Using OpenACC for Parallelizing Lawrence Livermore loops

Ahmad Lashgar
ECE Department
University of Victoria

Abstract - In this work, we use OpenACC directives for parallelizing Lawrence Livermore loops (LLL) on GPUs. Among the 24 kernels available in LLL, we could apply OpenACC directives to 18 kernels. We explain why the remaining 6 kernels cannot be parallelized by OpenACC directives. On average, our evaluations show 238% speedup from OpenACC, over the baseline serial version.

1. Introduction

Livermore loops [1] is a benchmark for parallel computers. It consists of 24 loops where each loop represents kernel of one scientific application. The benchmark is originally written in Fortran, but the C version of the benchmark is also available [2]. Here is the list of kernels:

1. Hydro fragment
2. ICCG excerpt (Incomplete Cholesky Conjugate Gradient)
3. Inner product
4. Banded linear equations
5. Tri-diagonal elimination, below diagonal
6. General linear recurrence equations
7. Equation of state fragment
8. ADI integration
9. Integrate predictors
10. Difference predictors
11. First sum
12. First difference
13. 2-D PIC (Particle In Cell)
14. 1-D PIC (Particle In Cell)
15. Casual Fortran. Development version
16. Monte Carlo search loop
17. Implicit, conditional computation
18. 2-D explicit hydrodynamics fragment
19. General linear recurrence equations
20. Discrete ordinates transport, conditional recurrence on xx
21. Matrix*matrix product
22. Planckian distribution
23. 2-D implicit hydrodynamics fragment
24. Find location of first minimum in array

OpenACC is an standard directive-based programming model for programming accelerators [3]. Compilers are available to run OpenACC applications on different accelerators. For example, IPMACC compiler [4] is able to run OpenACC applications on NVIDIA GPUs, AMD GPUs, and Intel XeonPhi co-processors.

2. OpenACC Acceleration

We build our OpenACC-accelerated Lawrence Livermore loops on top of [2]. We modified the kernels to create parallelism and make the kernels compatible with OpenACC loop-parallelism notation. Modifications include flattening multi-dimensional arrays, applying recursive doubling transformation [5], loop fusion, loop interchange, and loop splitting optimizations. Using these modifications, we could apply OpenACC directives for accelerating 18 kernels (out of 24 kernels). We could not parallelize the remaining 6 kernels for following reasons:

- Write dependencies between loop iterations, e.g. multiple writes to the same location. OpenACC has weak memory model and multiple writes to the same location might produce inconsistent results. This applies to kernel #20, and #23.
- Loop includes sophisticated control flow statements (goto, break, and continue). These statements are not valid in OpenACC, since the number of loop iterations is unknown upon entering the loop. This limitation applies to kernel #16 and #17.
- Read-after-write data dependency between loop iterations. This applies to kernel #5 and #19.

We explain our enhancement for each kernel in detail in Appendix A. Full source code can be found online at [6].

3. Methodology

We run our evaluations on two different GPUs; NVIDIA Tesla M2070 and NVIDIA Tesla K20c GPUs. We compare the speedup over serial version. We run serial on Intel Xeon E5649. We use IPMACC 1.0.3b [4] for compiling OpenACC directives on GPUs and GNU GCC v4.4.1 for compiling the serial version.

4. Results

Figure 1 reports the speedup achieved by accelerating kernels using OpenACC. The speedup is calculated by dividing CPU runtime by GPU runtime. On average, 2X speedup is achieved over all the kernels. One common reasons that the speedup are not very high is that the computations in all kernels are in double-precision floating point. GPUs are slow in

executing double-precision floating point since the double-precision units are half of the single-precision units on the chip. Below we investigate the speedup achieved under each kernel specifically.

As investigate by a previous work [7], not all kind of workloads map very well to GPUs. Indeed, in some case, executing the kernel serially on the CPU is faster. We observe slowdown in kernel #1, #2, #3, #4, #6, #8, #9, #10, #11, and #12.

- In kernel #6, #8, #9, and #10, memory accesses are *row-wise*. This severely slows the performance down on GPUs, because of the coalescing hardware. To have the best performance on GPU, neighbor threads should access neighbor words to minimize memory accesses by a warp (down to one memory line per warp). However, this row-wise memory pattern demands 32 memory lines per warp.
- In kernel #3, #4, #6, and #11, the parallel loop includes a *reduction*. This reduction limits the parallelism of the kernel and slows down the performance.
- In kernel #1 and #12, the body of the kernel is very tiny. Our evaluations shows these kernels return increasingly higher speedup on larger datasets. However, the GPU memory is limited and may not run larger problem sizes (1279MB of DRAM can fit up to three double-precision arrays with 56-million elements).
- In kernel #2, the parallelism is limited. This kernel iteratively shrinks the problem size by dividing it by 2. The first iteration is fast on the GPU bu remaining iterations run very small problem sizes which is expensive to execute on GPU.

We observe very high speedup under kernel #7, #14, #15, #18, #21, #22, and #24. Memory accesses are well-coalesced in these kernels, which opens and opportunity to exploit memory bandwidth on the GPU efficiently. Beside this, kernel #21 is the only kernel with two nested parallelizable loops. This opens an opportunity to an immense parallelization and speedup. In kernel 24, there is no double-precision operation and all operations are memory accesses and control flow statements.

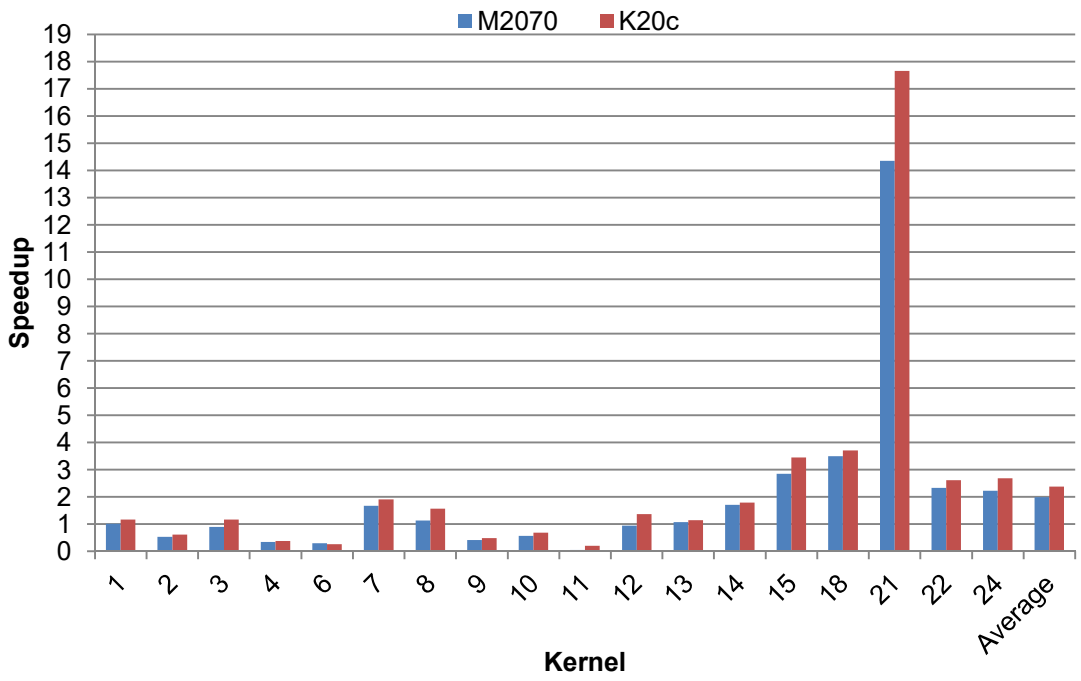

Figure 1. Speedup of OpenACC over the serial.

5. Related Work

We did not find previous work investigating performance of Lawrence Livermore loops on GPUs. Foe [5] investigate parallelization opportunities in all Lawrence Livermore loops. While they don't evaluate performance advantages from parallelization, they explain where vectorization, task-level, or loop-level parallelization are applicable. Guccione and Gonzalez [8] classify Lawrence Livermore loops into Fully Vectorizable (kernel #7, #1, #9, #3, #8, and #18), Partially Vectorizable (kernel #21, #22, #4, #12, #10, and #2), Unvectorizable (kernel #14, #6, #19, #23, #5, #11, #20, and #17), and Unstructured (#16, #15, #13, and #24) and evaluates performance of reconfigurable architectures under these kernels. The kernels that we could not parallelize in this work fall in Unvectorizable or Unstructured classes. Cann and Foe [9] parallelize the kernels on CPUs using SISAL programming model. They achieve an average speedup of 2.55X using five processors.

6. Conclusion

In this work, we analyzed the use of OpenACC directives for parallelizing Lawrence Livermore loops on GPUs. We found that 17 kernels are parallelizable. We found that kernels with regular memory access pattern and vast parallelization return highest speedup. We also found the reasons that lower down the speedup. Reasons are namely double-precision floating point operations, row-wise memory access pattern, reduction, and

limited parallelism. We also found that GPUs return increasingly higher speedup under larger datasets. However, GPU memory size limits the problem size that can run on the GPU.

References

[1] F. H. McMahon. Livermore fortran kernels: A computer test of numerical performance range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, Livermore, CA, December 1986. NTIS report #DE87009360

[2] "Livermore C". Available: http://www.netlib.org/benchmark/livermorec

[3] "The OpenACC Application Programming Interface," Available: http://www.openacc.org/content/openacc-2-5-final-specification

[4] A. Lashgar, A. Majidi, and A. Baniasadi, "IPMACC: open source OpenACC to CUDA/OpenCL translator," CoRR, vol. abs/1412.1127, 2014. [Online]. Available: http://arxiv.org/abs/1412.1127

[5] J. T Feo, "An analysis of the computational and parallel complexity of the Livermore Loops", Parallel Computing, Volume 7, Issue 2, 1988, Pages 163-185, ISSN 0167-8191, http://dx.doi.org/10.1016/0167-8191(88)90037-3.

[6] A. Lashgar, "Lawrence Livermore loops in OpenACC", Available: http://ece.uvic.ca/~lashgar/files/lfk_m.cpp

[7] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. 2010. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *Proceedings of the 37th annual* international symposium on Computer architecture (ISCA '10). ACM, New York, NY, USA, 451-460. DOI=http://dx.doi.org/10.1145/1815961.1816021

[8] S. A. Guccione and M. J. Gonzalez, " Supercomputing with reconfigurable architectures", In Proceedings of 5th International Workshop on Field-Programmable Logic and Applications, FPL '95 Oxford, United Kingdom, August 29--September 1, 1995. pp. 389-398

[9] D. Cann and J. Feo, "SISAL versus Fortran: a comparison using the Livermore Loops," *Proceedings SUPERCOMPUTING '90*, New York, NY, 1990, pp. 626-636.

Appendix A

We modified the code to accept two input arguments: i) kernel ID and ii) problem size to run. The outline of the program looks like this:

```
int main(int argc, char* argv[]){
  int kernel_id, problem_size;
  if(argc!=3) return -1;
  sscanf(argv[1], "%d", &kernel_id);
  sscanf(argv[1], "%d", &problem_size);
  switch(kernel_id){
    case 1: // run kernel #1 and report execution time
    break;
    case 2: // run kernel #2 and report execution time
    break;
    ...
    case 24: // run kernel #2 and report execution time
    break;
    default:
      printf("unknown kernel id %d\n", kernel_id);
      return -1;
  }
  return 0;
}
```

While different cases of the switch-case statement above run different kernels, they have a very similar structure. This structure is outlined below:

```
...
        case N: // run kernel #N and report execution time
        {
            char *kernelname = (char*)"<name of the kernel>";

            // declare & set parameters
            int loop = 1, n = probelm_size;
            printf("Testing %s; Parameters: %d %d\n", kernelname, loop, n);

            // allocate memories using init_double, init_int, init_float
            // double *<pointer> = init_double(<size>);
            // long   *<pointer> = init_long(<size>);

            // run the kernel
            printf("Running the kernel:\n");
            kernel_before_run();
            for ( l=1 ; l<=loop ; l++ ) {
              // KERNEL BODY
            }
            kernel_after_run();

            // dump output to file using dump_double, dump_int, dump_float
            // dump_double(<pointer>, <size>, kernel_id, argv[0], 1);
            // dump_long(<pointer>, <size>, kernel_id, argv[0], 1);

            // free memory
            free(<pointer>);
        }
        break;
...
```

KERNEL BODY is kernel specific and is where we apply OpenACC directives to parallelize the kernel on GPU. We explain this part for each kernel specifically later. In the structure above, we wrote five routines to simply memory allocation (init_double() and init_long()), runtime measurement (kernel_before_run() and kernel_after_run()), and correctness verification (dump_double() and dump_long()). Below we explain each of these routines:

- Memory Allocation: double and long are common datatypes among all kernels. We wrote init_double() and init_long() to allocate and initialize an array of certain size. The array size is passed as an argument in to these functions. Allocated initialized memory location is returned as a pointer. Notice that we use this function for multi-dimensional arrays too, since we flatten multi-dimensional arrays. Here is the declaration of these functions:

```
double* init_double(int n_element,
      long vrange_low = 0,
      long vrange_hi = 100,
      double factor = 1.0){
    double *ptr = (double*)malloc(sizeof(double)*n_element);
    int i = 0;
    for(i=0; i<n_element; ++i){
        ptr[i] = factor*(vrange_low+(i%(vrange_hi-vrange_low+1)));
    }
    return ptr;
}

long* init_long(int n_element,
      long vrange_low = 0,
      long vrange_hi = 100){
    long *ptr = (long*)malloc(sizeof(long)*n_element);
    int i = 0;
    for(i=0; i<n_element; ++i){
        ptr[i] = vrange_low+(i%(vrange_hi-vrange_low+1));
        if(ptr[i]>vrange_hi || ptr[i]<vrange_low){
            printf("failed\n");
            exit(1);
        }
    }
    return ptr;
}
```

- Runtime Measurement: We wrote to functions to start and stop timer for measuring the runtime of a code block, e.g. kernel. kernel_before_run() start the timer and kernel_after_run() stops the timer and prints the elapsed time on stdout. Here is the declaration of these functions:

```
struct timeval start, stop;
void kernel_before_run(){
    gettimeofday(&start, 0);
}
void kernel_after_run(){
    gettimeofday(&stop, 0);
    long long unsigned elapsed = (stop.tv_sec-start.tv_sec)*1000000 +
stop.tv_usec-start.tv_usec;
    printf("elapsed time> %llu us\n", elapsed);
}
```

- Correctness Verification: For verifying the correctness of our OpenACC version, we dump the output on to the disk and compare the results from CPU and GPU versions byte-by-byte. We wrote dump_double() and dump_long() for dumping double and long datatypes on to the disk. Here is the declaration of these functions:

```
void dump_double(double* reference, int n_element, int kernel_id, char*
pathprefix, int output_id)
{
    #ifdef VERIFY
    char dirpath[50];
    sprintf(dirpath, "./output/%s/", pathprefix);
    mkdir("./output/", 0777);
    mkdir(dirpath, 0777);
    char fname[60];
    sprintf(fname, "%s/output_kernel_%d_%d.txt", dirpath, kernel_id,
output_id);
    FILE *f = fopen(fname, "w");
    int i;
    for(i=0; i<n_element; ++i){
        fprintf(f, "%40.20f\n", reference[i]);
    }
    fclose(f);
    #endif
}
void dump_long(long* reference, int n_element, int kernel_id, char*
pathprefix, int output_id)
{
    #ifdef VERIFY
    char dirpath[50];
    sprintf(dirpath, "./output/%s/", pathprefix);
    mkdir("./output/", 0777);
    mkdir(dirpath, 0777);
    char fname[60];
    sprintf(fname, "%s/output_kernel_%d_%d.txt", dirpath, kernel_id,
output_id);
    FILE *f = fopen(fname, "w");
    int i;
    for(i=0; i<n_element; ++i){
        fprintf(f, "%lld\n", reference[i]);
    }
    fclose(f);
    #endif
}
```

In the rest of this appendix we focus on KERNEL BODY and explain modifications and directives we applied for parallelizing each kernel.

Kernel 1: Hydro fragment

Original code:

```
// original version in [2]
for ( l=1 ; l<=loop ; l++ ) {
    for ( k=0 ; k<n ; k++ ) {
        x[k] = q + y[k]*( r*z[k+10] + t*z[k+11] );
    }
}
```

Investigating parallelism:

The most-outer loop iteratively calls the kernel. The inner loop can be executed in parallel as output (x) is independent from inputs (y and z).

Modifications:

We made following modifications to parallelize this kernel:

- We calculated the size of arrays from boundaries of the loops.
- We used the size calculated above and added OpenACC data directive on top of the outer loop to copy in y and into the accelerator and copy out x from the accelerator.
- We added OpenACC kernels and loop directives on top of the most-inner loop [1]

OpenACC version:

```
// our OpenACC version
#pragma acc data pcopyin(y[0:n], z[0:n+11]) pcopyout(x[0:n])
for ( l=1 ; l<=loop ; l++ ) {
    #pragma acc kernels
    #pragma acc loop independent
    for ( k=0 ; k<n ; k++ ) {
        x[k] = q + y[k]*( r*z[k+10] + t*z[k+11] );
    }
}
```

[1] independent clause is a hint for compiler to skip checking the loop for possible parallelization and forces the compiler to parallelize the loop

Kernel 2: ICCG excerpt (Incomplete Cholesky Conjugate Gradient)

Original code:

```
// original version in [2]
for ( l=1 ; l<=loop ; l++ ) {
    ii = n;
    ipntp = 0;
    do {
        ipnt = ipntp;
        ipntp += ii;
        ii /= 2;
        i = ipntp - 1;
        for ( k=ipnt+1 ; k<(ipntp-1) ; k=k+2 ) {
            i++;
            x[i] = x[k] - v[k  ]*x[k-1] - v[k+1]*x[k+1];
        }
    } while ( ii>0 );
}
```

Investigating parallelism:

The most-outer loop iteratively calls the kernel. The while loop cannot be parallelize since a range of x is calculated by one iteration and immediately used by the next iteration. The most-inner for loop can be executed in parallel as different ranges of x is used as input and output.

Modifications:

We made following modifications to parallelize this kernel:

- We calculated the size of arrays from boundaries of the loops.
- We used the size calculated above and added OpenACC data directive on top of the outer loop to copy in x and v into the accelerator and copy out x and v from the accelerator.
- We added OpenACC kernels and loop directives on top of the most-inner for loop.

OpenACC version:

```
// our OpenACC version
#pragma acc data pcopy(x[0:2*n], v[0:2*n])
for ( l=1 ; l<=loop ; l++ ) {
    int i = 0, ii = n, ipnt = 0, ipntp = 0;
    do {
        ipnt = ipntp;
        ipntp += ii;
        ii /= 2;
        int ibase = ipntp - 1 ;
        #pragma acc kernels
        #pragma acc loop independent
        for ( k=ipnt+1 ; k<(ipntp-1) ; k=k+2 ) {
            int i = ibase + k - (ipnt+1) + 1;
            x[i] = x[k] - v[k  ]*x[k-1] - v[k+1]*x[k+1];
        }
    } while ( ii>1 );
}
```

Kernel 3: Inner product

Original code:

```
// original version in [2]
for ( l=1 ; l<=loop ; l++ ) {
    q = 0.0;
    for ( k=0 ; k<n ; k++ ) {
        q += z[k]*x[k];
    }
}
```

Investigating parallelism:

The most-outer loop iteratively calls the kernel. The most-inner for loop can be executed in parallel as long as writes to q are atomic. This can be done using reduction.

Modifications:

We made following modifications to parallelize this kernel:

- We calculated the size of arrays from boundaries of the loops.
- We used the size calculated above and added OpenACC data directive on top of the outer loop to copy in x and z into the accelerator.
- We added OpenACC kernels and loop directives on top of the most-inner for loop. Plus, we added reduction sum clause on the loop directive to accumulate writes to q. Notice that reduction clause implies copying q out of the accelerator. Therefore, copyout clause is not used in this kernel.

OpenACC version:

```
// our OpenACC version
#pragma acc data pcopyin(z[0:n], x[0:n])
for ( l=1 ; l<=loop ; l++ ) {
    q = 0.0;
    #pragma acc kernels
    #pragma acc loop independent reduction(+:q)
    for ( k=0 ; k<n ; k++ ) {
        q += z[k]*x[k];
    }
}
```

Kernel 4: Banded linear equations

Original code:

```
// original version in [2]
m = ( 1001-7 )/2;
for ( l=1 ; l<=loop ; l++ ) {
    for ( k=6 ; k<1001 ; k=k+m ) {
        lw = k - 6;
        temp = x[k-1];
        for ( j=4 ; j<n ; j=j+5 ) {
            temp -= x[lw]*y[j];
            lw++;
        }
        x[k-1] = y[4]*temp;
    }
}
```

Investigating parallelism:

The most-outer loop iteratively calls the kernel. The second outer loop cannot be parallelized since loop iterations depend on a range of x. The most-inner for loop can be executed in parallel as long as writes to temp are atomic. This can be done using reduction.

Modifications:

We made following modifications to parallelize this kernel:

- We calculated the size of arrays from boundaries of the loops.
- We used the size calculated above and added OpenACC data directive on top of the outer loop to copy in y into the accelerator.
- We used the size calculated above and added OpenACC data directive on top of the most-inner loop to copy in x into the accelerator and copy x out from the accelerator after completion of the kernel.
- We break the -= operation and replaced it with += sum reduction and introduced variable sum.
- We added OpenACC kernels and loop directives on top of the most-inner for loop. Plus, we added reduction sum clause on the loop directive to accumulate writes to sum. Notice that reduction clause implies copying sum out of the accelerator. Therefore, copyout clause is not used in this kernel.

OpenACC version:

```
// our OpenACC version
int m = ( 1001-7 )/2;
#pragma acc data pcopyin(y[0:n])
for ( l=1 ; l<=loop ; l++ ) {
    for ( k=6 ; k<1001 ; k=k+m ) {
        temp = x[k-1];
        double sum = 0;
        #pragma acc kernels pcopy(x[0:(1001+n)])
        #pragma acc loop independent reduction(+:sum)
        for ( j=4 ; j<n ; j=j+5 ) {
            int lw = k - 6 - 4 + j;
            sum += x[lw]*y[j];
        }
        temp -= sum;
        x[k-1] = y[4]*temp;
```

```
        }
}
```

Kernel 5: Tri-diagonal elimination, below diagonal

Original code:

```
// original version in [2]
for ( l=1 ; l<=loop ; l++ ) {
    for ( i=1 ; i<n ; i++ ) {
        x[i] = z[i]*( y[i] - x[i-1] );
    }
}
```

Investigating parallelism:

The most-outer loop iteratively calls the kernel. The most-inner for loop can be parallelized by recalculating x[i-1]. As we evaluate in Kernel 11, this is not beneficial in OpenACC and we avoided parallelizing this kernel. This is because recalculating transforms the complexity of algorithm from O(n) to O(n logn). Parallelizing this O(n logn) algorithm is beneficial if there are n parallel cores that each can reduce an array of n elements in log(n). However, in OpenACC, we need to call one kernel to reduce an array of n in O(logn). This implies that we need to call n kernels on accelerator. This is obviously very expensive than simply calculating the kernel on the CPU in O(n).

Kernel 6: General linear recurrence equations

Original code:

```
// original version in [2]
for ( l=1 ; l<=loop ; l++ ) {
    for ( i=1 ; i<n ; i++ ) {
        for ( k=0 ; k<i ; k++ ) {
            w[i] += b[k][i] * w[(i-k)-1];
        }
    }
}
```

Investigating parallelism:

The most-outer loop iteratively calls the kernel. The second outer loop cannot be parallelize since it depends on a range of w. The most-inner for loop can be executed in parallel as long as write to w[i] is atomic. w[i] can be replace by a simple variable to apply parallel reduction.

Modifications:

We made following modifications to parallelize this kernel:

- We calculated the size of arrays from boundaries of the loops.
- We used the size calculated above and added OpenACC data directive on top of the outer loop to copy in b and w into the accelerator. Notice that latest copy of w[] is on the host and copyout is not needed.
- We replace w[i] with variable sum to create reduction in the most-inner loop.
- We added OpenACC kernels and loop directives on top of the most-inner for loop. Plus, we add reduction sum clause on variable sum. Notice that reduction clause implies copying sum out of the accelerator. Therefore, copyout clause is not used in this kernel.

OpenACC version:

```
// our OpenACC version
#pragma acc data pcopyin(b[0:n*n],w[0:n])
for ( l=1 ; l<=loop ; l++ ) {
    for ( i=1 ; i<n ; i++ ) {
        double sum = 0;
        #pragma acc kernels pcopyin(w[0:n])
        #pragma acc loop independent reduction(+:sum)
        for ( k=0 ; k<i ; k++ ) {
            sum += b[k*n+i] * w[(i-k)-1];
        }
        w[i] += sum;
    }
}
```

Kernel 7: Equation of state fragment

Original code:

```
// original version in [2]
for ( l=1 ; l<=loop ; l++ ) {
    for ( k=0 ; k<n ; k++ ) {
        x[k] = u[k] + r*( z[k] + r*y[k] ) +
               t*( u[k+3] + r*( u[k+2] + r*u[k+1] ) +
                  t*( u[k+6] + r*( u[k+5] + r*u[k+4] ) ) );
    }
}
```

Investigating parallelism:

The most-outer loop iteratively calls the kernel. The most-inner for loop can be executed in parallel as output (x) is independent from inputs (u, z, and y).

Modifications:

We made following modifications to parallelize this kernel:

- We calculated the size of arrays from boundaries of the loops.
- We used the size calculated above and added OpenACC data directive on top of the outer loop to copy in u, z, and y into the accelerator and copy x out of the accelerator.
- We added OpenACC kernels and loop directives on top of the most-inner for loop.

OpenACC version:

```
// our OpenACC version
#pragma acc data pcopyin(u[0:n+6], z[0:n], y[0:n]) pcopyout(x[0:n])
for ( l=1 ; l<=loop ; l++ ) {
    #pragma acc kernels
    #pragma acc loop independent
    for ( k=0 ; k<n ; k++ ) {
        x[k] = u[k] + r*( z[k] + r*y[k] ) +
               t*( u[k+3] + r*( u[k+2] + r*u[k+1] ) +
               t*( u[k+6] + r*( u[k+5] + r*u[k+4] ) ) );
    }
}
```

Kernel 8: ADI integration

Original code:

```
// original version in [2]
for ( l=1 ; l<=loop ; l++ ) {
    nl1 = 0;
    nl2 = 1;
    for ( kx=1 ; kx<3 ; kx++ ){
        for ( ky=1 ; ky<n ; ky++ ) {
           du1[ky] = u1[nl1][ky+1][kx] - u1[nl1][ky-1][kx];
           du2[ky] = u2[nl1][ky+1][kx] - u2[nl1][ky-1][kx];
           du3[ky] = u3[nl1][ky+1][kx] - u3[nl1][ky-1][kx];
           u1[nl2][ky][kx]=
              u1[nl1][ky][kx]+a11*du1[ky]+a12*du2[ky]+a13*du3[ky] + sig*
                 (u1[nl1][ky][kx+1]-2.0*u1[nl1][ky][kx]+u1[nl1][ky][kx-1]);
           u2[nl2][ky][kx]=
              u2[nl1][ky][kx]+a21*du1[ky]+a22*du2[ky]+a23*du3[ky] + sig*
                 (u2[nl1][ky][kx+1]-2.0*u2[nl1][ky][kx]+u2[nl1][ky][kx-1]);
           u3[nl2][ky][kx]=
              u3[nl1][ky][kx]+a31*du1[ky]+a32*du2[ky]+a33*du3[ky] + sig*
                 (u3[nl1][ky][kx+1]-2.0*u3[nl1][ky][kx]+u3[nl1][ky][kx-1]);
        }
    }
}
```

Investigating parallelism:

The most-outer loop iteratively calls the kernel. The inner loops can be executed in parallel as outputs are written to different dimensions than the inputs.

Modifications:

We made following modifications to parallelize this kernel:

- We calculated the size of arrays from boundaries of the loops.
- We used the size calculated above and added OpenACC data directive on top of the outer loop to copy in u1, u2, u3, du1, du2, and du3 into the accelerator and copy u1, u2, u3, du1, du2, and du3 out of the accelerator. Notice that we flatten the three dimensional arrays and use a macro A3D to simplify array access notation.
- We added OpenACC kernels and loop directives on top of the most-inner for loop.

OpenACC version:

```
// our OpenACC version
#define A3D(ARRAY3DFLAT,a,bb,c,adim,bdim,ccdim)
ARRAY3DFLAT[(a)*((bdim)*(ccdim))+(bb)*(ccdim)+(c)]
#pragma acc data pcopy(u1[0:total_size], u2[0:total_size], u3[0:total_size],\
   du1[0:n], du2[0:n], du3[0:n])
for ( l=1 ; l<=loop ; l++ ) {
    nl1 = 0;
    nl2 = 1;
    for ( kx=1 ; kx<3 ; kx++ ){
        #pragma acc kernels
        #pragma acc loop independent
        for ( ky=1 ; ky<n ; ky++ ) {
           du1[ky] = A3D(u1,nl1,ky+1,kx,dim_1_size,dim_2_size,dim_3_size) -
A3D(u1,nl1,ky-1,kx,dim_1_size,dim_2_size,dim_3_size);
           du2[ky] = A3D(u2,nl1,ky+1,kx,dim_1_size,dim_2_size,dim_3_size) -
A3D(u2,nl1,ky-1,kx,dim_1_size,dim_2_size,dim_3_size);
```

```
            du3[ky] = A3D(u3,nl1,ky+1,kx,dim_1_size,dim_2_size,dim_3_size) -
A3D(u3,nl1,ky-1,kx,dim_1_size,dim_2_size,dim_3_size);
            A3D(u1,nl2,ky,kx,dim_1_size,dim_2_size,dim_3_size)=

A3D(u1,nl1,ky,kx,dim_1_size,dim_2_size,dim_3_size)+a11*du1[ky]+a12*du2[ky]+a13*du3[ky]
+ sig*
                  (A3D(u1,nl1,ky,kx+1,dim_1_size,dim_2_size,dim_3_size)-
2.0*A3D(u1,nl1,ky,kx,dim_1_size,dim_2_size,dim_3_size)+A3D(u1,nl1,ky,kx-
1,dim_1_size,dim_2_size,dim_3_size));
            A3D(u2,nl2,ky,kx,dim_1_size,dim_2_size,dim_3_size)=

A3D(u2,nl1,ky,kx,dim_1_size,dim_2_size,dim_3_size)+a21*du1[ky]+a22*du2[ky]+a23*du3[ky]
+ sig*
                  (A3D(u2,nl1,ky,kx+1,dim_1_size,dim_2_size,dim_3_size)-
2.0*A3D(u2,nl1,ky,kx,dim_1_size,dim_2_size,dim_3_size)+A3D(u2,nl1,ky,kx-
1,dim_1_size,dim_2_size,dim_3_size));
            A3D(u3,nl2,ky,kx,dim_1_size,dim_2_size,dim_3_size)=

A3D(u3,nl1,ky,kx,dim_1_size,dim_2_size,dim_3_size)+a31*du1[ky]+a32*du2[ky]+a33*du3[ky]
+ sig*
                  (A3D(u3,nl1,ky,kx+1,dim_1_size,dim_2_size,dim_3_size)-
2.0*A3D(u3,nl1,ky,kx,dim_1_size,dim_2_size,dim_3_size)+A3D(u3,nl1,ky,kx-
1,dim_1_size,dim_2_size,dim_3_size));
        }
    }
}
```

Kernel 9: Integrate predictors

Original code:

```
// original version in [2]
for ( l=1 ; l<=loop ; l++ ) {
    for ( i=0 ; i<n ; i++ ) {
        px[i][0] = dm28*px[i][12] + dm27*px[i][11] + dm26*px[i][10] +
                   dm25*px[i][ 9] + dm24*px[i][ 8] + dm23*px[i][ 7] +
                   dm22*px[i][ 6] + c0*( px[i][ 4] + px[i][ 5]) + px[i][ 2];
    }
}
```

Investigating parallelism:

The most-outer loop iteratively calls the kernel. The inner loop can be executed in parallel as outputs are written to px[i][X] and inputs are read from px[i][X] and i is different for each loop iteration.

Modifications:

We made following modifications to parallelize this kernel:

- We calculated the size of arrays from boundaries of the loops.
- We used the size calculated above and added OpenACC data directive on top of the outer loop to copy in px into the accelerator and copy px out of the accelerator.
- We flattened the px array to create 1D arrays for OpenACC.
- We added OpenACC kernels and loop directives on top of the most-inner for loop.

OpenACC version:

```
// our OpenACC version
#pragma acc data pcopy(px[0:13*n])
for ( l=1 ; l<=loop ; l++ ) {
    #pragma acc kernels
    #pragma acc loop independent
    for ( i=0 ; i<n ; i++ ) {
        px[i*13+0] = dm28*px[i*13+12] + dm27*px[i*13+11] + dm26*px[i*13+10] +
                   dm25*px[i*13+9] + dm24*px[i*13+8] + dm23*px[i*13+7] +
                   dm22*px[i*13+6] + c0*( px[i*13+4] + px[i*13+5]) + px[i*13+2];
    }
}
```

Kernel 10: Difference predictors

Original code:

```
// original version in [2]
for ( l=1 ; l<=loop ; l++ ) {
    for ( i=0 ; i<n ; i++ ) {
        ar        =      cx[i][ 4];
        br        = ar - px[i][ 4];
        px[i][ 4] = ar;
        cr        = br - px[i][ 5];
        px[i][ 5] = br;
        ar        = cr - px[i][ 6];
        px[i][ 6] = cr;
        br        = ar - px[i][ 7];
        px[i][ 7] = ar;
        cr        = br - px[i][ 8];
        px[i][ 8] = br;
        ar        = cr - px[i][ 9];
        px[i][ 9] = cr;
        br        = ar - px[i][10];
        px[i][10] = ar;
        cr        = br - px[i][11];
        px[i][11] = br;
        px[i][13] = cr - px[i][12];
        px[i][12] = cr;
    }
}
```

Investigating parallelism:

The most-outer loop iteratively calls the kernel. The inner loop can be executed in parallel as outputs are written to px[i][X] and inputs are read from px[i][X] and i is different for each loop iteration.

Modifications:

We made following modifications to parallelize this kernel:

- We calculated the size of arrays from boundaries of the loops.
- We used the size calculated above and added OpenACC data directive on top of the outer loop to copy in cx and px into the accelerator and copy px out of the accelerator.
- We flattened the px array to create 1D arrays for OpenACC.
- We added OpenACC kernels and loop directives on top of the most-inner for loop.

OpenACC version:

```
// our OpenACC version
#pragma acc data pcopyin(cx[0:n*5],px[0:n*14]) pcopyout(px[0:n*14])
for ( l=1 ; l<=loop ; l++ ) {
    #pragma acc kernels
    #pragma acc loop independent
    for ( i=0 ; i<n ; i++ ) {
        double ar        =      cx[i*5+4];
        double br        = ar - px[i*14+4];
        px[i*14+4] = ar;
        double cr        = br - px[i*14+5];
        px[i*14+5] = br;
        ar        = cr - px[i*14+6];
        px[i*14+6] = cr;
```

```
        br        = ar - px[i*14+7];
        px[i*14+7] = ar;
        cr        = br - px[i*14+8];
        px[i*14+8] = br;
        ar        = cr - px[i*14+9];
        px[i*14+9] = cr;
        br        = ar - px[i*14+10];
        px[i*14+10] = ar;
        cr        = br - px[i*14+11];
        px[i*14+11] = br;
        px[i*14+13] = cr - px[i*14+12];
        px[i*14+12] = cr;
    }
}
```

Kernel 11: First sum

Original code:

```
// original version in [2]
for ( l=1 ; l<=loop ; l++ ) {
    x[0] = y[0];
    for ( k=1 ; k<n ; k++ ) {
        x[k] = x[k-1] + y[k];
    }
}
```

Investigating parallelism:

The most-outer loop iteratively calls the kernel. The most-inner for loop can be parallelized by recalculating x[i-1].

Modifications:

We made following modifications to parallelize this kernel:

- We calculated the size of arrays from boundaries of the loops.
- We used the size calculated above and added OpenACC data directive on top of the outer loop to copy in y into the accelerator. Notice that copyout is not required since the latest copy of x is on the host.
- We transform the algorithm from O(n) to O(n logn) to parallelize the reduction.
- We added OpenACC kernels and loop directives on top of the most-inner for loop. Plus, we add reduction sum clause on variable sum.

OpenACC version:

```
// our OpenACC version
#pragma acc data pcopy(y[0:n])
for ( l=1 ; l<=loop ; l++ ) {
    for ( k=0 ; k<n ; k++ ) {
        double sum = 0;
        #pragma acc kernels
        #pragma acc loop independent reduction(+:sum)
        for(i=0; i<(k+1); i++){
            sum+= y[i];
        }
        x[k] = sum;
    }
}
```

Kernel 12: First difference

Original code:

```
// original version in [2]
for ( l=1 ; l<=loop ; l++ ) {
    for ( k=0 ; k<n ; k++ ) {
        x[k] = y[k+1] - y[k];
    }
}
```

Investigating parallelism:

The most-outer loop iteratively calls the kernel. The inner loop can be executed in parallel as output (x) is independent from input (y).

Modifications:

We made following modifications to parallelize this kernel:

- We calculated the size of arrays from boundaries of the loops.
- We used the size calculated above and added OpenACC data directive on top of the outer loop to copy in y and into the accelerator and copy out x from the accelerator.
- We added OpenACC kernels and loop directives on top of the most-inner loop.

OpenACC version:

```
// our OpenACC version
#pragma acc data pcopyin(y[0:n+1]) pcopyout(x[0:n])
for ( l=1 ; l<=loop ; l++ ) {
    #pragma acc kernels
    #pragma acc loop independent
    for ( k=0 ; k<n ; k++ ) {
        x[k] = y[k+1] - y[k];
    }
}
```

Kernel 13: 2-D PIC (Particle In Cell)

Original code:

```
// original version in [2]
for ( l=1 ; l<=loop ; l++ ) {
    for ( ip=0 ; ip<n ; ip++ ) {
        i1 = p[ip][0];
        j1 = p[ip][1];
        i1 &= 64-1;
        j1 &= 64-1;
        p[ip][2] += b[j1][i1];
        p[ip][3] += c[j1][i1];
        p[ip][0] += p[ip][2];
        p[ip][1] += p[ip][3];
        i2 = p[ip][0];
        j2 = p[ip][1];
        i2 = ( i2 & 64-1 ) - 1 ;
        j2 = ( j2 & 64-1 ) - 1 ;
        p[ip][0] += y[i2+32];
        p[ip][1] += z[j2+32];
        i2 += e[i2+32];
        j2 += f[j2+32];
        h[j2][i2] += 1.0;
    }
}
```

Investigating parallelism:

The most-outer loop iteratively calls the kernel. A huge portion of the inner loop can be executed in parallel and the only dependency is on h[][] array.

Modifications:

We made following modifications to parallelize this kernel:

- We calculated the size of arrays from boundaries of the loops.
- We used the size calculated above and added OpenACC data directive on top of the outer loop to copy in p, b, c, y, and z into the accelerator and copy p out of the accelerator.
- We flattened p, b, a, y, z, and c array to create 1D arrays for OpenACC.
- We split the most-inner loop into two loops; one can be parallelized and the second is serially executed on CPU. We added OpenACC kernels and loop directives on top of the first new loop.

OpenACC version:

```
// our OpenACC version
#pragma acc data pcopyin(p[0:n*4], b[0:64*64], c[0:64*64], y[0:(64+32)], z[0:(64+32)])
for ( l=1 ; l<=loop ; l++ ) {
    #pragma acc kernels pcopyout(p[0:n*4])
    #pragma acc loop independent
    for ( ip=0 ; ip<n ; ip++ ) {
        i1 = p[ip*4+0];
        j1 = p[ip*4+1];
        i1 &= 64-1;
        j1 &= 64-1;
        p[ip*4+2] += b[j1*64+i1];
        p[ip*4+3] += c[j1*64+i1];
        p[ip*4+0] += p[ip*4+2];
```

```
        p[ip*4+1] += p[ip*4+3];
    }
    for ( ip=0 ; ip<n ; ip++ ) {
        i2 = p[ip*4+0];
        j2 = p[ip*4+1];
        i2 = ( i2 & 64-1 ) - 1 ;
        j2 = ( j2 & 64-1 ) - 1 ;
        p[ip*4+0] += y[i2+32];
        p[ip*4+1] += z[j2+32];
        i2 += e[i2+32];
        j2 += f[j2+32];
        h[j2*(64+40)+i2] += 1.0;
    }
}
```

Kernel 14: 1-D PIC (Particle In Cell)

Original code:

```
// original version in [2]
for ( l=1 ; l<=loop ; l++ ) {
    for ( k=0 ; k<n ; k++ ) {
        vx[k] = 0.0;
        xx[k] = 0.0;
        ix[k] = (long) grd[k];
        xi[k] = (double) ix[k];
        ex1[k] = ex[ ix[k] - 1 ];
        dex1[k] = dex[ ix[k] - 1 ];
    }
    for ( k=0 ; k<n ; k++ ) {
        vx[k] = vx[k] + ex1[k] + ( xx[k] - xi[k] )*dex1[k];
        xx[k] = xx[k] + vx[k]  + flx;
        ir[k] = xx[k];
        rx[k] = xx[k] - ir[k];
        ir[k] = ( ir[k] & 2048-1 ) + 1;
        xx[k] = rx[k] + ir[k];
    }
    for ( k=0 ; k<n ; k++ ) {
        rh[ ir[k]-1 ] += 1.0 - rx[k];
        rh[ ir[k]   ] += rx[k];
    }
}
```

Investigating parallelism:

The most-outer loop iteratively calls the kernel. Among the inner loops, first and second can be executed in parallel. Indirect array access in the third loop make loop dependency analysis statically nondeterministic.

Modifications:

We made following modifications to parallelize this kernel:

- We calculated the size of arrays from boundaries of the loops.
- We used the size calculated above and added OpenACC data directive on top of the first two inner loops to copy in grd, xx, vx, ix, xi, ex, dex, ex1, dex1, ir, and rx into the accelerator and copy them out of the accelerator.
- We added OpenACC kernels and loop directives on top of the first and second inner loop.

OpenACC version:

```
// our OpenACC version
for ( l=1 ; l<=loop ; l++ ) {
    #pragma acc data pcopy(grd[0:n],vx[0:n],xx[0:n],ix[0:n],xi[0:n], \
                           ex[0:n],dex[0:n],ex1[0:n],dex1[0:n],ir[0:n],rx[0:n])
    {
        #pragma acc kernels
        #pragma acc loop independent
        for ( k=0 ; k<n ; k++ ) {
            vx[k] = 0.0;
            xx[k] = 0.0;
            ix[k] = (long) grd[k];
            xi[k] = (double) ix[k];
            ex1[k] = ex[ ix[k] - 1 ];
            dex1[k] = dex[ ix[k] - 1 ];
```

```
        }
        #pragma acc kernels
        #pragma acc loop independent
        for ( k=0 ; k<n ; k++ ) {
            vx[k] = vx[k] + ex1[k] + ( xx[k] - xi[k] )*dex1[k];
            xx[k] = xx[k] + vx[k]  + flx;
            ir[k] = xx[k];
            rx[k] = xx[k] - ir[k];
            ir[k] = ( ir[k] & 2048-1 ) + 1;
            xx[k] = rx[k] + ir[k];
        }
    }
    for ( k=0 ; k<n ; k++ ) {
        rh[ ir[k]-1 ] += 1.0 - rx[k];
        rh[ ir[k]   ] += rx[k];
    }
}
```

Kernel 15: Casual Fortran. Development version

Original code:

```
// original version in [2]
for ( l=1 ; l<=loop ; l++ ) {
    ng = 7;
    nz = n;
    ar = 0.053;
    br = 0.073;
    for ( j=1 ; j<ng ; j++ ) {
        for ( k=1 ; k<nz ; k++ ) {
            if ( (j+1) >= ng ) {
                vy[j][k] = 0.0;
                continue;
            }
            if ( vh[j+1][k] > vh[j][k] ) {
                t = ar;
            }
            else {
                t = br;
            }
            if ( vf[j][k] < vf[j][k-1] ) {
                if ( vh[j][k-1] > vh[j+1][k-1] )
                    r = vh[j][k-1];
                else
                    r = vh[j+1][k-1];
                s = vf[j][k-1];
            }
            else {
                if ( vh[j][k] > vh[j+1][k] )
                    r = vh[j][k];
                else
                    r = vh[j+1][k];
                s = vf[j][k];
            }
            vy[j][k] = sqrt( vg[j][k]*vg[j][k] + r*r )* t/s;
            if ( (k+1) >= nz ) {
                vs[j][k] = 0.0;
                continue;
            }
            if ( vf[j][k] < vf[j-1][k] ) {
                if ( vg[j-1][k] > vg[j-1][k+1] )
                    r = vg[j-1][k];
                else
                    r = vg[j-1][k+1];
                s = vf[j-1][k];
                t = br;
            }
            else {
                if ( vg[j][k] > vg[j][k+1] )
                    r = vg[j][k];
                else
                    r = vg[j][k+1];
                s = vf[j][k];
                t = ar;
            }
            vs[j][k] = sqrt( vh[j][k]*vh[j][k] + r*r )* t / s;
        }
    }
}
```

Investigating parallelism:

The most-outer loop iteratively calls the kernel. Two inner loops can be executed in parallel.

Modifications:

We made following modifications to parallelize this kernel:

- We calculated the size of arrays from boundaries of the loops.
- We used the size calculated above and added OpenACC data directive on top of the outer loop to copy in vy, vh, vf, vg, and vs into the accelerator and copy them out of the accelerator.
- We flattened the vy, vh, vf, vg, and vs array to create 1D arrays for OpenACC.
- We added OpenACC kernels and loop directives on top of the two inner for loops.

OpenACC version:

```
// our OpenACC version
#pragma acc data pcopy(vy[0:nz*ng],vh[0:nz*ng],vf[0:nz*ng],vg[0:nz*ng],vs[0:nz*ng])
for ( l=1 ; l<=loop ; l++ ) {
    #pragma acc kernels
    #pragma acc loop independent
    for ( j=1 ; j<ng ; j++ ) {
        #pragma acc loop independent
        for ( k=1 ; k<nz ; k++ ) {
            if ( (j+1) >= ng ) {
                vy[j*nz+k] = 0.0;
            }else{
                double t = 0, s = 0, r =0;
                if ( vh[(j+1)*nz+k] > vh[j*nz+k] ) {
                    t = ar;
                }
                else {
                    t = br;
                }
                if ( vf[j*nz+k] < vf[j*nz+k-1] ) {
                    if ( vh[j*nz+k-1] > vh[(j+1)*nz+k-1] )
                        r = vh[j*nz+k-1];
                    else
                        r = vh[(j+1)*nz+k-1];
                    s = vf[j*nz+k-1];
                }
                else {
                    if ( vh[j*nz+k] > vh[(j+1)*nz+k] )
                        r = vh[j*nz+k];
                    else
                        r = vh[(j+1)*nz+k];
                    s = vf[j*nz+k];
                }
                vy[j*nz+k] = sqrt( vg[j*nz+k]*vg[j*nz+k] + r*r )* t/s;
            }
            if ( (k+1) >= nz ) {
                vs[j*nz+k] = 0.0;
            }else{
                double t = 0, s = 0, r =0;
                if ( vf[j*nz+k] < vf[(j-1)*nz+k] ) {
                    if ( vg[(j-1)*nz+k] > vg[(j-1)*nz+k+1] )
                        r = vg[(j-1)*nz+k];
                    else
                        r = vg[(j-1)*nz+k+1];
                    s = vf[(j-1)*nz+k];
                    t = br;
                }
```

```
                else {
                    if ( vg[j*nz+k] > vg[j*nz+k+1] )
                        r = vg[j*nz+k];
                    else
                        r = vg[j*nz+k+1];
                    s = vf[j*nz+k];
                    t = ar;
                }
                vs[j*nz+k] = sqrt( vh[j*nz+k]*vh[j*nz+k] + r*r )* t / s;
            }
        }
    }
}
```

Kernel 16: Monte Carlo search loop

Original code:

```
// original version in [2]
ii = n / 3;
lb = ii + ii;
k3 = k2 = 0;
for ( l=1 ; l<=loop ; l++ ) {
    i1 = m = 1;
    label410:
    j2 = ( n + n )*( m - 1 ) + 1;
    for ( k=1 ; k<=n ; k++ ) {
        k2++;
        j4 = j2 + k + k;
        j5 = zone[j4-1];
        if ( j5 < n ) {
            if ( j5+lb < n ) {              /* 420 */
                tmp = plan[j5-1] - t;       /* 435 */
            } else {
                if ( j5+ii < n ) {          /* 415 */
                    tmp = plan[j5-1] - s;   /* 430 */
                } else {
                    tmp = plan[j5-1] - r;   /* 425 */
                }
            }
        } else if( j5 == n ) {
            break;                          /* 475 */
        } else {
            k3++;                           /* 450 */
            tmp=(d[j5-1]-(d[j5-2]*(t-d[j5-3])*(t-d[j5-3])+(s-d[j5-4])*
                          (s-d[j5-4])+(r-d[j5-5])*(r-d[j5-5])));
        }
        if ( tmp < 0.0 ) {
            if ( zone[j4-2] < 0 )           /* 445 */
                continue;                   /* 470 */
            else if ( !zone[j4-2] )
                break;                      /* 480 */
        } else if ( tmp ) {
            if ( zone[j4-2] > 0 )           /* 440 */
                continue;                   /* 470 */
            else if ( !zone[j4-2] )
                break;                      /* 480 */
        } else break;                       /* 485 */
        m++;                                /* 455 */
        if ( m > zone[0] )
            m = 1;                          /* 460 */
        if ( i1-m )                         /* 465 */
            goto label410;
        else
            break;
    }
}
```

Investigating parallelism:

The most-outer loop iteratively calls the kernel. The most-inner loop cannot be loop-parallelized since it includes complex control flow (e.g. continue, break, and goto statements) making the number of loop iterations uncertain upon entering the loop. This loop cannot be parallelized in OpenACC since in OpenACC the number of loop iterations should be determined upon entering the loop. We did not port this kernel to OpenACC.

Kernel 17: Implicit, conditional computation

Original code:

```
// original version in [2]
for ( l=1 ; l<=loop ; l++ ) {
    i = n-1;
    j = 0;
    ink = -1;
    scale = 5.0 / 3.0;
    xnm = 1.0 / 3.0;
    e6 = 1.03 / 3.07;
    goto l61;
    l60:
    e6 = xnm*vsp[i] + vstp[i];
    vxne[i] = e6;
    xnm = e6;
    ve3[i] = e6;
    i += ink;
    if ( i==j ) goto l62;
    l61:
    e3 = xnm*vlr[i] + vlin[i];
    xnei = vxne[i];
    vxnd[i] = e6;
    xnc = scale*e3;
    if ( xnm > xnc ) goto l60;
    if ( xnei > xnc ) goto l60;
    ve3[i] = e3;
    e6 = e3 + e3 - xnm;
    vxne[i] = e3 + e3 - xnei;
    xnm = e6;
    i += ink;
    if ( i != j ) goto l61;
    l62:;
}
```

Investigating parallelism:

The most-outer loop iteratively calls the kernel. The most-inner loop cannot be loop-parallelized since it includes complex control flow (e.g. continue, break, and goto statements) making the number of loop iterations uncertain upon entering the loop. This loop cannot be parallelized in OpenACC since in OpenACC the number of loop iterations should be determined upon entering the loop. We did not port this kernel to OpenACC.

Kernel 18: 2-D explicit hydrodynamics fragment

Original code:

```
// original version in [2]
for ( l=1 ; l<=loop ; l++ ) {
    t = 0.0037;
    s = 0.0041;
    kn = 6;
    jn = n;
    for ( k=1 ; k<kn ; k++ ) {
      for ( j=1 ; j<jn ; j++ ) {
          za[k][j] = ( zp[k+1][j-1] +zq[k+1][j-1] -zp[k][j-1] -zq[k][j-1] )*
                     ( zr[k][j] +zr[k][j-1] ) / ( zm[k][j-1] +zm[k+1][j-1]);
          zb[k][j] = ( zp[k][j-1] +zq[k][j-1] -zp[k][j] -zq[k][j] ) *
                     ( zr[k][j] +zr[k-1][j] ) / ( zm[k][j] +zm[k][j-1]);
      }
    }
    for ( k=1 ; k<kn ; k++ ) {
        for ( j=1 ; j<jn ; j++ ) {
            zu[k][j] += s*( za[k][j]   *( zz[k][j] - zz[k][j+1] ) -
                            za[k][j-1] *( zz[k][j] - zz[k][j-1] ) -
                            zb[k][j]   *( zz[k][j] - zz[k-1][j] ) +
                            zb[k+1][j] *( zz[k][j] - zz[k+1][j] ) );
            zv[k][j] += s*( za[k][j]   *( zr[k][j] - zr[k][j+1] ) -
                            za[k][j-1] *( zr[k][j] - zr[k][j-1] ) -
                            zb[k][j]   *( zr[k][j] - zr[k-1][j] ) +
                            zb[k+1][j] *( zr[k][j] - zr[k+1][j] ) );
        }
    }
    for ( k=1 ; k<kn ; k++ ) {
        for ( j=1 ; j<jn ; j++ ) {
            zr[k][j] = zr[k][j] + t*zu[k][j];
            zz[k][j] = zz[k][j] + t*zv[k][j];
        }
    }
}
```

Investigating parallelism:

The most-outer loop iteratively calls the kernel. There are three two-nested loops inside the top loop. All these loops can be executed in parallel.

Modifications:

We made following modifications to parallelize this kernel:

- We calculated the size of arrays from boundaries of the loops.
- We used the size calculated above and added OpenACC data directive on top of the outer loop to copy in zp, zr, zq, zm, zz, za, zb, zu, and zv into the accelerator and copy them out of the accelerator.
- We flattened zp, zr, zq, zm, zz, za, zb, zu, and zv arrays to create 1D arrays for OpenACC.
- We added OpenACC kernels and loop directives on top of the all inner new loop and execute each of the three two-nested loops in a separate kernel..

OpenACC version:

```
// our OpenACC version
```

```
#pragma acc data
pcopy(zp[0:dim],zr[0:dim],zq[0:dim],zm[0:dim],zz[0:dim],za[0:dim],zb[0:dim],zu[0:dim],
zv[0:dim])
for ( l=1 ; l<=loop ; l++ ) {
    double t = 0.0037;
    double s = 0.0041;
    #pragma acc kernels
    #pragma acc loop independent
    for ( k=1 ; k<kn ; k++ ) {
        #pragma acc loop independent
        for ( j=1 ; j<jn ; j++ ) {
            za[k*jn+j] = ( zp[(k+1)*(1+jn)+j-1] +zq[(k+1)*(1+jn)+j-1]      -
zp[k*(1+jn)+j-1] -zq[(k  )*(1+jn)+j-1] )*
                         ( zr[(k  )*(1+jn)+j]   +zr[(k  )*(1+jn)+j-1] ) / (
zm[k*(1+jn)+j-1] +zm[(k+1)*(1+jn)+j-1]);
            zb[k*jn+j] = ( zp[(k  )*(1+jn)+j-1] +zq[(k  )*(1+jn)+j-1]      -
zp[k*(1+jn)+j]   -zq[(k  )*(1+jn)+j] ) *
                         ( zr[(k  )*(1+jn)+j]   +zr[(k-1)*(1+jn)+j] ) / (
zm[k*(1+jn)+j]   +zm[(k  )*(1+jn)+j-1]);
        }
    }
    #pragma acc kernels
    #pragma acc loop independent
    for ( k=1 ; k<kn ; k++ ) {
        #pragma acc loop independent
        for ( j=1 ; j<jn ; j++ ) {
            zu[k*jn+j] += s*( za[(k  )*(jn+1)+j]   *( zz[k*(1+jn)+j] - zz[(k
)*(1+jn)+j+1] ) -
                              za[(k  )*(jn+1)+j-1] *( zz[k*(1+jn)+j] - zz[(k
)*(1+jn)+j-1] ) -
                              zb[(k  )*(jn+1)+j]   *( zz[k*(1+jn)+j] - zz[(k-
1)*(1+jn)+j] ) +
                              zb[(k+1)*(jn+1)+j]   *( zz[k*(1+jn)+j] -
zz[(k+1)*(1+jn)+j] ) );
            zv[k*jn+j] += s*( za[(k  )*(jn+1)+j]   *( zr[k*(1+jn)+j] - zr[(k
)*(1+jn)+j+1] ) -
                              za[(k  )*(jn+1)+j-1] *( zr[k*(1+jn)+j] - zr[(k
)*(1+jn)+j-1] ) -
                              zb[(k  )*(jn+1)+j]   *( zr[k*(1+jn)+j] - zr[(k-
1)*(1+jn)+j] ) +
                              zb[(k+1)*(jn+1)+j]   *( zr[k*(1+jn)+j] -
zr[(k+1)*(1+jn)+j] ) );
        }
    }
    #pragma acc kernels
    #pragma acc loop independent
    for ( k=1 ; k<kn ; k++ ) {
        #pragma acc loop independent
        for ( j=1 ; j<jn ; j++ ) {
            zr[k*jn+j] = zr[k*jn+j] + t*zu[k*jn+j];
            zz[k*jn+j] = zz[k*jn+j] + t*zv[k*jn+j];
        }
    }
}
```

Kernel 19: General linear recurrence equations

Original code:

```
// original version in [2]
kb5i = 0;
for ( l=1 ; l<=loop ; l++ ) {
    for ( k=0 ; k<n ; k++ ) {
        b5[k+kb5i] = sa[k] + stb5*sb[k];
        stb5 = b5[k+kb5i] - stb5;
    }
    for ( i=1 ; i<=n ; i++ ) {
        k = n - i ;
        b5[k+kb5i] = sa[k] + stb5*sb[k];
        stb5 = b5[k+kb5i] - stb5;
    }
}
```

Investigating parallelism:

The most-outer loop iteratively calls the kernel. The two most-inner loops cannot be loop-parallelized since there is dependency among loop iterations over stb5 variable.

Kernel 20: Discrete ordinates transport, conditional recurrence on xx

Original code:

```
// original version in [2]
for ( l=1 ; l<=loop ; l++ ) {
    for ( k=0 ; k<n ; k++ ) {
        di = y[k] - g[k] / ( xx[k] + dk );
        dn = 0.2;
        if ( di ) {
            dn = z[k]/di ;
            if ( t < dn ) dn = t;
            if ( s > dn ) dn = s;
        }
        x[k] = ( ( w[k] + v[k]*dn )* xx[k] + u[k] ) / ( vx[k] + v[k]*dn );
        xx[k+1] = ( x[k] - xx[k] )* dn + xx[k];
    }
}
```

Investigating parallelism:

The most-outer loop iteratively calls the kernel. The most-inner loop cannot be loop-parallelized since there is dependency among loop iterations over xx[] array.

Kernel 21: Matrix*matrix product

Original code:

```
// original version in [2]
for ( l=1 ; l<=loop ; l++ ) {
    for ( k=0 ; k<25 ; k++ ) {
        for ( i=0 ; i<25 ; i++ ) {
            for ( j=0 ; j<n ; j++ ) {
                px[j][i] += vy[k][i] * cx[j][k];
            }
        }
    }
}
```

Investigating parallelism:

The most-outer loop iteratively calls the kernel. There are three nested loops inside this loop. All these loops can be executed fully in parallel, except the most-inner loop that includes a reduction.

Modifications:

We made following modifications to parallelize this kernel:

- We calculated the size of arrays from boundaries of the loops.
- We used the size calculated above and added OpenACC data directive on top of the outer loop to copy in vy, cx, and px into the accelerator and copy them out of the accelerator.
- We re-arrange the loops to avoid reduction in the most inner loop. In this case, we can run two outer loops in parallel. We run the most-inner loop serially.
- We flattened yv, cx, and px array to create 1D arrays for OpenACC.
- We added OpenACC kernels and loop directives on the two outer loops.

OpenACC version:

```
// our OpenACC version
#pragma acc data pcopy(vy[0:25*25],cx[0:n*25],px[0:n*25])
for ( l=1 ; l<=loop ; l++ ) {
    #pragma acc kernels
    #pragma acc loop independent
    for ( j=0 ; j<n ; j++ ) {
        #pragma acc loop independent
        for ( i=0 ; i<25 ; i++ ) {
            double sum = px[j*25+i];
            for ( k=0 ; k<25 ; k++ ) {
                sum += cx[j*25+k] * vy[k*25+i];
            }
            px[j*25+i] = sum;
        }
    }
}
```

Kernel 22: Planckian distribution

Original code:

```
// original version in [2]
expmax = 20.0;
u[n-1] = 0.99*expmax*v[n-1];
for ( l=1 ; l<=loop ; l++ ) {
    for ( k=0 ; k<n ; k++ ) {
        y[k] = u[k] / v[k];
        w[k] = x[k] / ( exp( y[k] ) -1.0 );
    }
}
```

Investigating parallelism:

The most-outer loop iteratively calls the kernel. The inner loop can be executed fully in parallel.

Modifications:

We made following modifications to parallelize this kernel:

- We calculated the size of arrays from boundaries of the loops.
- We used the size calculated above and added OpenACC data directive on top of the outer loop to copy in y, u, v, w, and x into the accelerator and copy them out of the accelerator.
- We added OpenACC kernels and loop directives on the two outer loops.

OpenACC version:

```
// our OpenACC version
#pragma acc data pcopy(y[0:n],u[0:n],v[0:n],w[0:n],x[0:n])
for ( l=1 ; l<=loop ; l++ ) {
    #pragma acc kernels
    #pragma acc loop independent
    for ( k=0 ; k<n ; k++ ) {
        y[k] = u[k] / v[k];
        w[k] = x[k] / ( exp( y[k] ) -1.0 );
    }
}
```

Kernel 23: 2-D implicit hydrodynamics fragment

Original code:

```
// original version in [2]
for ( l=1 ; l<=loop ; l++ ) {
    for ( j=1 ; j<6 ; j++ ) {
        for ( k=1 ; k<n ; k++ ) {
            qa = za[j+1][k]*zr[j][k] + za[j-1][k]*zb[j][k] +
                 za[j][k+1]*zu[j][k] + za[j][k-1]*zv[j][k] + zz[j][k];
            za[j][k] += 0.175*( qa - za[j][k] );
        }
    }
}
```

Investigating parallelism:

The most-outer loop iteratively calls the kernel. The most-inner loops cannot be loop-parallelized since there is dependency among loop iterations over za[] array. Loop iterated by j cannot be parallelized since za[j][k] depends on za[j-1][k]. Loop iterated by k cannot be parallelized since za[j][k] depends on za[j][k-1].

Kernel 24: Find location of first minimum in array

Original code:

```
// original version in [2]
x[n/2] = -1.0e+10;
for ( l=1 ; l<=loop ; l++ ) {
    m = 0;
    for ( k=1 ; k<n ; k++ ) {
        if ( x[k] < x[m] ) m = k;
    }
}
```

Investigating parallelism:

The most-outer loop iteratively calls the kernel. The inner loop cannot be executed in parallel since it tightly does two tasks: i) finding the minimum value and finding the associated array index. Breaking these two tasks into two loops allows parallelizing the loop with a reduction clause.

Modifications:

We made following modifications to parallelize this kernel:

- We calculated the size of arrays from boundaries of the loops.
- We used the size calculated above and added OpenACC data directive on top of the outer loop to copy in x into the accelerator.
- As explained above, we add another loop to perform two tasks serially. The first task is to find the minimum value and second task is to find the lowest index (first minimum) in the array having this minimum value. Both loops can be loop-parallelized by reduction clause.
- We added OpenACC kernels and loop directives on top of the both loop. Plus, we add reduction min clause to the loop directive to find minimum value and minimum index.

OpenACC version:

```
// our OpenACC version
x[n/2] = -1.0e+10;
#pragma acc data pcopyin(x[0:n])
for ( l=1 ; l<=loop ; l++ ) {
    m = 0;
    float val = 1024*1024;
    #pragma acc kernels
    #pragma acc loop independent reduction(min:val)
    for ( k=1 ; k<n ; k++ ) {
        val = x[k]<val? x[k] : val;
    }
    #pragma acc kernels
    #pragma acc loop independent reduction(min:m)
    for(k=1; k<n; k++){
        m = x[k]<=val? k : m;
    }
}
```