Evaluation of Hand-Coded Transformations of Explicitly-Parallel OpenMP-C Rodinia Benchmarks on Intel Xeon Phi

Prasanth Chatarasi (S01216877)
Course: Multi-Core Computing (COMP 522)
Course Instructor: Prof. John Mellor-Crummey

December 17, 2014

Contents

1 Introduction **3**

2 Intel Xeon Phi **3**

3 Rodinia Benchmarks **4**

4 Back Propagation **5**
4.1 Optimizations . . . 6
4.1.1 Loop permutation . . . 6
4.1.2 Loop collapsing . . . 8
4.2 Summary and Observations . . . 9

5 LU Decomposition **10**
5.1 Optimizations . . . 10
5.1.1 Loop permutation + Array reduction . . . 11
5.2 Summary and Observations . . . 12

6 Particle Filter **14**
6.1 Optimizations . . . 15
6.1.1 Loop Fusion . . . 15

7 Hotspot **18**
7.1 Optimizations . . . 19
7.1.1 Loop fusion + Do-across pipelined parallelism . . . 19
7.1.2 Loop fusion + Do-across pipelined parallelism + Vectorization . . . 20
7.2 Summary and Observations . . . 22

8 Conclusions **23**

9 Acknowledgments **24**

1 Introduction

In this work, we studied 18 explicitly-parallel OpenMP-C benchmarks from the Rodinia benchmark suite, and found that these benchmarks use six classes of non-affine constructs that are commonly found in parallel scientific applications: 1) Non-affine subscript expressions, 2) Indirect array subscripts, 3) Use of `structs`, 4) Calls to user-defined functions, 5) Non-affine loop bounds, and 6) Non-affine `if` conditions. Even though there are known techniques from past to handle some of these non-affine constructs, we would like to explore the use of `explicit parallelism` to enable a larger set of transformations for some of these programs, compared to what might have been possible if the input program was sequential. As part of this exploration, we begin with evaluation of hand-coded transformations of explicitly parallel OpenMP-C Rodinia benchmarks [1] on Intel Xeon Phi machine.

The rest of report is organized as follows. Section 2 presents an overview of `Intel Xeon Phi`, used in our experimental evaluation. Section 3 gives a summary of potential loop transformations for each explicitly parallel OpenMP-C benchmark in Rodinia suite. In the remainder of this report, we give a comprehensive experimental study of potential loop transformations for some of the benchmarks such as `Back propagation, LU decomposition, Particle filter, Hotspot`.

2 Intel Xeon Phi

In this section, we present a summary of Intel Xeon Phi machine, for the evaluation of benchmarks. Xeon Phi is a brand name used for all Intel's products based on their Many Integrated Core architecture (MIC). In our experimental setup, we have one Xeon Phi coprocessor that is associated with a host Xeon machine. The coprocessor[1] consists of 57 cores with each core consists of 4 SMT threads, a total of 228 SMT threads. Each core operates at a frequency of 1.1 GHz and total capacity of RAM is 5GB. Each core contains a vector processing unit (VPU) responsible for the execution of 512 bit wide SIMD vector instructions. Each core has a 32KB L1 data cache, a 32KB L1 instruction cache, and a 512KB L2 cache. The L2 caches of all cores are interconnected with each other and the memory controllers via a bidirectional ring bus, effectively creating a shared last-level cache of up to 32MB. The coprocessor can perform a maximum of 1.2 Teraflops of double precision floating point instructions. There are two modes of affinity such as `compact` and `scatter` on Intel Xeon Phi machine. If the affinity is set to `scatter`, then OpenMP runtime will assign threads such that they are spread among the cores first and then subsequent threads greater than the number of cores will then be assigned starting again at the first core. If the affinity mode is set to `compact`, then OpenMP runtime will distribute by setting all four threads to a single core before moving onto the next core.

[1] Most of these details are derived from /proc/cpuinfo, /proc/meminfo.

3 Rodinia Benchmarks

In this section, we provide an overview of each explicitly parallel OpenMP-C benchmark in Rodinia suite. This overview includes the non-affine constructs used in the benchmark and possible loop transformations. For all 18 benchmarks in the suite, Table 1 summarizes the following:

1. Constructs used in the benchmarks that limit the use of compiler frameworks, and leading to conservative analysis and transformations. These constructs include non-affine array subscripts (NAS), indirect array accesses (I), use of structs (S), and use of function calls (F).

2. Potential opportunities for loop transformations. For instance, the hotspot benchmark has non-affine array subscripts (NAS), but can benefit from loop fusion, skewing, tiling, and doacross transformations. The table also shows that non-affine subscripts and function calls are common in this benchmark suite, while indirect array accesses and structs are found in a few benchmarks.

In the remainder of this report, we give a comprehensive experimental study of potential loop transformations for some of the benchmarks such as `Back propagation`, `LU decomposition`, `Particle filter`, `Hotspot`.

Limitations

Kernel	NAS	I	S	F	Transformations
b+ tree		✓	✓		
backprop				✓	**perm, fuse, collape, vect**
bfs		✓	✓		
cfd	✓			✓	
heartwall				✓	
hotspot	✓				**doacross, fuse, skew, tile, vect**
kmean				✓	perm, fuse, vect
lavaMD	✓	✓	✓		
leukocyte				✓	fuse, vect
lud	✓				**perm**
mummergpu			✓	✓	
myocyte	✓			✓	
nn	✓			✓	
nw	✓			✓	doacross, skew, perm
particle filter	✓			✓	**fuse**
path finder					doacross
srad	✓				
streamcluster	✓	✓	✓	✓	
Total	10	4	5	11	

Table 1: Limitations and possible transformations in Rodinia benchmarks (NAS: non-affine array subscript, I: indirect array access, S: structure, F: function, and perm/fuse/skew/tile/collapse/doacross/vect: loop permutation/fusion/skewing/tiling/collapsing/doacross parallelism/vectorization)

4 Back Propagation

Back Propagation [1] is a machine-learning algorithm that trains the weights of connecting nodes on a layered neural network. The application is comprised of two phases: `forward phase`, in which the activations are propagated from the input to the output layer, and `backward Phase`, in which the error between the observed and requested values in the output layer is propagated backwards to adjust the weights and bias values. In this work, we consider a kernel associated with adjusting weights of nodes on a layered neural network and the corresponding code is shown in Figure 1.

```
1 // Input to program: nly
2 // Value of ndelta: 16, a constant value in the kernel
3 #pragma omp parallel for  shared(oldw, w, delta) private(j, k, new_dw)  firstprivate(ndelta,↩
      nly)
4 for (j = 1; j <= ndelta; j++) {
5   for (k = 0; k <= nly; k++) {
6     new_dw = ((ETA * delta[j] * ly[k]) + (MOMENTUM * oldw[k][j])); //S1
7     w[k][j] += new_dw; //S2
8     oldw[k][j] = new_dw; //S3
9   }
10 }
```

Figure 1: A part of back propagation kernel, in which `nly` is an input parameter and `ndelta` is a constant

In the kernel, statements `S1, S2 and S3` are enclosed in a doubly nested loop, in which the outer most loop is annotated as parallel using OpenMP-C `parallel for` construct. Figure 2 and Figure 3 shows performance of the kernel on Intel Xeon Phi machine, described in Section 2, with `affinity` mode set to `compact` and `scatter`, respectively. The gigaflops value was calculated by dividing the total number of floating point operations by the time taken to execute the kernel.

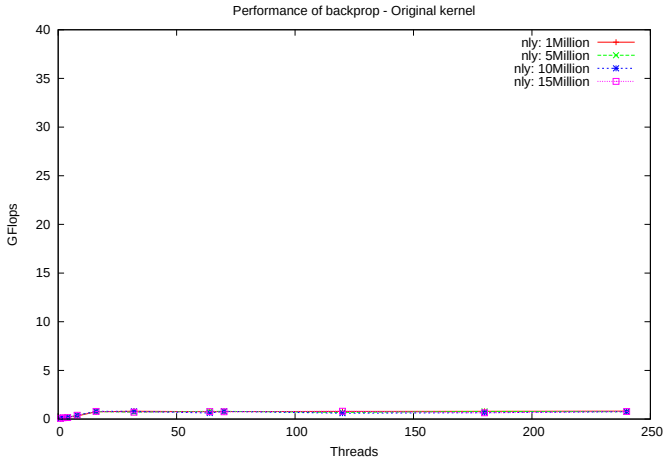

Figure 2: Performance of kernel in Figure 1 with `affinity` mode as `compact`

†Performance on largest input with one thread: 0.06 Gflops

As can be seen from figures, the scalability of the kernel has been saturated after the thread count increased to 16. This is because of maximum parallelism available at outer most loop is `ndelta` (value: 16, constant in kernel). In the remainder of this section, we discuss on possible

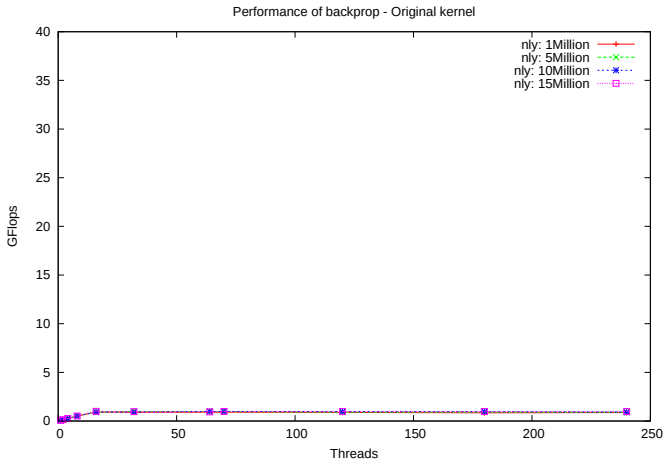

Figure 3: Performance of kernel in Figure 1 with `affinity` mode as `scatter`

†Performance on largest input with one thread: 0.06 Gflops

transformations to improve the scalability and performance of the kernel.

4.1 Optimizations

The major bottle necks in Figure 1 are fixed amount of parallelism at the outer most loop level, poor spatial locality and poor vectorization. In this section, we discuss two major optimizations – 1) Loop permutation for better spatial locality and vectorization 2) Loop collapsing to increase parallelism.

4.1.1 Loop permutation

In the access `oldw[k][j]`, each iteration of loop `k` results in accessing an element distant from `nly` elements from current location. Hence, it exhibits poor spatial locality and can be improved by performing loop permutation. The legality of loop permutation is easily established by the fact that there are no cross-iteration dependences in the kernel. After the loop permutation, vectorization can be performed on inner loop `j` since the accesses are contiguous and there are no cross-iteration dependences. Figure 4 shows the transformed code after applying loop permutation to kernel in Figure 1.

```
1 // Input to program: nly
2 // Value of ndelta: constant in program
3 #pragma omp parallel for  shared(oldw, ndelta, w, delta) private(j, k, new_dw, nly)
4 for (k = 0; k <= nly; k++) {
5   #pragma ivdep always
6   for (j = 1; j <= ndelta; j++) {
7     new_dw = ((ETA * delta[j] * ly[k]) + (MOMENTUM * oldw[k][j])); //S1
8     w[k][j] += new_dw; //S2
9     oldw[k][j] = new_dw; //S3
10  }
11 }
```

Figure 4: Transformed part of back propagation kernel after Loop permutation

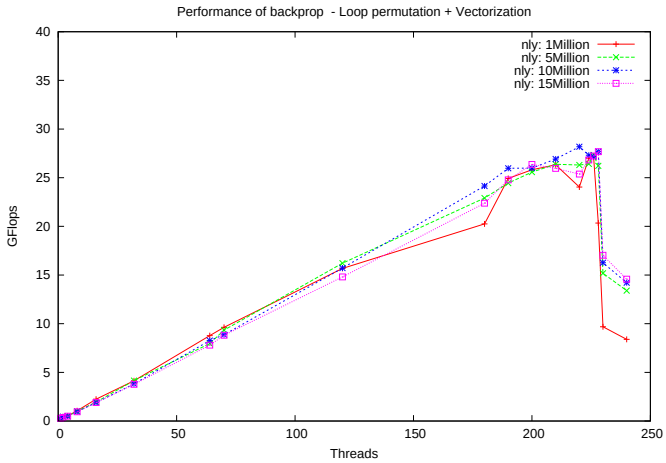

Figure 5: Performance of kernel in Figure 4 with `affinity` mode as `compact`

†Performance on largest input with one thread: 0.264 Gflops

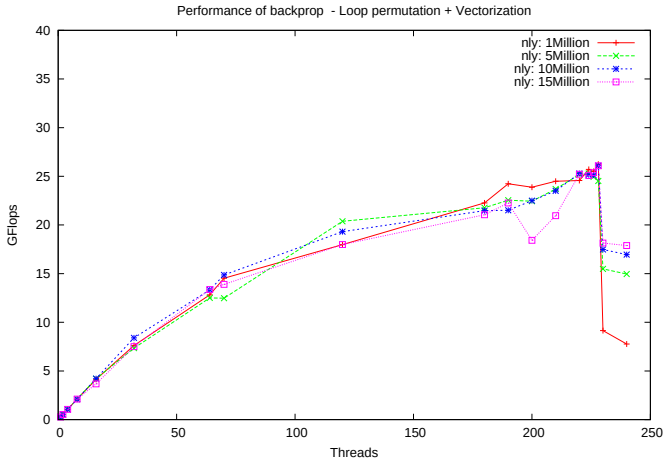

Figure 6: Performance of kernel in Figure 4 with `affinity` mode as `scatter`

†Performance on largest input with one thread: 0.264 Gflops

Figure 5 and Figure 6 shows the performance of permuted kernel in Figure 4 on Intel Xeon machine, described in Section 2, with `affinity` mode set to `compact` and `scatter`, respectively. As can be seen from figures, the transformed code is scalable till the maximum number of threads in the machine and the performance is improved from about 1Gflops to 28 Gflops (28X improvement) due to this transformation.

4.1.2 Loop collapsing

After applying loop permutation, loops can be further collapsed to increase the parallelism as there are no cross-iteration dependences in Figure 4. Figure 7 shows the kernel after applying loop collapsing through OpenMP directive `collapse`.

```
1 // Input to program: nly
2 // Value of ndelta: constant in program
3 #pragma omp parallel for collapse(2) shared(oldw, ndelta, w, delta) private(j, k, new_dw, ↩
      nly)
4 for (k = 0; k <= nly; k++) {
5   #pragma ivdep always
6   for (j = 1; j <= ndelta; j++) {
7     new_dw = ((ETA * delta[j] * ly[k]) + (MOMENTUM * oldw[k][j])); //S1
8     w[k][j] += new_dw; //S2
9     oldw[k][j] = new_dw; //S3
10  }
11 }
```

Figure 7: Transformed part of back propagation kernel after Loop collapsing

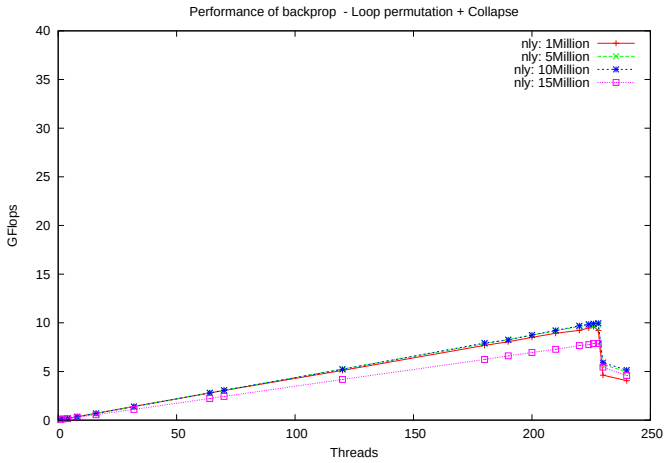

Figure 8: Performance of kernel in Figure 7 with `affinity` mode as `compact`

†Performance on largest input with one thread: 0.080 Gflops

Figure 8 and Figure 9 shows the performance of transformed code in Figure 7 on Intel Xeon machine, described in Section 2, with `affinity` mode set to `compact` and `scatter`, respectively. As can be seen from figures, the transformed code is scalable till the maximum number of available threads and the performance is improved from about 1Gflops to 10 Gflops (10X improvement) with respect to original kernel in Figure 1. But, the performance of kernel with only loop permutation is larger (2.8X improvement) with respect to kernel with loop collapsing. As the single loop after collapsing is utilized for the purpose of only parallelism, there hasn't been scope for vectorization.

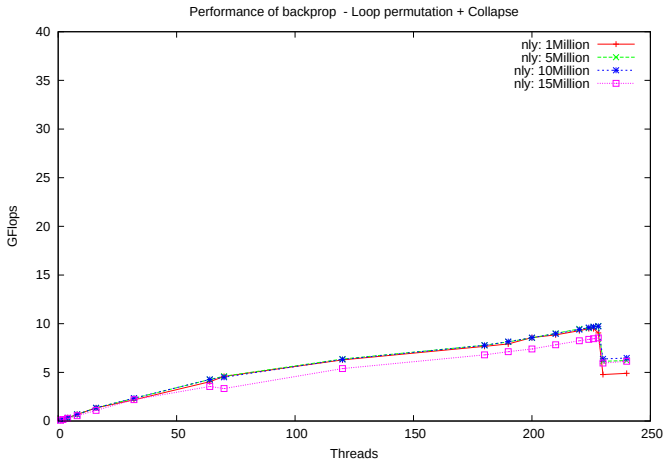

Figure 9: Performance of kernel in Figure 7 with `affinity` mode as `scatter`

†Performance on largest input with one thread: 0.080 Gflops

4.2 Summary and Observations

Table 2 shows the performance improvement of back propagation kernel with respect to original kernel.

Loop permutation	Loop collapsing
28X	10X

Table 2: Performance improvements with respect to original kernel in Figure 1

As part of addressing for better spatial locality, vectorization and parallelism for kernel in Figure 1, we have observed the following key points.

- The kernel needs to be optimized by maintaining a right balance between parallelism and vectorization, as Intel Xeon Phi machine is designed for the combination of both objectives.
- There is a significant reduction ($>$ 2X) in performance after setting OpenMP threads more than the available number of threads in the machine.
- The performance in the affinity mode with `compact` mostly gives better performance (1.1X) than affinity mode with `scatter`.

5 LU Decomposition

LU Decomposition [1] is an algorithm to calculate the solutions of a set of linear equations. The LUD kernel decomposes a matrix as the product of a lower triangular matrix and an upper triangular matrix. Figure 10 shows the part of LUD kernel.

```
1 for (i=0; i <size; i++){
2   #pragma omp parallel for default(none)  private(j,k,sum) shared(size,i,a)
3   for (j=i; j <size; j++){
4     for (k=0; k<i; k++){
5       a[i*size+j] -= a[i*size+k]*a[k*size+j]; //S
6     }
7   }
8   ......
9 }
```

Figure 10: A part of LUD kernel, in which `size` is an input parameter

In the kernel, statement `S` is enclosed in a triply nested loop with loop `'j'` annotated as parallel using OpenMP `parallel for`. The kernel is compiled using `Intel ICC` with `-O3 -vec` flags. Figure 11 and Figure 12 shows the performance of the kernel on Intel Xeon Phi machine, with `affinity` mode set to `compact` and `scatter` respectively. As can be seen from figures, the part of original kernel is scalable till the maximum number of threads in the machine. Still, this kernel can be optimized as there is poor spatial locality with access `a[k*size+j]` in the statement `S`.

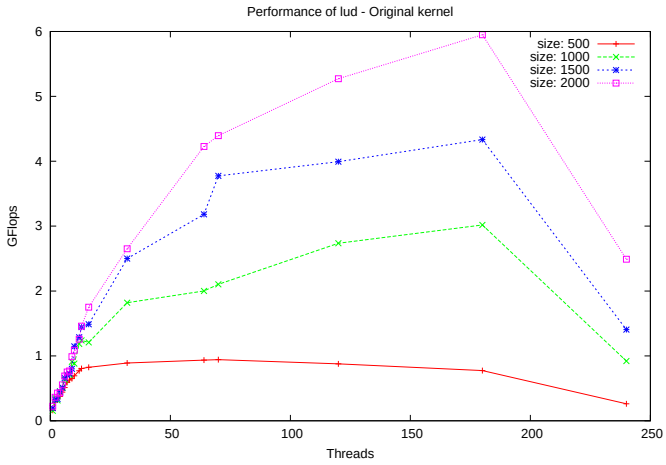

Figure 11: Performance of kernel in Figure 10 with `affinity` mode as `compact`

†Performance on largest input with one thread: 0.224 Gflops

5.1 Optimizations

Even though the kernel has scalability, it can be optimized by improving spatial locality. So, we consider loop permutation optimization thereby to improve spatial locality.

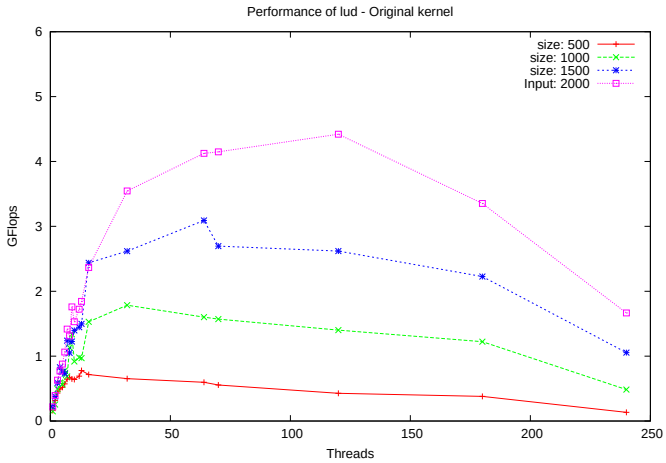

Figure 12: Performance of kernel in Figure 10 with `affinity` mode as `scatter`

†Performance on largest input with one thread: 0.218 Gflops

5.1.1 Loop permutation + Array reduction

The statement `S` in the kernel is enclosed by a triply nested loop having iterators `i, j and k` in the order of loop depth. The statement `S` contains a memory access *a[k*size+j]*, in which each iteration of the loop *k* results in accessing an element with a distance of *size* elements from current location. As a result, it exhibits poor spatial locality. The legality of loop permutation on loops `j, k` is easily established by the fact that there are no cross-iteration dependences on those loops in the kernel. After permutation, array reduction has to be performed since the array element `i*size+j` is accessed in each iteration of parallel `k-loop` . Figure 13 shows the kernel after performing loop permutation on the kernel. As OpenMP-C 4.0 doesn't have support for array reductions, we have

```
1 for (i=0; i <size; i++) {
2 #pragma omp parallel for reduction(+:a)   private(j)
3   for (k=0; k<i; k++) {
4     for (j=i; j <size; j++) {
5       a[i*size+j]-=a[i*size+k]*a[k*size+j];  //S1
6     }
7   }
8     ....
9 }
```

Figure 13: Transformed part of LUD Kernel after Loop permutation

provided the implementation. Figure 14 shows the manual implementation of array reduction after permutation of `j, k loops`. Figure 15 and Figure 16 shows the performance of kernel in Figure 14 on Intel Xeon Phi machine, with `affinity` mode to `compact` and `scatter` respectively. As seen from figures, the transformed is scalable and maximum performance improved from 5.8Gflops to 6.6 Gflops (1.15X) improvement due to this transformation.

```
    // Initialization
  float **_r_a;
   _r_a = (float **) calloc(size, sizeof(float *));
   for(id=0; id < size;id++)
       _r_a[id] = (float *) calloc(size, sizeof(float));

// Kernel
for (i=0; i <size; i++){
  int _lw1 = i*size + i;
  int _up1 = i*size + (size-1);
  int _size1 = _up1 - _lw1 + 1;
  #pragma omp parallel
  {
    int _p1;
    int _id = omp_get_thread_num();
    int _total = omp_get_num_threads();

    #pragma omp for private(k)
    for (k = 0; k < i; k ++) {
      #pragma ivdep
      for (j = i; j < size; j ++) {
        _r_a[_id][j-i] -= a[i*size+k]*a[k*size+j];
      }
    }
    #pragma omp barrier

    // Parallel Aggregation of results
    #pragma omp for private(_id)
    for (_p1 = 0; _p1 < _size1; _p1++)
    {
      for (_id = 0; _id < _total; _id++)
      {
            a[_lw1+_p1] += _r_a[_id][_p1];
            _r_a[_id][_p1] = 0;
        }
    }

  }
  ......
}
```

Figure 14: Permuted LUD kernel - Parallel aggregation in array reduction

5.2 Summary and Observations

We have performed variants of array reduction i.e. sequential and parallel aggregation of results with dynamic memory allocation of temporary array near to the loop on which OpenMP`reduction` is mentioned. Table 3 shows the performance improvement of back propagation kernel with respect to original kernel. As part of addressing for better spatial locality for kernel in Figure 10, we have

Loop permutation + Sequential Aggregation + Local initialization of array	Loop permutation + Parallel Aggregation + Local initialization of array	Loop permutation + Parallel Aggregation + Global initialization of reduction array
-1.74X (Loss)	-1.82X (Loss)	1.15X (Gain)

Table 3: Performance improvements with respect to original kernel in Figure 10

observed the following key points.

- While performing array reduction, the memory allocation and initialization for reduction array (temporary) needs to be performed before the loops. There was a significant performance

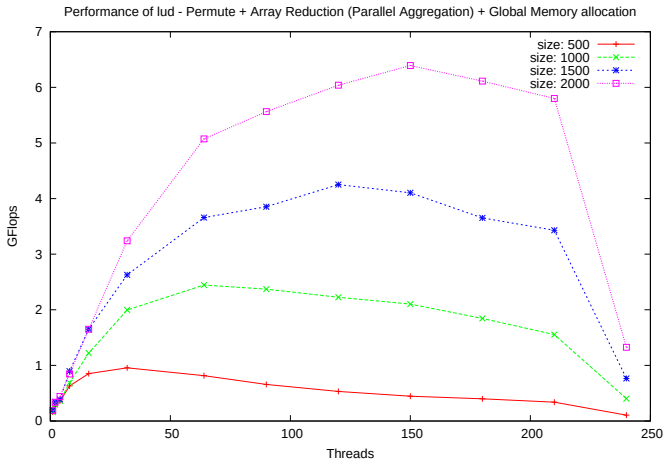

Figure 15: Performance of kernel in Figure 14 with `affinity` mode as `compact`

†Performance on largest input with one thread: 0.189 Gflops

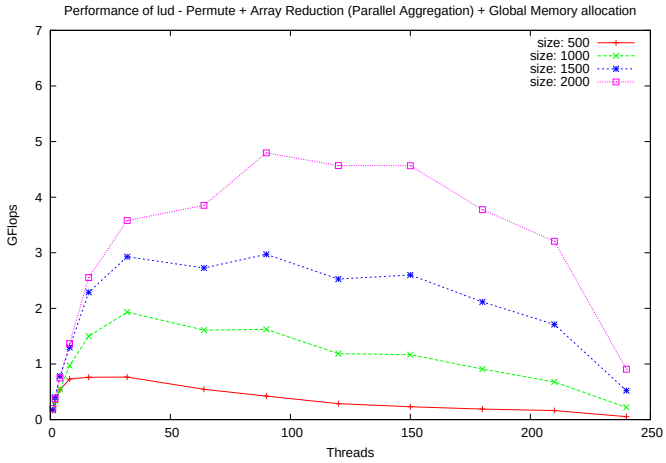

Figure 16: Performance of kernel in Figure 14 with `affinity` mode as `scatter`

†Performance on largest input with one thread: 0.201 Gflops

reduction (1.8X) in keeping dynamic memory allocation for reduction array inside `i-loop`.

- The performance in the affinity mode with `compact` mostly gives better performance (1.35X) than affinity mode with `scatter`.

6 Particle Filter

The particle filter (PF) [1] is a statistical estimator of the location of a target object given noisy measurements of that target's location and an idea of the object's path in a Bayesian framework. The PF has a plethora of applications ranging from video surveillance in the form of tracking vehicles, cells and faces to video compression. This particular implementation is optimized for tracking cells, particularly leukocytes and myocardial cells. Figure 17 shows the part of particle filter kernel.

```
#pragma omp parallel for
for(x = 0; x < Nparticles; x++){
  arrayX[x] += 1 + 5*randn(seed, x);
  arrayY[x] += -2 + 2*randn(seed, x);
}

#pragma omp parallel for private(y, indX, indY)
for(x = 0; x < Nparticles; x++){
  for(y = 0; y < countOnes; y++){
    indX = roundDouble(arrayX[x]) + objxy[y*2+1];
    indY = roundDouble(arrayY[x]) + objxy[y*2];
    ind[x*countOnes+y] = fabs(indX ... indY ...);
    ...
    likelihood[x] += ...I[ind[x*countOnes+y]]...
  }
  ...
}

#pragma omp parallel for
for(x = 0; x < Nparticles; x++){
  weights[x] = weights[x] * exp(likelihood[x]);

#pragma omp parallel for private(x)   reduction(+:sumWeights)
for(x = 0; x < Nparticles; x++)
  sumWeights += weights[x];
}
```

Figure 17: Part of particle filter kernel

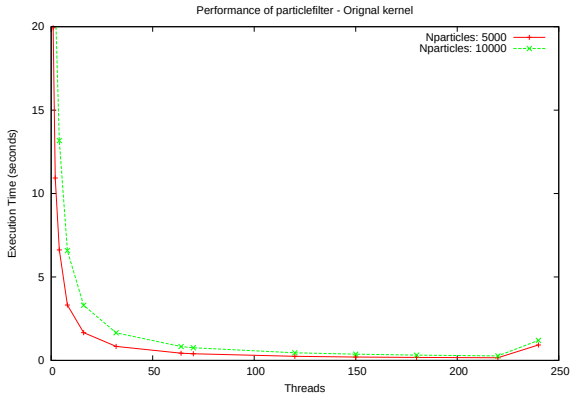

Figure 18: Performance of kernel in Figure 17 with `affinity` mode as `compact`

†Performance on largest input with one thread: 39.8 Seconds

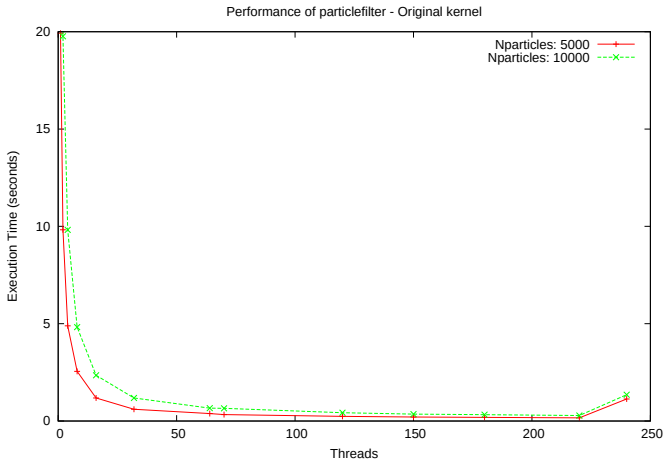

Figure 19: Performance of kernel in Figure 17 with `affinity` mode as `scatter`

†Performance on largest input with one thread: 39.9 Seconds

Figure 18 and Figure 19 shows the performance of Figure 17 on Intel Xeon Phi machine, with `affinity` mode set to `compact` and `scatter` respectively. We couldn't compute Gflops manually as the kernel involves user-defined function calls and math library calls. Hence, performance figures are plotted with `execution time` on `y-axis`. As can be seen from figures that, there is a steady decline in execution time as the thread count increases, resulting as scalable kernel.

6.1 Optimizations

The sequence of loop nests in kernel is a good candidate for loop fusion as there are no dependences across loop nests. As part of optimizations, we perform loop fusion to enhance spatial locality.

6.1.1 Loop Fusion

```
#pragma omp parallel for private(x, y, \
  indX, indY) reduction(+:sumWeights)
for(x = 0; x < Nparticles; x++){
  arrayX[x] += 1 + 5*randn(seed, x);
  arrayY[x] += -2 + 2*randn(seed, x);
  for(y = 0; y < countOnes; y++){
    indX = roundDouble(arrayX[x]) + objxy[y*2+1];
    indY = roundDouble(arrayY[x]) + objxy[y*2];
    ind[x*countOnes+y] = fabs(indX ... indY ...);
    ...
    likelihood[x] += ...I[ind[x*countOnes+y]]...
  }
  ...
  weights[x] = weights[x] * exp(likelihood[x]);
  sumWeights += weights[x];
}
```

Figure 20: Part of particle filter kernel after Loop fusion

The sequence of loops in Figure 17 is an excellent candidate for loop fusion since they have the

same iteration domain and share multiple arrays with the same access patterns, e.g., `arrayX[x], arrayY[x], likelihood[x]`. The legality of loop fusion is easily established by the fact that all variables that cross multiple loops have affine accesses with no fusion-preventing dependences. The key information needed from the parallel program is that the second loop (lines 7-17 in Figure 20) has no loop carried dependence. This ensures that the fused loop can also be made parallel. Figure 20 shows the code after fusion of loop nests. Figure 21 and Figure 22 shows the performance of Figure 20 on Intel Xeon Phi machine, with `affinity` mode set to `compact` and `scatter` respectively. As seen from performance figures, there is slight improvement (1.05X) on fused ker-

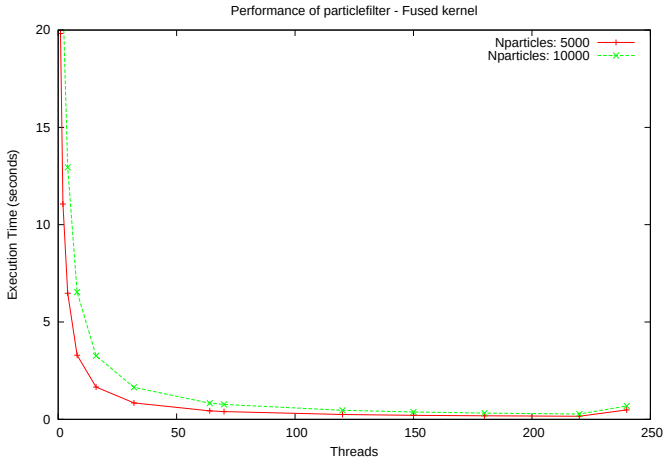

Figure 21: Performance of kernel in Figure 20 with `affinity` mode as `compact`

†Performance on largest input with one thread: 40.25 Seconds

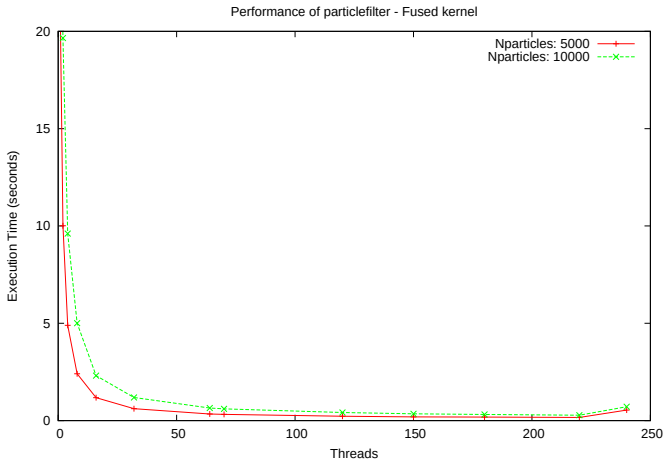

Figure 22: Performance of kernel in Figure 20 with `affinity` mode as `scatter`

†Performance on largest input with one thread: 39.86 Seconds

nel with respect to original kernel on largest input. In case of auto-tuning applications for spatial locality with loop fusion, vectorization could add more benefit towards performance.

7 Hotspot

The Hotspot (HS) [1] used to estimate processor temperature based on an architectural floorplan and simulated power measurements. The kernel is a 5-point 2-dimensional stencil computation where 2-D arrays are linearized to 1-D and hence contain non-affine subscripts. Figure 23 shows code corresponding to part of hotspot kernel. The kernel contains two imperfect loop nests sharing

```
for (int i = 0; i < num_iterations ; i++) {
#pragma omp parallel for
  for (r = 0; r < row; r++) {
    for (c = 0; c < col; c++) {
      if (...) {
        ... // handling corner cases
      } else {
        delta = (step / Cap) * (power[r*col+c] +
        (temp[(r+1)*col+c] + temp[(r-1)*col+c] -
        2.0*temp[r*col+c])/Ry + (temp[r*col+c+1]
        + temp[r*col+c-1] - 2.0*temp[r*col+c])
        / Rx + (amb_temp - temp[r*col+c]) / Rz);
        result[r*col+c] =temp[r*col+c]+ delta;
      }
    }
  }

  #pragma omp parallel for
  for(r = 0; r < row; r++) {
    for (c=0; c < col; c++) {
      temp[r*col+c] = result[r*col + c];
      }
    }
}
```

Figure 23: A part of hotspot kernel

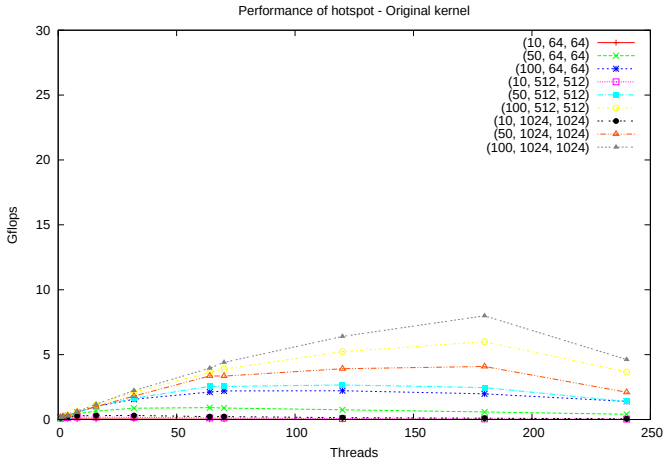

Figure 24: Performance of kernel in Figure 23 with affinity mode as `compact`, in which triplet values represent num_iterations, row and col, respectively.

†Performance on largest input with one thread: 0.144 Gflops

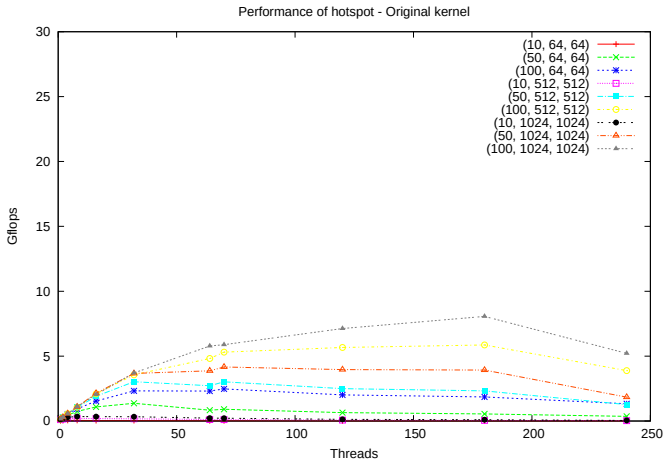

Figure 25: Performance of kernel in Figure 23 with affinity mode as `scatter`, in which triplet values represent num_iterations, row and col, respectively.

†Performance on largest input with one thread: 0.144 Gflops

a common outermost loop. There are no cross-iteration dependences inside both loop nests. But, there are flow and anti-flow dependences across both loop nests. This kernel is compiled using Intel `icc` compiler with `-O3 -vec` flags. Figure 24 and Figure 25 show the performance of the kernel on Intel Xeon machine with affinity mode as `compact` and `scatter` respectively. As can be seen from the performance figures, there is a scalability till the maximum number of threads available in the machine.

7.1 Optimizations

The inner loops of the kernel are exploited for parallelism by specifying though OpenMP `parallel for` construct. Even though the input kernel is scalable, we exploit parallelism at outer loop in the remainder of section.

7.1.1 Loop fusion + Do-across pipelined parallelism

In the input kernel, there are anti-flow dependences between the loop nests in the same outer most loop iteration and there are flow dependences between the loop nests in the consecutive outer most loop iterations. These dependences can be specified using do-across construct [2]. As this construct is applicable for only perfect loop nests, the loop nests in the input kernel needs to be fused. We perform loop fusion after applying index-shifting to preserve semantics of kernel after fusion. Figure 26 shows the kernel after applying loop fusion and do-across parallelism on the input kernel.

Figure 28 and Figure 27 shows the performance of kernel in Figure 26 on Intel Xeon Phi machine with affinity mode as `compact` and `scatter` respectively. As can be seen from performance figures, there is a slight reduction (1.25X) in performance with respect to original kernel. While exploiting outer loop parallelism through `do-across`, there is a significant amount of execution time spent in waiting for dependent iterations to complete. Less granularity of work per thread and no vector utilization are some more reasons for reduction in performance.

```
  #pragma omp parallel for ordered(3)
for (int i = 0; i < num_iterations ; i++) {
  for (r = 0; r < row; r++) {
    for (c = 0; c < col; c++) {
      #pragma omp ordered depend(sink: t-1, r+2, c+1)
      if (...) {
        ... // handling corner cases
      } else {
        delta = (step / Cap) * (power[r*col+c] +
        (temp[(r+1)*col+c] + temp[(r-1)*col+c] -
        2.0*temp[r*col+c])/Ry + (temp[r*col+c+1]
        + temp[r*col+c-1] - 2.0*temp[r*col+c])
        / Rx + (amb_temp - temp[r*col+c]) / Rz);
        result[r*col+c] =temp[r*col+c]+ delta;
      }

    if( r >=1 && c >= 1)
      temp[(r-1)*col+c-1] = result[(r-1)*col + c-1];
      }
    #pragma omp ordered depend(source: t, r, c)
  }
 }
```

Figure 26: Part of hotspot kernel after fusing `r, c` loops and applying do-across parallelism

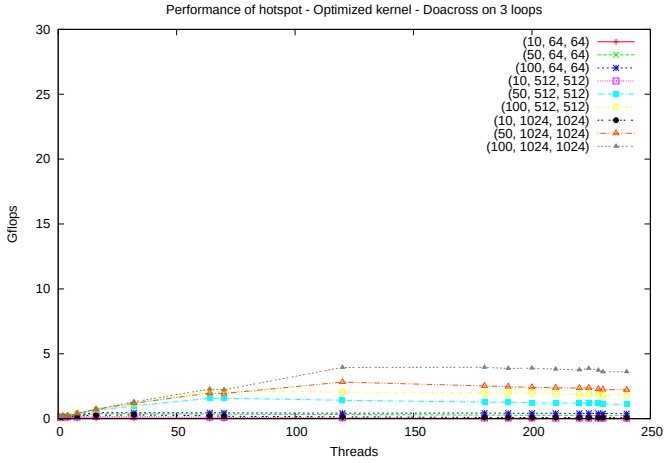

Figure 27: Performance of kernel in Figure 26 with thread affinity as `compact`, in which triplet values represent num_iterations, row and col, respectively.

†Performance on largest input with one thread: 0.117 Gflops

7.1.2 Loop fusion + Do-across pipelined parallelism + Vectorization

In kernel (Figure 26), do-across construct is specified on all loops for exploiting parallelism after fusing all loops. As we observed in earlier benchmarks, vectorization gives a better scope for performance in case of fusion. So, we fused only `r-loop` and retained `c-loop` for vectorization. As a result, we have two loop nests sharing `t-loop and r-loop` as their outer loops and vectorization on inner most loop. Figure 29 shows the kernel after applying index shifting of `r-loop`, fusing `r-loop` and do-across on shared loops i.e. `t-loop` and `r-loop`. In kernel Figure 29, the inner most loop i.e. `c-loop` doesn't have loop carried dependences and can be vectorized. This kernel is compiled using Intel `icc` compiler using `-O3 -vec` flags. Figure 30 and Figure 31 show the

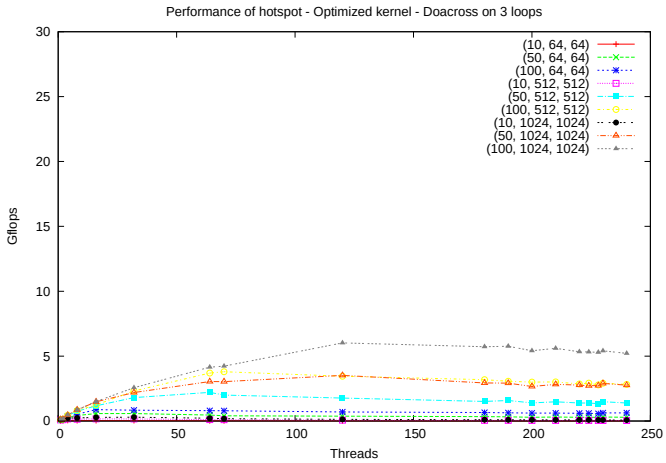

Figure 28: Performance of kernel in Figure 26 with thread affinity as `scatter`, in which triplet values represent num_iterations, row and col, respectively.

†Performance on largest input with one thread: 0.117 Gflops

```
  #pragma omp parallel for ordered(2)
for (int i = 0; i < num_iterations ; i++) {
  for (r = 0; r < row; r++) {
    #pragma omp ordered depend(sink: t-1, r+2)
    #pragma simd vectorlength(512)
    for (c = 0; c < col; c++) {
      if (...) {
        ... // handling corner cases
      } else {
        delta = (step / Cap) * (power[r*col+c] +
        (temp[(r+1)*col+c] + temp[(r-1)*col+c] -
        2.0*temp[r*col+c])/Ry + (temp[r*col+c+1]
        + temp[r*col+c-1] - 2.0*temp[r*col+c])
        / Rx + (amb_temp - temp[r*col+c]) / Rz);
        result[r*col+c] =temp[r*col+c]+ delta;
      }
    }

    #pragma simd vectorlength(512)
    for( c = 0; c < col; c++) {
      temp[r*col+c] = result[r*col+c];
    }
    #pragma omp ordered depend(source: t, r)
  }
}
```

Figure 29: Part of hotspot kernel after fusing `r-loop` and do-across parallelism on `t, c-loops`

performance of kernel on Intel Xeon Phi machine with affinity mode as `compact` and `scatter` respectively. As can be seen from performance figures, there is performance improvement (2.25X) with respect to original kernel.

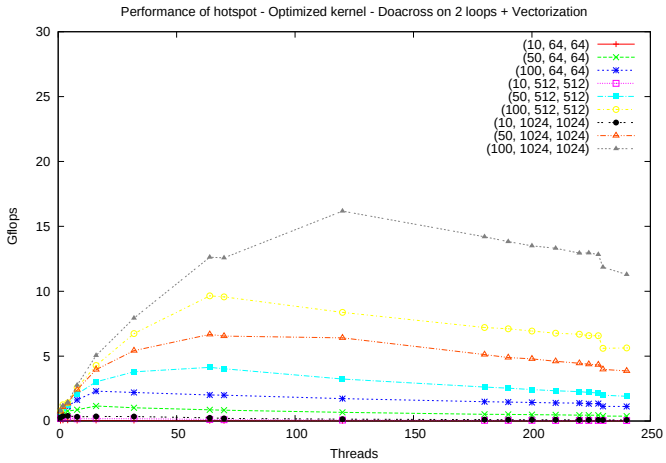

Figure 30: Performance of kernel in Figure 29 with affinity as `compact`, in which triplet values represent num_iterations, row and col, respectively.

†Performance on largest input with one thread: 0.117 Gflops

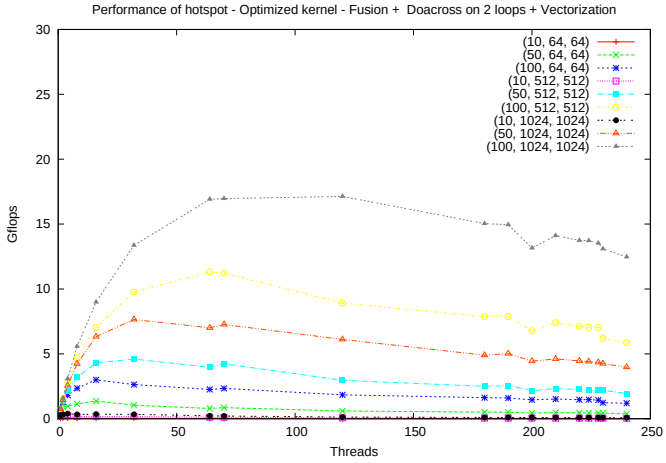

Figure 31: Performance of kernel in Figure 29 with affinity as `scatter`, in which triplet values represent num_iterations, row and col, respectively.

†Performance on largest input with one thread: 0.117 Gflops

7.2 Summary and Observations

Table 4 shows the performance improvement of hotspot kernel with respect to original kernel.

As part of addressing for better spatial locality, vectorization and parallelism for kernel in Figure 23, we have observed the following key points.

- While exploiting pipelined parallelism using `do-across` construct, there should be sufficient granularity of work per thread and can be achieved through loop skewing and tiling.[2]

[2]As part of future work, we would like to conduct study of hotspot after applying do-across pipelined parallelism,

Index Shifting + Loop fusion (r, c-loop) + Do-across parallelism (t, r, c-loops)	Index Shifting + Loop fusion (r-loop) + Do-across parallelism (t, r, loops) + Vectorization
-1.25X (Loss)	2.25X

Table 4: Performance improvements with respect to original kernel in Figure 23

- The performance in the affinity mode with `compact` mostly gives better performance (1.15X) than affinity mode with `scatter`.

8 Conclusions

In this work, we aimed at exploring the use of of `explicit parallelism` to enable a larger set of transformations for some of the explicitly parallel OpenMP-C Rodinia benchmarks [1] on Intel Xeon Phi machine, compared to what might have been possible if the input program was sequential. The remainder of this section presents the overall findings.

- The Rodinia suite has a right set of benchmarks with constructs ranging from affine to non-affine, access patterns such as linearized accesses , computations such as stencil computations, parallel constructs ranging from OpenMP `parallel for` to `do-across`. These diverse benchmarks helped us in applying various transformations such as `fusion, permutation, tiling, skewing, index shifting, do-across pipelined parallelism and vectorizations`.
- As Intel Xeon Phi coprocessor is designed with an objective of exploiting parallelism and vectorization, there needs to be a right balance between them while tuning for high performance.
- While considering array reductions into OpenMP-C, memory allocation and initialization for temporary array has to be given utmost importance as this can reduce the entire performance improvement.
- Even though loop fusion increases spatial locality, vectorization can add significantly benefit in performance. So, vectorization should be rigorously exploited while performing loop fusion transformation.
- While exploiting pipelined parallelism using do-across construct, there needs to be sufficient granularity of work per thread to compensate inter-thread waiting time. This can be achieved through a sequence of skewing and tiling.
- While targeting for spatial locality, the affinity mode with `compact` mostly gives better performance than affinity mode with `scatter`.

Finally, we leverage the implicit happens-before relations in explicitly parallel construct to enable larger set of transformations and we could see the overall performance improvement ranging from 1.05X to 28X. Hence, we would like to automate the entire process of leveraging explicit parallelism for transformations as part of future work.

loop skewing and tiling

9 Acknowledgments

I heartily thank Prof. John Mellor-Crummey, Prof. Vivek Sarkar, Jun Shirako, Karthik S. Murthy, Milind Chabbi, Shams Imam and Deepak for the valuable feedback and insightful suggestions.

References

[1] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54, Oct 2009.

[2] J. Shirako, P. Unnikrishnan, S. Chatterjee, K. Li, and V. Sarkar. Expressing DOACROSS Loop Dependencies in OpenMP. In *9th International Workshop on OpenMP (IWOMP)*, 2011.