimization in which **a loop is broken into multiple loops over the same index range with each taking only a part of the original loop's body**.
- Benefits:
  - Break down a large loop body into smaller ones to achieve better utilization of locality of reference.
  - In multi-core processors that can split a task into multiple tasks for each processor.
- In ZPIC we use it to enable the detection of the pattern “sparse reduction”, by separating the part of the loop body that cannot be matched to a pattern by Codee tools.

Before loop fission the code has a loop that computes several output variables:

```
for(int i=0; i<n; ++i) {
   x[i] = 0;
   y[i] = y[i-1] + 1;
}
```

After loop fission: the code contains several loops dedicated to the computation of different outputs:

```
for(int i=0; i<n; ++i) {
   x[i] = 0;
}
for(int i=0; i<n; ++i) {
   y[i] = y[i-1] + 1;
}
```

EM2D #4 (loop fission) (cont'd)

- Let's use *pwloops* to figure out which variables lead to *n/a*, to split the loop body accordingly.
- So we need to split the loop body so that the computation of following variables is in a separate loop:

  spec_ux spec_uy spec_uz spec_x spec_y spec_ix spec_iy

```
$ cd em2d-3-aos/

$ pwloops --datascoping particles.c:690 --single-loop --brief -- -Ofast -lm
Compiler flags: -Ofast -lm

[C] target compiler: <none> (Compiler Agnostic Mode)

Loop                              Variable     Kind    Read/Write Temporary Compute Pattern Memory Pattern OpenMP (multi)           OpenACC (offload)
--------------------------------- ------------ ------- ---------- --------- ---------------- -------------- ----------------------- ----------------------
particles.c
`- spec_advance_outlining:690:3
   |->                            Bpx          scalar  rw         x
   |->                            Bpy          scalar  rw         x
   |->                            Bpz          scalar  rw         x
   |->                            Bx           pointer ro                                                   shared(Bx)              copyin(Bx)
   |->                            By           pointer ro                                                   shared(By)              copyin(By)
   |->                            Bz           pointer ro                                                   shared(Bz)              copyin(Bz)
...
   |->                            qnx          scalar  ro                                                   shared(qnx)             copyin(qnx)
   |->                            qny          scalar  ro                                                   shared(qny)             copyin(qny)
   |->                            qvz          scalar  rw         x                                         private(qvz)
   |->                            rg           scalar  rw         x
   |->                            spec_ix[]    pointer n/a        n/a       n/a              linear         n/a                     n/a
   |->                            spec_iy[]    pointer n/a        n/a       n/a              linear         n/a                     n/a
   |->                            spec_np      scalar  ro                                                   shared(spec_np)         copyin(spec_np)
   |->                            spec_q       scalar  ro                                                   shared(spec_q)          copyin(spec_q)
   |->                            spec_ux[]    pointer n/a        n/a       n/a              linear         n/a                     n/a
   |->                            spec_uy[]    pointer n/a        n/a       n/a              linear         n/a                     n/a
   |->                            spec_uz[]    pointer n/a        n/a       n/a              linear         n/a                     n/a
   |->                            spec_x[]     pointer n/a        n/a       n/a              linear         n/a                     n/a
   |->                            spec_y[]     pointer n/a        n/a       n/a              linear         n/a                     n/a
...

1 file successfully analyzed and 0 failures in 1309 ms
```

EM2D #4 (loop fission) (cont'd)

Before loop fission

```
ux = utx + uty * Bpz - utz * Bpy;
uy = uty + utz * Bpx - utx * Bpz;
uz = utz + utx * Bpy - uty * Bpx;

// Perform second half of the rotation

Bpx *= otsq;
Bpy *= otsq;
Bpz *= otsq;

utx += uy * Bpz - uz * Bpy;
uty += uz * Bpx - ux * Bpz;
utz += ux * Bpy - uy * Bpx;

// Perform second half of electric field acceleration
ux = utx + Epx;
uy = uty + Epy;
uz = utz + Epz;

// Store new momenta
spec_ux[i] = ux;
spec_uy[i] = uy;
spec_uz[i] = uz;

// push particle
rg = 1.0f / sqrtf(1.0f + ux * ux + uy * uy + uz * uz);

dx = dt_dx * rg * ux;
dy = dt_dy * rg * uy;

x1 = spec_x[i] + dx;
y1 = spec_y[i] + dy;

di = ltrim(x1);
dj = ltrim(y1);

x1 -= di;
y1 -= dj;

qvz = spec_q * uz * rg;

// dep_current_zamb(spec_ix[i], spec_iy[i], di, dj, spec_x[i], spec_y[i],
//                  dx, dy, qnx, qny, qvz, current_nrow, current_Jx,
//                  current_Jy, current_Jz);
```

After loop fission

```
ux = utx + uty * Bpz - utz * Bpy;
uy = uty + utz * Bpx - utx * Bpz;
uz = utz + utx * Bpy - uty * Bpx;

// Perform second half of the rotation

Bpx *= otsq;
Bpy *= otsq;
Bpz *= otsq;

utx += uy * Bpz - uz * Bpy;
uty += uz * Bpx - ux * Bpz;
utz += ux * Bpy - uy * Bpx;

// Perform second half of electric field acceleration
ux = utx + Epx;
uy = uty + Epy;
uz = utz + Epz;

// Store new momenta
spec_ux[i] = ux;
spec_uy[i] = uy;
spec_uz[i] = uz;
}

for (int i = 0; i < spec_np; i++) {
  // Load particle momenta
  t_part_data ux = spec_ux[i];
  t_part_data uy = spec_uy[i];
  t_part_data uz = spec_uz[i];

  // push particle
  t_part_data rg = 1.0f / sqrtf(1.0f + ux * ux + uy * uy + uz * uz);

  dx = dt_dx * rg * ux;
  dy = dt_dy * rg * uy;

  x1 = spec_x[i] + dx;
  y1 = spec_y[i] + dy;

  di = ltrim(x1);
  dj = ltrim(y1);

  x1 -= di;
  y1 -= dj;
```

Applying loop fission requires to duplicate the loop header.

In this code it also requires to duplicate some computation in the two loops resulting from the fission.

EM2D #4 (loop fission) (cont'd)

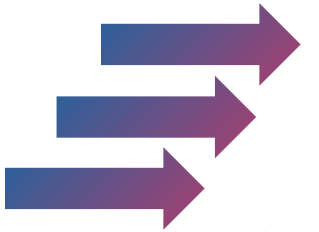
- **Now we have three loops, two outermost loops (lines 694 and 755) due to loop fission.**
- And the **second loop is analyzable (*spec_advance_outlining:755*), showing a parallelization opportunity with a sparse reduction pattern**.

```
$ cd em2d-4-loopfission/

$ pwreport --checks --include-tags all particles.c:spec_advance_outlining --brief -- -Ofast -lm

...
```

Parallel ZPIC/em2d: from serial to CPU & GPU...

- EM2D #7a (Basic GPU+OpenACC Atomic)
- EM2D #6a (Basic GPU+OpenMP Atomic)
- EM2D #5a (Basic CPU+OpenMP Atomic)
- ✓ EM2D #4 (loop fission)
- ✓ EM2D #3 (change Array of Structs by separate arrays)
- ✓ EM2D #2 (inlining deb_current_zamb)
- ✓ EM2D #1 (outlining, gather-scatter data)
- ✓ EM2D #0 (serial version)

Parallel ZPIC/em2d: from serial to CPU & GPU...



EM2D #7a (Basic GPU+OpenACC Atomic)

EM2D #6a (Basic GPU+OpenMP Atomic)

EM2D #5a (Basic CPU+OpenMP Atomic)

✓ EM2D #4 (loop fission)

✓ EM2D #3 (change Array of Structs by separate arrays)

✓ EM2D #2 (inlining deb_current_zamb)

✓ EM2D #1 (outlining, gather-scatter data)

✓ EM2D #0 (serial version)

EM2D #5a
(Basic CPU+OpenMP Atomic)

- Now that the sparse reduction loop is a parallelization opportunity reported by Codee, we can take advantage of the corresponding parallelization strategies using OpenMP on CPU:
  - Parallel loop w/ atomic
  - Parallel loop w/ explicit privatization
- **Let's generate the "atomic" version as it is the easiest to code and maintain**.
- Running this version built with *clang-11* on a Cori GPU node with 8 threads @NERSC takes 9.19 seconds (speedup 1.22x)

```
755  #pragma omp parallel default(none)                                    \
756      shared(current_nrow, current_Jx, current_Jy, current_Jz, dt_dx, dt_dy,  \
757             qnx, qny, spec_ix, spec_iy, spec_np, spec_q, spec_ux, spec_uy,   \
758             spec_uz, spec_x, spec_y) private(di, dj, dx, dy, qvz, x1, y1)
759    {
760    #pragma omp for private(di, dj, dx, dy, qvz, x1, y1) schedule(auto)
761    for (int i = 0; i < spec_np; i++) {
762      // Load particle momenta
763      t_part_data ux = spec_ux[i];
764      t_part_data uy = spec_uy[i];
765      t_part_data uz = spec_uz[i];
766
767      // push particle
768      t_part_data rg = 1.0f / sqrtf(1.0f + ux * ux + uy * uy + uz * uz);
       ...
947       #pragma omp atomic update
948       current_Jz[vp_ix[k] + 1 + nrow * (vp_iy[k] + 1)] +=
949           vp_qvz[k] * (S0x[1] * S0y[1] + S1x[1] * S1y[1] +
950                        (S0x[1] * S1y[1] - S1x[1] * S0y[1]) / 2.0f);
951     }
952
953     ////////////////////////////////////
954     //   dep_current_zamb inlined  END //
955     ////////////////////////////////////
956
957     // Store results
958     spec_x[i] = x1;
959     spec_y[i] = y1;
960     spec_ix[i] += di;
961     spec_iy[i] += dj;
962   }
963   } // end parallel
```

EM2D #6a
(Basic GPU+OpenMP Atomic)

- The advantage of the “atomic” strategy is that it is also available for OpenMP on GPU.
- **Let’s generate the “atomic” version as it is the easiest to code and maintain**.
  - Note that to implement the data transfers you will need an additional parameter *current_size* for *spec_advance_outlining()*, as needed to transfer arrays *current_Jx*, *current_Jy* and *current_Jz*.
- Running this version built with *clang-11* on a Cori GPU @NERSC takes 9,87 seconds (speedup 1,13x)
- Running this version built with *cray* on a Cori GPU @NERSC takes 8,27 seconds (speedup 1,35x)

```
755   int Jsize = current_size;
756 #pragma omp target teams distribute parallel for                          \
757               shared(dt_dx, dt_dy, qnx, qny, spec_np, spec_q, spec_ux,     \
758                         spec_uy, spec_uz)                                  \
759               map(to: dt_dx, dt_dy, qnx, qny, spec_np, spec_q,         \
760               spec_ux[0:spec_np], spec_uy[0:spec_np], spec_uz[0:spec_np])   \
761               private(di, dj, dx, dy, qvz, x1, y1)                         \
762       map(tofrom: current_Jx[0:Jsize], current_Jy[0:Jsize],                \
763       current_Jz[0:Jsize], spec_ix[0:spec_np],                  \
764                                   spec_iy[0:spec_np], spec_x[0:spec_np], spec_y[0:spec_np]) \
765               schedule(auto)
766   for (int i = 0; i < spec_np; i++) {
767     // Load particle momenta
768     t_part_data ux = spec_ux[i];
769     t_part_data uy = spec_uy[i];
770     t_part_data uz = spec_uz[i];
771
772     // push particle
773     t_part_data rg = 1.0f / sqrtf(1.0f + ux * ux + uy * uy + uz * uz);
```

...

```
952   #pragma omp atomic update
953         current_Jz[vp_ix[k] + 1 + nrow * (vp_iy[k] + 1)] +=
954             vp_qvz[k] * (S0x[1] * S0y[1] + S1x[1] * S1y[1] +
955                          (S0x[1] * S1y[1] - S1x[1] * S0y[1]) / 2.0f);
956       }
957
958       ////////////////////////////////////
959       //   dep_current_zamb inlined  END //
960       ////////////////////////////////////
961
962       // Store results
963       spec_x[i] = x1;
964       spec_y[i] = y1;
965       spec_ix[i] += di;
966       spec_iy[i] += dj;
967     }
```

EM2D #7a (Basic GPU+OpenACC Atomic)

- And the “atomic” strategy is also available for OpenACC on GPU.
- **Let’s generate the “atomic” version as it is the easiest to code and maintain**.
  - Note that to implement the data transfers you will need an additional parameter *current_size* for *spec_advance_outlining()*, as needed to transfer arrays *current_Jx*, *current_Jy* and *current_Jz*.
- Running this version built with *nvc* on a Cori GPU @NERSC takes 7,49 seconds (speedup 1,49x)

```
755   int Jsize = current_size;
756 #pragma acc data copyin(dt_dx, dt_dy, qnx, qny, spec_np, spec_q,            \
757                         spec_ux [0:spec_np], spec_uy [0:spec_np],           \
758                         spec_uz [0:spec_np])                                \
759     copy(current_Jx [0:Jsize], current_Jy [0:Jsize], current_Jz [0:Jsize],  \
760          spec_ix [0:spec_np], spec_iy [0:spec_np], spec_x [0:spec_np],      \
761          spec_y [0:spec_np])
762   {
763   #pragma acc parallel
764   {
765   #pragma acc loop
766   for (int i = 0; i < spec_np; i++) {
767     // Load particle momenta
768     t_part_data ux = spec_ux[i];
769     t_part_data uy = spec_uy[i];
770     t_part_data uz = spec_uz[i];
       ...
952       #pragma acc atomic update
953       current_Jz[vp_ix[k] + 1 + nrow * (vp_iy[k] + 1)] +=
954           vp_qvz[k] * (S0x[1] * S0y[1] + S1x[1] * S1y[1] +
955                        (S0x[1] * S1y[1] - S1x[1] * S0y[1]) / 2.0f);
956     }
957
958     ////////////////////////////////////
959     //   dep_current_zamb inlined  END //
960     ////////////////////////////////////
961
962     // Store results
963     spec_x[i] = x1;
964     spec_y[i] = y1;
965     spec_ix[i] += di;
966     spec_iy[i] += dj;
967   }
968   } // end parallel
969   } // end data
```

Benchmarking on Perlmutter using Nvidia & GNU toolchains

Version	#threads	Nvidia @ GPU node NVC 22.7 Time (sec) [Speedup]	GNU @ GPU node GCC 12.1 Time (sec) [Speedup]
EM2D #0 (serial version)	n/a	9.95 s	11.44 s
EM2D #1 (outlining, gather-scatter data)	n/a	10.95 s [0,91x]	11.72 s [0,98x]
EM2D #2 (inlining deb_current_zamb)	n/a	10.67 s [0,93x]	11.46 s [0,99x]
EM2D #3 (refactor AoS)	n/a	10.23 s [0,97x]	11.32 s [1,01x]
EM2D #4 (apply loop fission)	n/a	6.47 s [1,54x]	9.41 s [1,21x]
EM2D #5a (basic CPU OpenMP Atomic)	32 threads	5.48 s [1,82x]	n/a (gcc 7.5 yields 6.33 seconds, 1.82x speedup)
EM2D #6a (basic GPU OpenMP Atomic)	n/a	n/a (nvc requires schedule(static) and Codee generates schedule(auto))	n/a
EM2D #7a (basic GPU OpenACC Atomic)	n/a	n/a (nvc 22.7 combined with Codee 2023.1 raises a runtime error under investigation)	9.26s [1,23x]

OpenMP multithreading yields speedup 1.82x

OpenACC offload yields speedup 1.23x, a starting point for further optimization

Parallel ZPIC/em2d: from serial to CPU & GPU...

- ✓ EM2D #7a (Basic GPU+OpenACC Atomic)
- ✓ EM2D #6a (Basic GPU+OpenMP Atomic)
- ✓ EM2D #5a (Basic CPU+OpenMP Atomic)
- ✓ EM2D #4 (loop fission)
- ✓ EM2D #3 (change Array of Structs by separate arrays)
- ✓ EM2D #2 (inlining deb_current_zamb)
- ✓ EM2D #1 (outlining, gather-scatter data)
- ✓ EM2D #0 (serial version)

Further optimization of ZPIC is possible!

- Now we have a baseline implementation of ZPIC/em2d, using OpenMP/OpenACC on CPU/GPU.
- Optimizations for CPU+OpenMP:
  - You can try the performance of several parallelization strategies, like explicit privatization.
  - We suggest that first you parallelize the fist loop that arises after loop fission, which is a FORALL pattern that can be parallelized in a straightforward manner.
  - Next, we suggest that you use one unique parallel region for the two loops: it covers both the FORALL loop and the SPARSE REDUCTION loop.
  - And third, if your want to go a step beyond, move the parallel region outside the simulation loop.
- Optimizations for GPU+OpenMP/OpenACC:
  - We suggest that first you parallelize the fist loop that arises after loop fission, which is a FORALL patterns that can be parallelized in a straightforward manner.
  - Next, we suggest that you use one unique data movement region for the two loops: it covers both the FORALL loop and the SPARSE REDUCTION loop.
  - And third, if your want to go a step beyond, move the data movement region outside the simulation loop.
- Note the similarities between the optimized version: for CPU you reason about minimizing parallel regions, and for GPUs you reason about minimizing data movement.
- Code the corresponding version and benchmark the performance improvements on the computer.

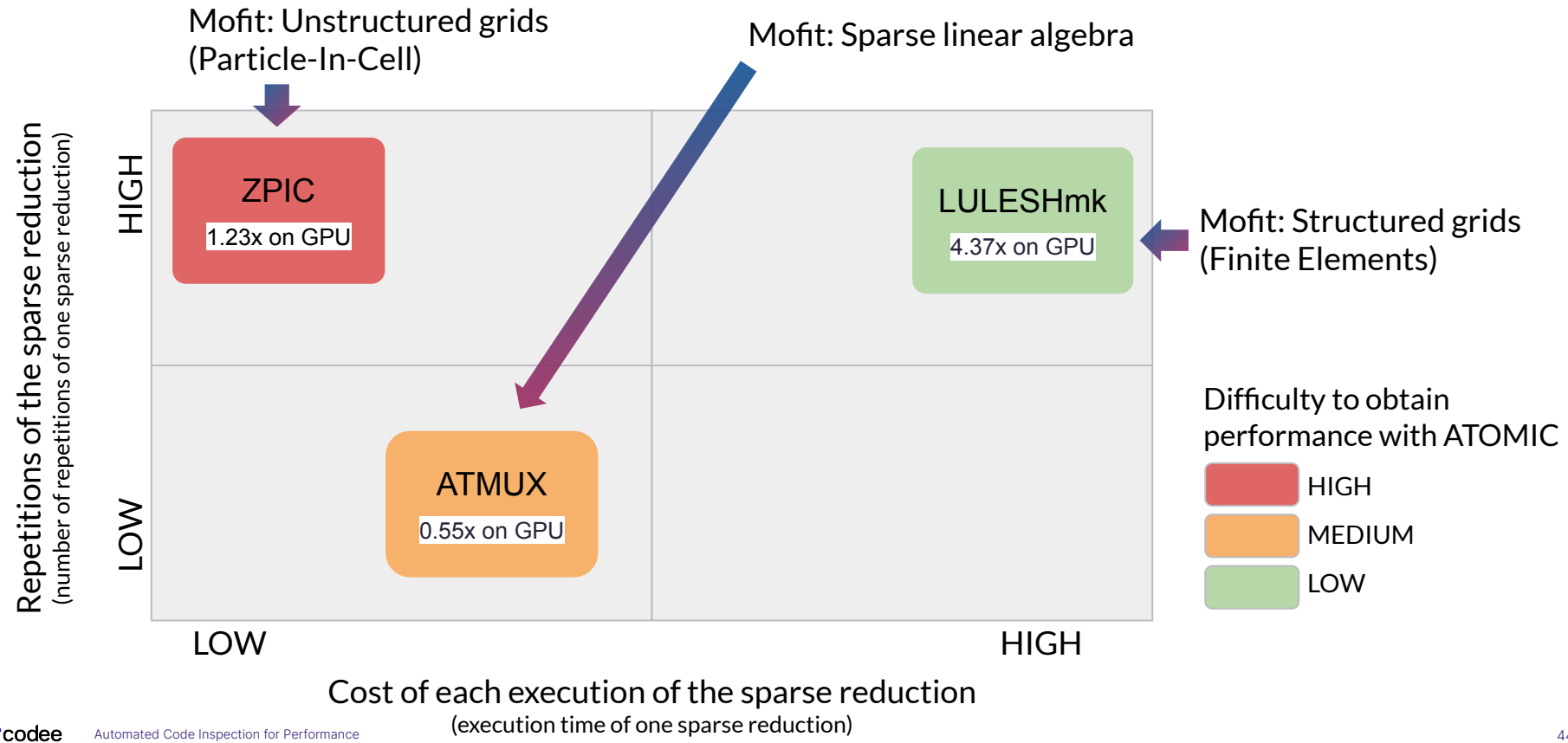
Final considerations about optimization of Sparse Reductions!

```
// Convergence loop of the simulation
for(iter=0, err = tol;
    err >= tol && iter < iter max;
    iter++){

  // Forall "B"
  for (j=0; j<n; j++ ) {
    B[j] = j;
  }

  // Sparse reduction "A"
  for (j=0; j<n; j++ ) {
    A[C[j]] += B[j];
  }
}
```

	ATMUX (n = 17000)	LULESHmk (n = 30000)	ZPIC (128x128 cells, 8 particles per cell)
No. repetitions (final value of "iter")	10	223.680.000	65.000.000
No. loop iterations of sparse reduction (value of "n")	17.000	30.000	3
Execution time of one loop iteration	LOW	HIGH	TINY

- **ATMUX: it is challenging due to large memory requirements for sparse matrices...**
  - Note that the cost of each sparse reduction is directly related to the memory consumption of the sparse matrix, which leads to high parallelization overheads due to data movement on the GPU.
- **LULESHmk: it is challenging to find parallelization opportunities in the source code...**
  - In this case the sparse reduction is time-consuming and so the parallelization overhead has less impact on performance.
  - So it is easier to obtain performance for LULESHmk than for ZPIC.
- **ZPIC: it is challenging to obtain performance on both CPU and GPU because...**
  - The parallelization overhead of the sparse reduction is very high compared to the execution time of one run of the sparse reduction.
  - So parallelization overhead must be amortized across all of the repetitions (e.g. move data movement and thread creation out of the loop).
  - This also make it key to parallelize other code sections that interact with the sparse reduction (e.g. parallelize the forall pattern).

The Performance Quadrant of Sparse Reductions



codee

www.codee.com

info@codee.com

Subscribe: codee.com/newsletter/

USA - Spain

codee_com

company/codee-com/