Computación Paralela. Laboratorio II. Tiny_MD

González Federico(i); Mérida Julián(j)

(i) Universidad Nacional de Rosario; (j) Universidad Nacional de Córdoba

Introducción

En este laboratorio buscamos alternativas para conseguir utilizar el conjunto de instrucciones vectoriales Avx2 para mejorar la performance de nuestro problema `Tiny Molecular Dinamics`. En un primer intento tratamos de ayudar al compilador a vectorizar automáticamente cambiando la estructura de datos del problema de AoS a Soa, simplificando funciones inline y pasando distintas flags para identificar donde el compilador no podía vectorizar, principalmente analizamos la función forces.

Estas ideas no funcionaron así que proseguimos a implementar el problema central del proyecto, la ejecución de la función `forces`, en ISPC. Una vez hecho esto conseguimos vectorizar consiguiendo una mejora aproximada del 56% con respecto a los mejores resultados del lab 1.

Por último, mientras estábamos obteniendo las métricas, descubrimos que al utilizar el compilador Clang de Intel que viene dentro del conjunto de herramientas oneApi(*Intel(R) oneAPI DPC++ Compiler 2021.2.0*) el problema se resolvía por si solo ya que este compilador si podía vectorizar forces automáticamente, tanto para las versiones del código AoS como SoA. Es decir con Clang de Intel utilizando el programa original (sin ninguna modificación) conseguimos la misma performance que usando ISPC. En cuanto al uso de ISPC en este problema, la ventaja que pudimos observar es que podemos lograr la mejora del 56% independientemente del compilador utilizado.

Optimizaciones

Análisis código auto vectorizable con Clang

Nuestro primer intento consistió en correr clang con las siguientes flags para intentar encontrar donde estaban los problemas al vectorizar:

- -Rpass=loop-vectorize
- -Rpass-missed=loop-vectorize
- -Rpass-analysis=loop-vectorize

Y obtuvimos los siguientes datos:

```
clang -std=c11 -Wall -Wextra -g  -O3 -march=native -DN=500 -Rpass=loop-vectorize -Rpass-missed=loop-vectorize -Rpass-analysis=loop-vectorize -c core.c
core.c:20:13: remark: the cost-model indicates that vectorization is not beneficial [-Rpass-missed=loop-vectorize]
            for (int k = 0; k < nucells; k++) {
            ^
core.c:20:13: remark: the cost-model indicates that interleaving is not beneficial [-Rpass-missed=loop-vectorize]
core.c:51:23: remark: loop not vectorized: call instruction cannot be vectorized [-Rpass-analysis=loop-vectorize]
        vxyz[i + 0] = rand() / (double)RAND_MAX - 0.5;
                      ^
core.c:50:5: remark: loop not vectorized: instruction cannot be vectorized [-Rpass-analysis=loop-vectorize]
    for (int i = 0; i < 3 * N; i += 3) {
    ^
core.c:50:5: remark: loop not vectorized [-Rpass-missed=loop-vectorize]
core.c:69:5: remark: the cost-model indicates that vectorization is not beneficial [-Rpass-missed=loop-vectorize]
    for (int i = 0; i < 3 * N; i += 3) { // elimina la velocidad del centro de masa
    ^
core.c:69:5: remark: the cost-model indicates that interleaving is not beneficial [-Rpass-missed=loop-vectorize]
core.c:111:9: remark: loop not vectorized: unsafe dependent memory operations in loop. Use #pragma loop distribute(enable) to allow loop distribution to attempt to 
s into a separate loop [-Rpass-analysis=loop-vectorize]
        for (int j = i + 3; j < 3 * N; j += 3) {
        ^
core.c:111:9: remark: loop not vectorized [-Rpass-missed=loop-vectorize]
core.c:184:5: remark: the cost-model indicates that vectorization is not beneficial [-Rpass-missed=loop-vectorize]
    for (int i = 0; i < 3 * N; i += 3) { // actualizo posiciones
    ^
core.c:184:5: remark: the cost-model indicates that interleaving is not beneficial [-Rpass-missed=loop-vectorize]
core.c:206:15: remark: loop not vectorized: cannot prove it is safe to reorder floating-point operations; allow reordering by specifying '#pragma clang loop vectori
by providing the compiler option '-ffast-math'. [-Rpass-analysis=loop-vectorize]
        sumv2 += vxyz[i + 0] * vxyz[i + 0] + vxyz[i + 1] * vxyz[i + 1]
                 ^
core.c:201:5: remark: loop not vectorized [-Rpass-missed=loop-vectorize]
    for (int i = 0; i < 3 * N; i += 3) { // actualizo velocidades
    ^
```

En estas líneas, sabiendo que la función forces va desde las líneas 91 hasta 164 de `core.c`, vemos que el compilador no está vectorizando ningún ciclo.

Ayudas al compilador

Intentamos pasarle el parámetro `#pragma loop distribute(enable)` pero el mensaje siguió siendo el mismo. Luego probamos modificando la estructura original de `Array of Structures` a `Structures of Array`. Así, ahora la función forces recibe 6 arreglos:

- 3 arreglos rx, ry, y rz para las posiciones de las partículas, en vez de un arreglo rxyz.
- 3 arreglos fx, fy, y fz para las fuerzas de las partículas, en vez un fxyz.

Pero el resultado fue el mismo, Clang informa que no pudo vectorizar ningún loop. Esto mismo fue verificado con compiladores intel y gcc dando resultados también negativos.

Por último, antes de implementar `forces` en `ISPC`, probamos transformar `minimum_image` en una función inline dentro de forces pensando que quizás esta era una causa por la que el compilador tenía problemas para vectorizar, pero una vez más, no hubo éxito.

Implementación de forces en ISPC

Si bien no pudimos lograr que autovectorice con la versión SoA, esta modificación ayudó para realizar la implementación en ISPC. En base a la versión SoA realizamos:

- Primero implementamos `minimum_image` en `ISPC` ya que ésta se llama en `forces` y queremos que también se vectorice. Su implementación es igual al código original (únicamente tuvimos que quitar `static` en el return de la función). Como dentro de la función solo se hace una comparación, es totalmente paralelizable.

- Y para la implementación de `forces` a continuación vemos su implementación en ISPC:

```
export void forces(const double uniform rx[], const double uniform ry[],
                   const double uniform rz[], double uniform fx[],
                   double uniform fy[], double uniform fz[],
                   uniform double uniform epot[],
```

```
                    uniform double uniform pres[],
                    uniform const double uniform temp[],
                    uniform const double rho, uniform const double V,
                    uniform const double L)
{

foreach(i = 0 ... N)
{
    fx[i] = 0.0d;
    fy[i] = 0.0d;
    fz[i] = 0.0d;
}

uniform double pres_vir;
double pres_vir_partial = 0.0d ;
uniform double rcut2 = RCUT * RCUT;
double epot_partial = 0.0d;


foreach(i = 0 ... N-1)
{
    double xi = rx[i];
    double yi = ry[i];
    double zi = rz[i];

    for (int j = i + 1; j <  N; j++)
    {
        double xj = rx[j];
        double yj = ry[j];
        double zj = rz[j];

        double rxd = xi - xj;
        rxd = minimum_image(rxd, L);
        double ryd = yi - yj;
        ryd = minimum_image(ryd, L);
        double rzd = zi - zj;
        rzd = minimum_image(rzd, L);

        double rij2 = rxd * rxd + ryd * ryd + rzd * rzd;
        if (rij2 <= rcut2) {
            double r2inv = 1.0d / rij2;
            double r6inv = r2inv * r2inv * r2inv;

            double fr = 24.0d * r2inv * r6inv * (2.0d * r6inv - 1.0d);

            fx[i] += fr * rxd;
            fy[i] += fr * ryd;
            fz[i] += fr * rzd;

            fx[j] -= fr * rxd;
            fy[j] -= fr * ryd;
            fz[j] -= fr * rzd;

            epot_partial +=  4.0d * r6inv * (r6inv - 1.0d) - ECUT;
```

```
                pres_vir_partial += fr * rij2 ;
            }
        }
    }

    *epot=reduce_add(epot_partial);
    pres_vir=reduce_add(pres_vir_partial);
    pres_vir /= (V * 3.0d);
    *pres = *temp * rho + pres_vir;
    }
```

El código implementado es muy similar al original salvo que es necesario hacer el `reduce` de todas las sumas parciales que se calculan en cada una de las `programCount` instancias y deben declararse como `uniform` las constantes y variables de entrada que toma la función.

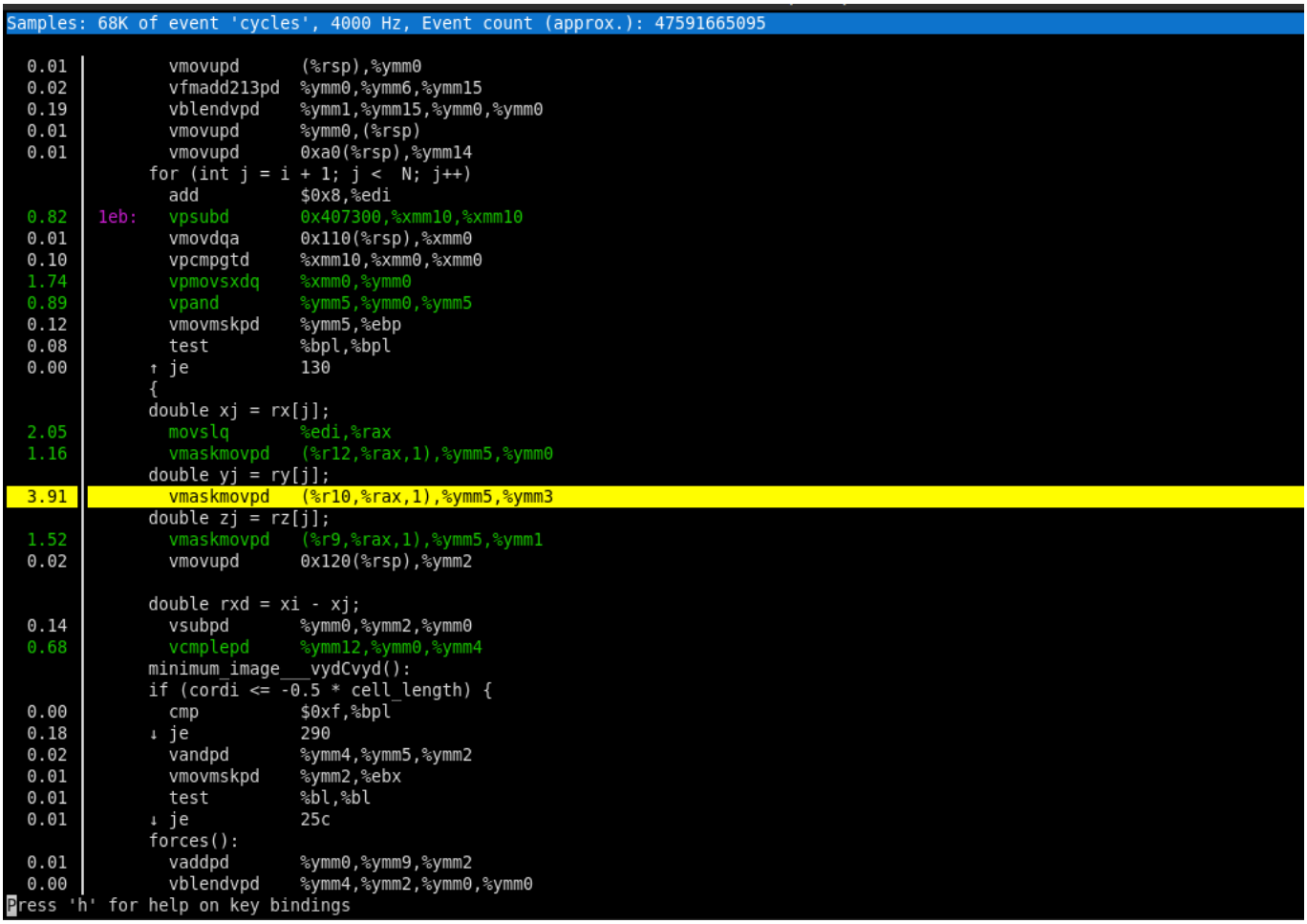
Intento de mejora implementación ISPC

Una mejora que quisimos implementar fue reemplazar el if del segundo ciclo por un cif, ya que pensábamos que este camino se toma más veces y por lo tanto el programa daría un mejor rendimiento.

Al probarlo, utilizando `perf record` y `perf report` pudimos ver que la instrucción más cargada en el caso del if es `vfmadd231pd`, la cual constituye el 3.90% del total de ciclos de ejecución de la función `forces`.

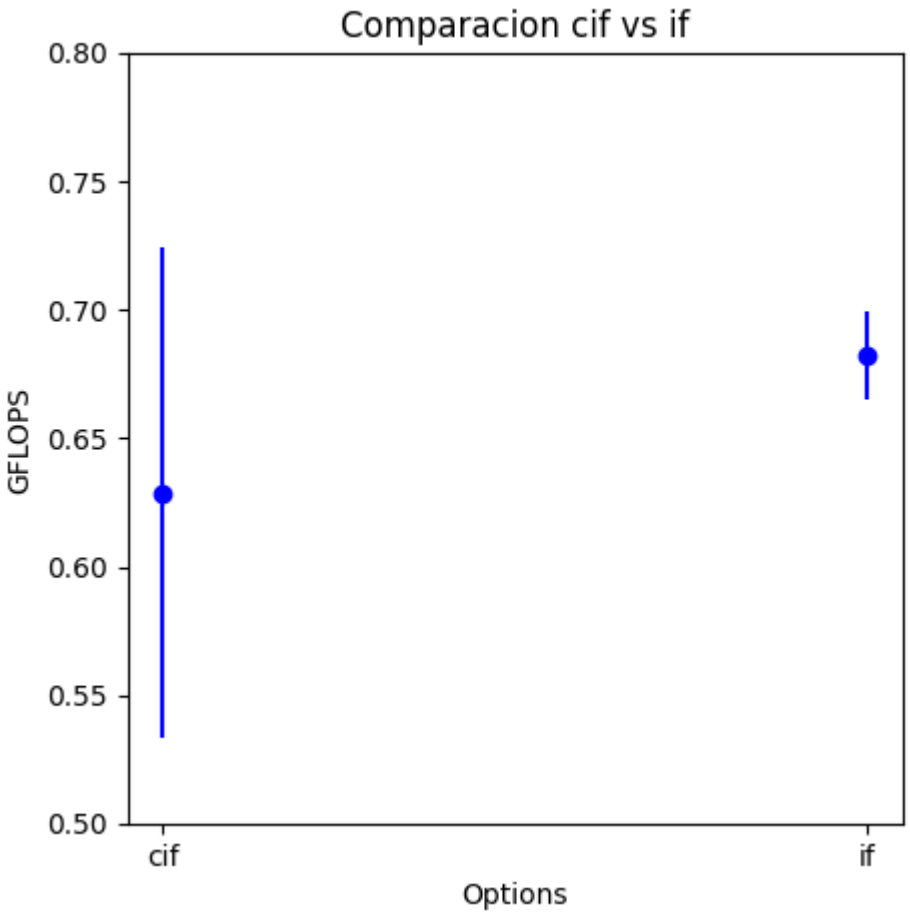
```
Samples: 69K of event 'cycles', 4000 Hz, Event count (approx.): 48324772113

  0.04 |       vmovmskpd    %ymm2,%ebp
       |       test         %bpl,%bpl
  0.00 |     ↓ je           3e8
       |     forces():
  0.01 |       vaddpd       %ymm15,%ymm9,%ymm2
  0.00 |       vblendvpd    %ymm1,%ymm2,%ymm15,%ymm15
       |     minimum_image___vydCvyd():
       |     cordi += cell_length;
  0.02 | 3e8:  vandpd       %ymm0,%ymm6,%ymm1
  0.01 |       vmovmskpd    %ymm1,%ebp
       |     forces():
  0.01 |       test         %bpl,%bpl
  0.00 |     ↓ je           40a
  0.15 | 3f5:  vcmpltpd     %ymm15,%ymm13,%ymm1
  0.03 |       vsubpd       %ymm9,%ymm15,%ymm2
  0.17 |       vandpd       %ymm0,%ymm1,%ymm0
  1.01 |       vblendvpd    %ymm0,%ymm2,%ymm15,%ymm15
       |     double rij2 = rxd * rxd + ryd * ryd + rzd * rzd;
  0.06 | 40a:  vmulpd       %ymm11,%ymm11,%ymm4
  1.29 |       vfmadd231pd  %ymm5,%ymm5,%ymm4
  3.90 |       vfmadd231pd  %ymm15,%ymm15,%ymm4
       |     if (rij2 <= rcut2) {
  0.07 |       vbroadcastsd 0x134e(%rip),%ymm0        # 4052c0 <_IO_stdin_used+0x2c0>
  1.64 |       vcmplepd     %ymm0,%ymm4,%ymm0
  0.21 |       vandpd       %ymm6,%ymm0,%ymm12
  1.67 |       vmovmskpd    %ymm12,%ebp
  0.04 |       test         %bpl,%bpl
  0.09 |     ↑ je           1ba
  0.70 |       vmovapd      %ymm8,%ymm13
       |     double r2inv = 1.0d / rij2;
  0.52 |       vbroadcastsd 0x10b1(%rip),%ymm0        # 405048 <_IO_stdin_used+0x48>
  2.22 |       vdivpd       %ymm4,%ymm0,%ymm0
       |     double r6inv = r2inv * r2inv * r2inv;
  0.63 |       vmulpd       %ymm0,%ymm0,%ymm1
  0.64 |       vmulpd       %ymm1,%ymm0,%ymm1
       |     double fr = 24.0d * r2inv * r6inv * (2.0d * r6inv - 1.0d);
  0.00 |       vbroadcastsd 0x131c(%rip),%ymm2        # 4052c8 <_IO_stdin_used+0x2c8>
       |       vmulpd       %ymm2,%ymm0,%ymm0
  1.04 |       vmulpd       %ymm1,%ymm0,%ymm0
Press 'h' for help on key bindings
```

Y para el caso del cif, reemplazando la línea `if (rij2 <= rcut2)` por `cif (rij2 <= rcut2)` encontramos que la instrucción con mas ejecución es `vmaskmovpd`, con un porcentaje de tiempo de ejecución casi idéntico de 3.91%.



Aunque las dos instrucciones tengan el mismo porcentaje, los resultados muestran que el código que implementa el `coherent if` es mucho más inestable.

Al realizar 10 simulaciones para cada caso, obtuvimos la siguiente gráfica:



Por lo tanto, concluimos que utilizar el cif no era conveniente para esta implementación, ya que en la mayoría de casos su desempeño fue peor al del if.

Descubrimiento Intel DPC++ oneApi Clang

Cuando estábamos realizando simulaciones para obtener las métricas finales de este informe, notamos que luego de setear las variables en `setvars.sh` para usar `icc` la llamada a clang se sobrescribía con el compilador `DPC++ Intel oneApi Clang`.

Al probarlo vimos que este compilador autovectorizaba tanto las versiones AoS y SoA, sin utilizar `ISPC` y los resultados obtenidos eran iguales a los de la implementación con `forces.ispc`. Por lo tanto, usando este compilador no habría necesidad de re-implementar código.

Plataforma de cálculos

Los simulaciones se corrieron sobre Jupiterace, la cual tiene las siguientes prestaciones

CPU:

- Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.4
- 28 cores, 56 threads con smt habilitado,
- Processor frequency : 2.4 - 3.3 GHz
- Caches:
    - L1 data: 896 KiB
    - L1 instr.: 896 KiB

    - L2: 7 MiB
    - L3: 70 MiB

Memoria:

- Memoria RAM: 128 GB

Compiladores

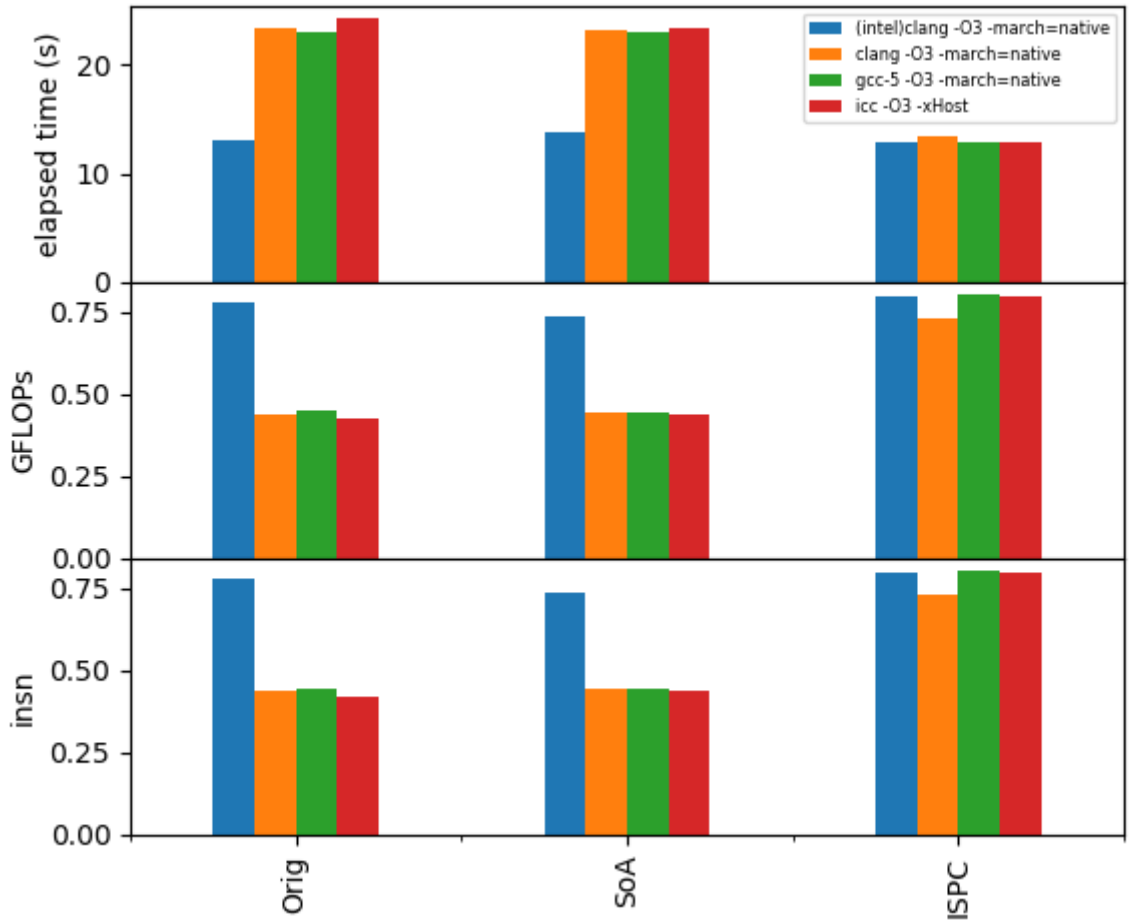
Para poder obtener el mejor resultado de la nueva implementación del programa, usamos las siguientes versiones de compiladores:

- Intel(R) oneAPI DPC++ Compiler 2021.2.0 (Intel oneApi Clang)}
- Debian Clang version 11.0.1-2}
- GCC versión 5.4.1-4}
- ICC 2021.2.0 (gcc version 10.2.1 compatibility)}

Resultados

En la siguiente figura se muestran los resultados para las diferentes versiones de código y diferentes compiladores con un tamaño de simulación N=500 para todos los casos.

- `Original` es la versión original sin ninguna modificación.
- `SoA` es la versión con estructura de arreglos.
- `ISPC` es la versión con `forces` y `minimum_image` implementadas en ISPC.



Métricas obtenidas

Original

CFLAG	Tiempo	GFLOPS	Insn
(intel)clang -O3 -march=native	13.13	0.78	1.59
clang -O3 -march=native	23.45	0.44	1.02
gcc-5 -O3 -march=native	23.00	0.45	1.05
icc -O3 -xHost	24.30	0.42	1.07

SoA

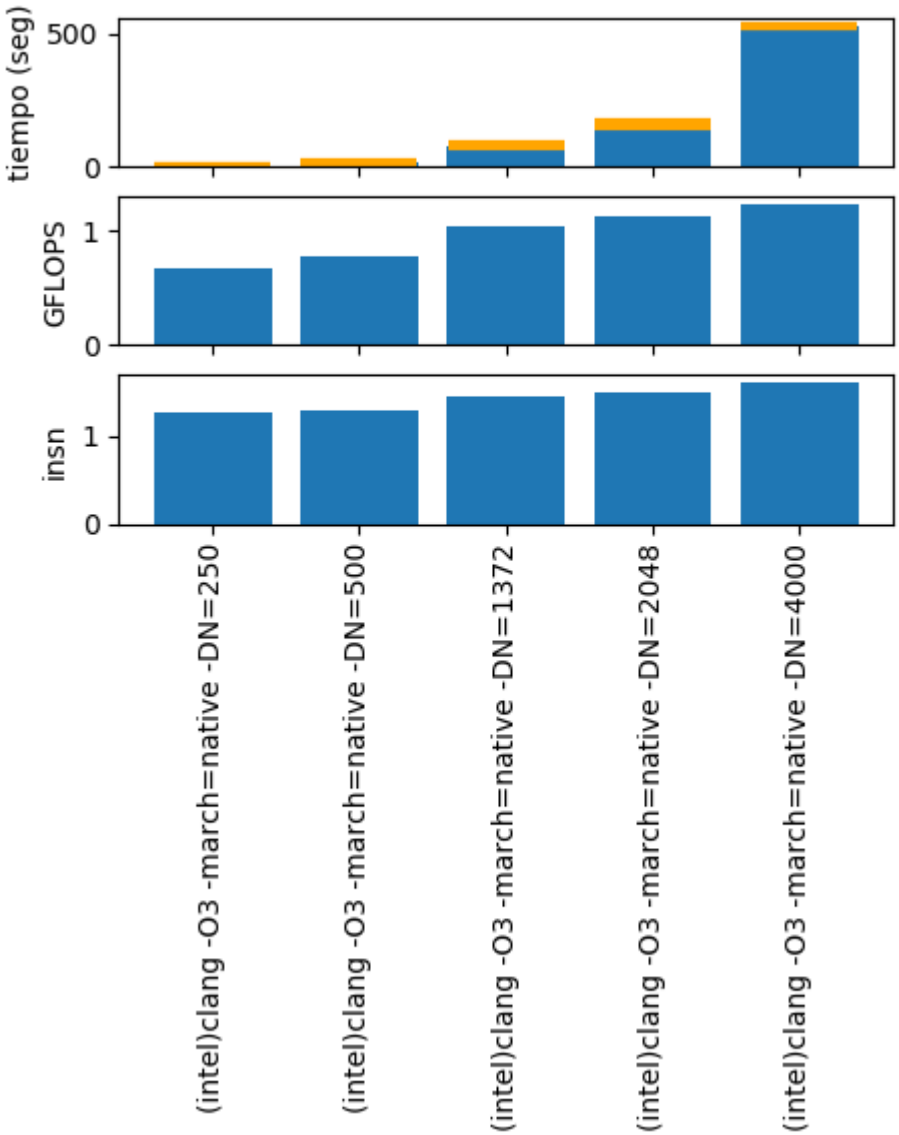
CFLAG	Tiempo	GFLOPS	Insn
(intel)clang -O3 -march=native	13.83	0.74	1.71
clang -O3 -march=native	23.19	0.44	1.03
gcc-5 -O3 -march=native	23.12	0.44	1.04
icc -O3 -xHost	23.36	0.44	1.02

ISPC

CFLAG	Time	GFLOPS	Insn
(intel)clang -O3 -march=native	12.86	0.80	1.28
clang -O3 -march=native	13.41	0.73	1.23
gcc-5 -O3 -march=native	12.88	0.80	1.28
icc -O3 -xHost	12.80	0.79	1.28

Se observa el aumento del 56% para los compiladores clang, gcc y icc al utilizar la versión ISPC, en cambio para el compilador clang de intel los resultados son iguales para todas las versiones.

En la siguiente imagen se muestra la escalabilidad del problema, es decir el tiempo, los GFLOPS y el insn para diferentes tamaños de muestra N.



Métricas

CFLAG	Time	errtime	GFLOPS	Insn

CFLAG	Time	errtime	GFLOPS	Insn
(intel)clang -O3 -march=native -DN=250	4.0557	0.08	0.67	1.28
(intel)clang -O3 -march=native -DN=512	14.051	0.90	0.78	1.30
(intel)clang -O3 -march=native -DN=1372	81.19	6.51	1.03	1.45
(intel)clang -O3 -march=native -DN=2048	159.45	5.52	1.12	1.51
(intel)clang -O3 -march=native -DN=4000	532.1610	0.02	1.23	1.62

Comparativa contra la mejor versión del laboratorio 1

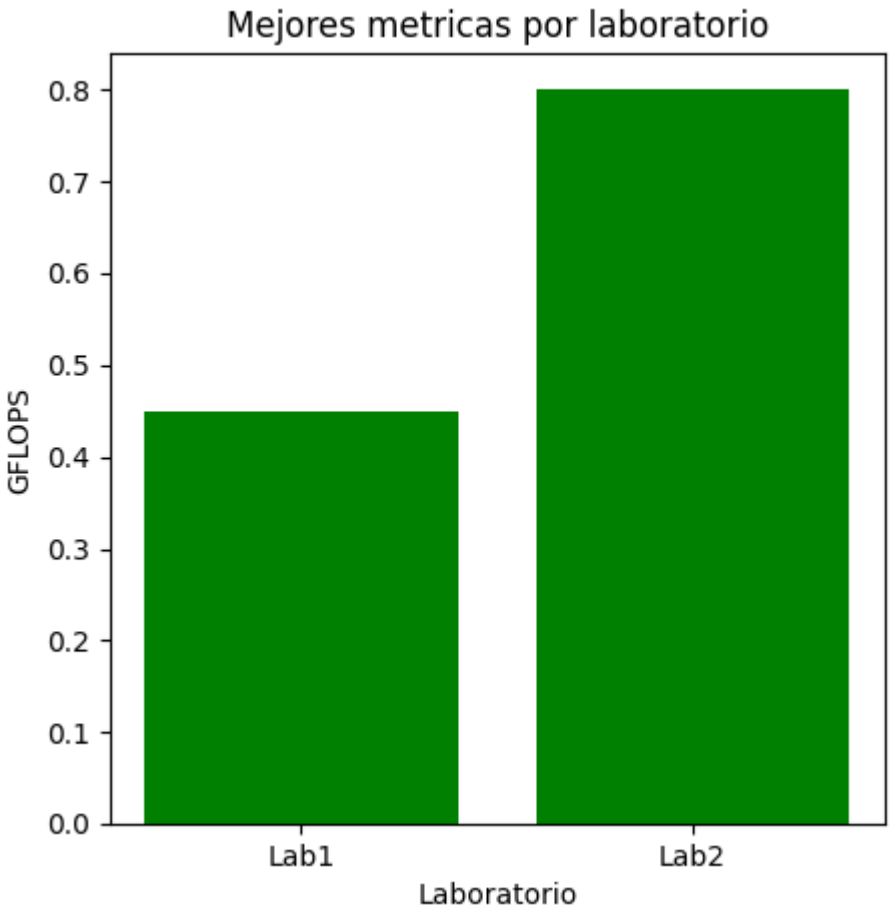
Laboratorio 1:

- Usando GCC
- 0.45 GFlops
- Sobre código original
- Tamaño de simulación N=500

Laboratorio 2:

- Usando DPC++ Clang Intel OneApi
- 0.80 GFLops
- Sobre versión con forces implementada en ISPC
- Tamaño de simulación N=500

Esto en total representa una mejora aproximada de un x1.56 o un 56 % más de Gflops.



Conclusiones

- El código en ISPC tiene una performance aproximadamente 56% mejor.
- Utilizar cif en forces.ispc no mejoró el rendimiento.
- El compilador Clang DPC++ de Intel vectoriza todo el código automáticamente, tanto las versiones AoS como SoA.

Repositorio

- https://github.com/JukMR/tiny_md/