CÓMPUTO PARALELO (OPENACC – EJEMPLOS)

Francisco J. Hernández López

fcoj23@cimat.mx

NVACCELINFO

```
fco@n-gpu01:~/comp_paralelo/ejemplos/openacc$ nvaccelinfo

CUDA Driver Version:           11020
NVRM version:                  NVIDIA UNIX x86_64 Kernel Module  460.27.04  Fri Dec 11 23:35:05 UTC 2020

Device Number:                 0
Device Name:                   Tesla K40c
Device Revision Number:        3.5
Global Memory Size:            11996954624
Number of Multiprocessors:     15
Concurrent Copy and Execution: Yes
Total Constant Memory:         65536
Total Shared Memory per Block: 49152
Registers per Block:           65536
Warp Size:                     32
Maximum Threads per Block:     1024
Maximum Block Dimensions:      1024, 1024, 64
Maximum Grid Dimensions:       2147483647 x 65535 x 65535
Maximum Memory Pitch:          2147483647B
Texture Alignment:             512B
Clock Rate:                    745 MHz
Execution Timeout:             No
Integrated Device:             No
Can Map Host Memory:           Yes
Compute Mode:                  default
Concurrent Kernels:            Yes
ECC Enabled:                   Yes
Memory Clock Rate:             3004 MHz
Memory Bus Width:              384 bits
L2 Cache Size:                 1572864 bytes
Max Threads Per SMP:           2048
Async Engines:                 2
Unified Addressing:            Yes
Managed Memory:                Yes
Concurrent Managed Memory:     No
Default Target:                cc35
fco@n-gpu01:~/comp_paralelo/ejemplos/openacc$
```

SUMA DE VECTORES

```
7   #include <stdio.h>
8   #include <stdlib.h>
9   #include <iostream>
10  #include <chrono>
11
12  // Función que se ejecuta en la CPU.
13  void Suma_vectores_secuencial(float* c, float* a, float* b, long long N);
14
15  // Función paralelizada con OpenACC (Se puede ejecutar en el Multicore o la GPU)
16  void Suma_vectores_paralelo(float* c, float* a, float* b, long long N);
17
18  // Código principal que se ejecuta en el Host (CPU)
19  int main(int argc, char** argv) {
20      std::chrono::high_resolution_clock::time_point t_ini, t_fin;
21      std::chrono::duration<double> proc_time_sec, proc_time_par;
22      float* a_h, * b_h, * c1_h, *c2_h;  //Punteros a arreglos en el Host
23      long long N = 100000000;  //Número de elementos en los arreglos
24      int M = 10;//Número de elementos a desplegar de los vectores (M<N)
25      size_t size = N * sizeof(float);
```

```
32      //Pedimos memoria en el Host
33      a_h = (float*)malloc(size);
34      b_h = (float*)malloc(size);
35      c1_h = (float*)malloc(size);
36      c2_h = (float*)malloc(size);
37
38      //Inicializamos los arreglos a,b en el Host
39      for (long long i = 0; i < N; i++) {
40          a_h[i] = (float)i;
41          b_h[i] = (float)(i + 1);
42      }
49      t_ini = std::chrono::high_resolution_clock::now();
50      //Realizamos el cálculo en el CPU
51      Suma_vectores_secuencial(c1_h, a_h, b_h, N);
52      t_fin = std::chrono::high_resolution_clock::now();
53      proc_time_sec = std::chrono::duration_cast<std::chrono::duration<double>>(t_fin - t_ini);
55      t_ini = std::chrono::high_resolution_clock::now();
56      //Realizamos el cálculo en algún device
57      Suma_vectores_paralelo(c2_h, a_h, b_h, N);
58      t_fin = std::chrono::high_resolution_clock::now();
59      proc_time_par = std::chrono::duration_cast<std::chrono::duration<double>>(t_fin - t_ini);
65      //Validando resultado
66      int ban = 0;
67      for (long long idx = 0; idx < N; idx++) {
68          if (c1_h[idx] != c2_h[idx]) {
69              ban = 1;
70              break;
71          }
72      }
73      if (ban)printf("Error en los resultados...");
```

```
75      printf("\n\nTiempo de procesamiento secuencial: %lf segundos.",
76             proc_time_sec.count());
77      printf("\nTiempo de procesamiento paralelo: %lf segundos.",
78             proc_time_par.count());
81      // Liberamos la memoria del Host
82      free(a_h);
83      free(b_h);
84      free(c1_h);
85      free(c2_h);
86
87      return(0);
88  }
90  // Función que se ejecuta en la CPU.
91  void Suma_vectores_secuencial(float* c, float* a, float* b, long long N)
92  {
93      for (long long idx = 0; idx < N; idx++) {
94          c[idx] = a[idx] + b[idx];
95      }
96  }
97  // Función paralelizada con OpenACC (Se puede ejecutar en el Multicore o la GPU)
98  void Suma_vectores_paralelo(float* c, float* a, float* b, long long N)
99  {
100 #pragma acc parallel loop
101     for (long long idx = 0; idx < N; idx++) {
102         c[idx] = a[idx] + b[idx];
103     }
104 }
```

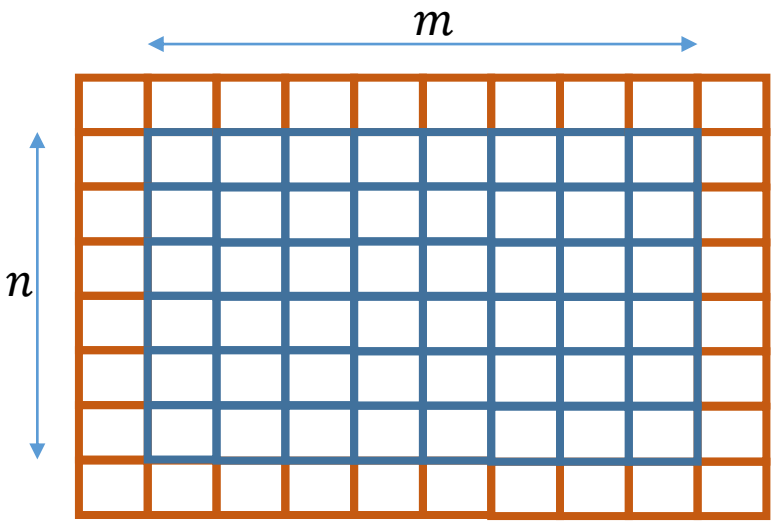
COMPILACIÓN

- Para ejecución en el host:
  - nvc++ suma_vectores.cpp -o suma_vect
- Para ejecución en el Multi-core:
  - nvc++ -acc=multicore suma_vectores.cpp -o suma_vect
- Para ejecución en la GPU:
  - nvc++ -acc suma_vectores.cpp -o suma_vect

ECUACIÓN DEL CALOR

$$\Delta u = \nabla^2 u = 0$$

$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$ son las segundas derivadas parciales



Aprox. por diferencias finitas tenemos

$$\frac{\partial^2 u}{\partial x^2} = \frac{u_{i,j-1} + u_{i,j+1} - 2u_{i,j}}{h_x^2}$$

$$\frac{\partial^2 u}{\partial y^2} = \frac{u_{i-1,j} + u_{i+1,j} - 2u_{i,j}}{h_y^2}$$

Asumiendo $h_x = h_y$ y condiciones de temperatura inicial en las orillas de la malla, iteramos la siguiente relación:

$$u_{i,j}^{k+1} = \frac{\left(u_{i+1,j}^k + u_{i-1,j}^k + u_{i,j+1}^k + u_{i,j-1}^k\right)}{4}$$

Chandrasekaran, S., & Juckeland, G. (2017). *OpenACC for Programmers: Concepts and Strategies*. Addison-Wesley Professional.

LaplaceEq_MemEstaticaDinamica.cpp

LaplaceEq | (Global Scope)

```
7   Actualización: 27/Feb/2021
8   */
9
10  #include <stdio.h>
11  #include <chrono>
12
13  //OpenCV
14  #include <opencv2/opencv.hpp>
15
16  const int imageW = 321;//Considerando dos filas y dos columnas
17  const int imageH = 241;//para las condiciones iniciales en los límites de la imagen
18
19  void condicionesIniciales_memEstatica(unsigned char* data, float temp_1, float temp_2,
20                                        float temp_3, float temp_4, float rho);
21  void condicionesIniciales_memDinamica(unsigned char* data, float temp_1, float temp_2,
22                                        float temp_3, float temp_4,float rho);
23
24  //secuencial
25  void Laplace_Jacobi_memEstatica(unsigned char* data, int MaxIter, float tol);
26  void Laplace_Jacobi_memDinamica(unsigned char* data, int MaxIter,float tol);
27
28  //acc
29  void Laplace_Jacobi_memEstatica_acc(unsigned char* data, int MaxIter, float tol);
30  void Laplace_Jacobi_memDinamica_acc(unsigned char* data, int MaxIter, float tol);
```

```
32  int main(void) {
33      std::chrono::high_resolution_clock::time_point t_ini, t_fin;
34      std::chrono::duration<double> proc_time;
35      //Parámetros
36      /*float temp_1 = 150;//Temperaturas en cierta región de las orillas de la malla
37      float temp_2 = 100;
38      float temp_3 = 200;
39      float temp_4 = 25;*/
40      float temp_1 = 200;//Temperaturas en cierta región de las orillas de la malla
41      float temp_2 = 200;
42      float temp_3 = 200;
43      float temp_4 = 200;
44      float rho    = 0.5f;//Proporción para llenar la región con cierta temperatura
45      int MaxIter = 50000;//Máximo de Iteraciones del método de Jacobi
46      float tol = 1e-7f;//Tolerancia
47      //Crear la imagen o malla 2D con condiciones iniciales de temperatura
48
49      //float u[imageH][imageW];
50      cv::Mat I, I_Gray, I_RGB;
51      I.create(imageH,imageW,CV_32FC1);//malla 2D con tipo de dato float
52      I_Gray.create(imageH, imageW, CV_8UC1);
53      I_RGB.create(imageH, imageW, CV_8UC3);
```

```
55      //Condiciones iniciales
56      condicionesIniciales_memEstatica(I.data,temp_1, temp_2, temp_3, temp_4,rho);
57      //condicionesIniciales_memDinamica(I.data, temp_1, temp_2, temp_3, temp_4, rho);
58
59      //Resolviendo Ec. de Laplace
60      t_ini = std::chrono::high_resolution_clock::now();
61      Laplace_Jacobi_memEstatica(I.data, MaxIter,tol);
62      //Laplace_Jacobi_memEstatica_acc(I.data, MaxIter, tol);
63      //Laplace_Jacobi_memDinamica(I.data, MaxIter, tol);
64      //Laplace_Jacobi_memDinamica_acc(I.data, MaxIter, tol);
65      t_fin = std::chrono::high_resolution_clock::now();
66      proc_time = std::chrono::duration_cast<std::chrono::duration<double>>(t_fin - t_ini);
67      printf("\nTiempo de procesamiento: %lf segundos.\n",proc_time.count());
68
69      //Mostrar resultados
70      //memcpy(I.data,u,imageH*imageW*sizeof(float));
71      cv::normalize(I, I, 1.0, 0.0,cv::NORM_MINMAX);
72      I.convertTo(I_Gray, CV_8UC1, 255, 0);
73      cv::applyColorMap(I_Gray,I_RGB,cv::COLORMAP_HOT);
74
75      cv::imwrite("I_LaplaceEq.png", I_RGB);
76
81      //Liberar memoria
82      I.release();
83      I_Gray.release();
84      I_RGB.release();
85
86      return (0);
87  }
```

```
89  void condicionesIniciales_memEstatica(unsigned char *data, float temp_1, float temp_2,
90                                        float temp_3, float temp_4, float rho) {
91      //float* im_data = reinterpret_cast<float*>(data);
92      float data_float[imageH][imageW];
93      //Iniciamos todos los datos en ceros
94      memset(data_float, 0, imageW* imageH * sizeof(float));
95
96      //paredes horizontales o a lo ancho de la malla
97      int inc_x = (int)((rho * imageW) / 2);
98      int ini_x = imageW / 2 - inc_x;
99      int fin_x = imageW / 2 + inc_x;
100     //paredes verticales o a lo alto de la malla
101     int inc_y = (int)((rho * imageH) / 2);
102     int ini_y = imageH / 2 - inc_y;
103     int fin_y = imageH / 2 + inc_y;
105     for (int j = ini_x; j < fin_x; j++) {
106         //i=0
107         data_float[0][j] = temp_1;
108         //i=imageH-1
109         data_float[imageH - 1][j] = temp_3;
110     }
111     for (int i = ini_y; i < fin_y; i++) {
112         //j=0
113         data_float[i][0] = temp_4;
114         //j=imageW-1
115         data_float[i][imageW - 1] = temp_2;
116     }
117
118     memcpy(data, data_float, imageW * imageH * sizeof(float));
119 }
```

```
121 void condicionesIniciales_memDinamica(unsigned char *data,float temp_1,float temp_2,
122                                         float temp_3,float temp_4,float rho) {
123     float *data_float= reinterpret_cast<float*>(data);
124     //Iniciamos todos los datos en ceros
125     memset(data_float, 0, imageW * imageH * sizeof(float));
126 
127     //paredes horizontales o a lo ancho de la malla
128     int inc_x = (int)((rho * imageW)/2);
129     int ini_x = imageW / 2 - inc_x;
130     int fin_x = imageW / 2 + inc_x;
131     //paredes verticales o a lo alto de la malla
132     int inc_y = (int)((rho * imageH) / 2);
133     int ini_y = imageH / 2 - inc_y;
134     int fin_y = imageH / 2 + inc_y;
135     long int idx1,idx2;
136     long int idx3 = (long int)(imageH - 1) * imageW;
138     for (int j = ini_x; j < fin_x; j++) {
139         idx1 = j;//i=0
140         data_float[idx1] = temp_1;
141         idx2 = idx3 + j;//i=imageH-1
142         data_float[idx2] = temp_3;
143     }
144     for (int i = ini_y; i < fin_y; i++) {
145         idx1 = (long int)i * imageW;//j=0
146         data_float[idx1] = temp_4;
147         idx2 = idx1 + imageW - 1;//j=imageW-1
148         data_float[idx2] = temp_2;
149     }
150 }
```

```
151 void Laplace_Jacobi_memEstatica(unsigned char *data, int MaxIter, float tol) {
152     float u_kp0[imageH][imageW];
153     float u_kp1[imageH][imageW];
154     memcpy(u_kp0, data, imageH * imageW * sizeof(float));
155     int k;
156     float mse=1e7;//error cuadrático medio
157     long int tam_imag = imageW * imageH;
158     for (k = 0; k < MaxIter && mse>tol; k++) {
159         mse = 0.0f;
160         for (int i = 1; i < imageH - 1; i++) {//Vamos a procesar dejando un margen de 1 pixel
161             for (int j = 1; j < imageW - 1; j++) {
162                 u_kp1[i][j] = (u_kp0[i-1][j] + u_kp0[i][j-1] + u_kp0[i+1][j] + u_kp0[i][j+1]) / 4.0f;
163 
164                 float resta = u_kp1[i][j] - u_kp0[i][j];
165                 mse += resta * resta;
166             }
167         }
168         mse /= tam_imag;
169         if (k % 10000 == 0) {
170             printf("\niter: %d, mse: %.10f", k, mse);
171         }
172         //actualizamos u_kp0
173         //memcpy(u_kp0, u_kp1, size_imag_bytes);
174         for (int i = 1; i < imageH - 1; i++) {
175             for (int j = 1; j < imageW - 1; j++) {
176                 u_kp0[i][j] = u_kp1[i][j];
177             }
178         }
179     }
180     memcpy(data, u_kp0, imageH * imageW * sizeof(float));
181     printf("\nNúmero de Iteraciones: %d, mse: %e", k, mse);
182 }
```

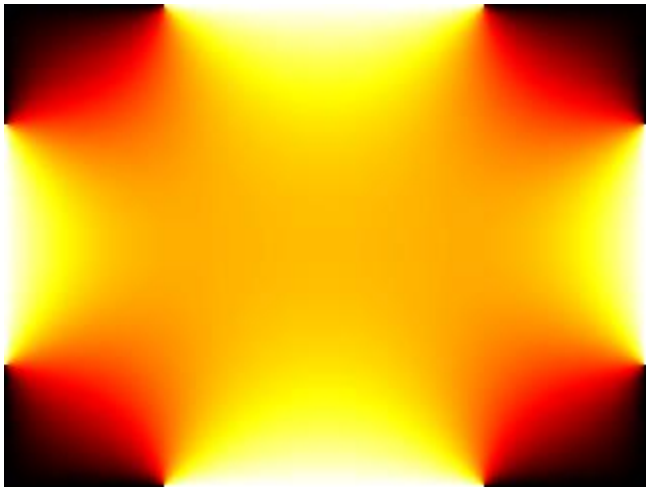
EJECUTANDO EL CÓDIGO SECUENCIAL

- nvc++ `pkg-config --cflags opencv4` LaplaceEq_MemEstaticaDinamica.cpp -o LaplaceEq_sec `pkg-config --libs opencv4`

./LaplaceEq_sec

iter: 0, mse: 18.0969734192
iter: 10000, mse: 0.0000156189
iter: 20000, mse: 0.0000040188
iter: 30000, mse: 0.0000010534
iter: 40000, mse: 0.0000002762
Número de Iteraciones: 47602,
mse: 9.999529e-08
Tiempo de procesamiento: **22.05 segundos**.

Resultado



```
184 void Laplace_Jacobi_memEstatica_acc(unsigned char* data, int MaxIter, float tol) {
185     float u_kp0[imageH][imageW];
186     float u_kp1[imageH][imageW];
187     memcpy(u_kp0, data, imageH * imageW * sizeof(float));
188     int k;
189     float mse = 1e7;//error cuadrático medio
190     long int tam_imag = imageW * imageH;
191 
192     for (k = 0; k < MaxIter && mse>tol; k++) {
193         mse = 0.0f;
194 #pragma acc kernels
195         for (int i = 1; i < imageH - 1; i++) {//Vamos a procesar dejando un margen de 1 pixel
196             for (int j = 1; j < imageW - 1; j++) {
197                 u_kp1[i][j] = (u_kp0[i - 1][j] + u_kp0[i][j - 1] + u_kp0[i + 1][j] + u_kp0[i][j + 1]) / 4.0f;
198 
199                 float resta = u_kp1[i][j] - u_kp0[i][j];
200                 mse += resta * resta;
201             }
202         }
203         mse /= tam_imag;
204         if (k % 10000 == 0) {
205             printf("\niter: %d, mse: %.10f", k, mse);
206         }
207         //actualizamos u_kp0
208 #pragma acc kernels
209         for (int i = 1; i < imageH - 1; i++) {
210             for (int j = 1; j < imageW - 1; j++) {
211                 u_kp0[i][j] = u_kp1[i][j];
212             }
213         }
214     }
215     memcpy(data,u_kp0, imageH * imageW * sizeof(float));
216     printf("\nNúmero de Iteraciones: %d, mse: %e", k, mse);
217 }
```

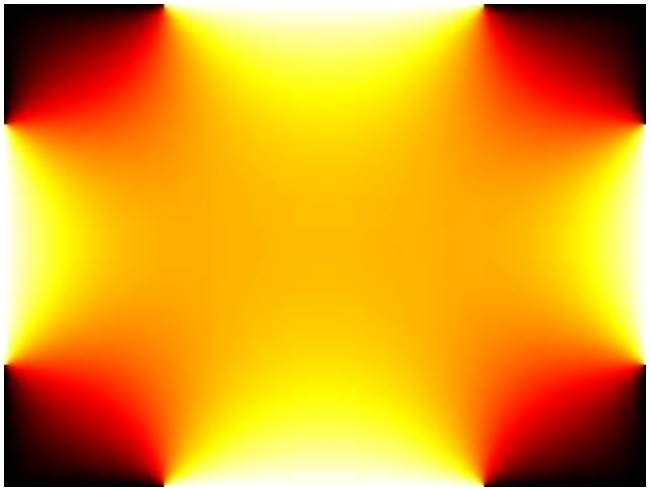
EJECUTANDO EL CÓDIGO PARALELO

- nvc++ -acc `pkg-config --cflags opencv4` LaplaceEq_MemEstaticaDinamica.cpp -o LaplaceEq_par1 `pkg-config --libs opencv4`

./LaplaceEq_par1

iter: 0, mse: 18.0969734192
iter: 10000, mse: 0.0000156186
iter: 20000, mse: 0.0000040187
iter: 30000, mse: 0.0000010534
iter: 40000, mse: 0.0000002763
Número de Iteraciones: 47605,
mse: 9.999439e-08
Tiempo de procesamiento: **16.76 segundos**.
Speedup = **1.32x**

Resultado



-MINFO=ACCEL

- nvc++ -acc **-Minfo=accel** `pkg-config --cflags opencv4` LaplaceEq_MemEstaticaDinamica.cpp -o LaplaceEq_par1 `pkg-config --libs opencv4`

```
Laplace_Jacobi_memEstatica_acc(unsigned char *, int, float):
    193, Generating implicit copyout(u_kp1[1:239][1:319]) [if not already present]
         Generating implicit copyin(u_kp0[:][:]) [if not already present]
         Generating implicit copy(mse) [if not already present]
    195, Loop is parallelizable
    196, Loop is parallelizable
         Generating Tesla code
        195, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
             Generating implicit reduction(+:mse)
        196,   /* blockIdx.x threadIdx.x auto-collapsed */
    206, Generating implicit copyin(u_kp1[1:239][1:319]) [if not already present]
         Generating implicit copyout(u_kp0[1:239][1:319]) [if not already present]
    209, Loop is parallelizable
    210, Loop is parallelizable
         Generating Tesla code
        209, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
        210,   /* blockIdx.x threadIdx.x auto-collapsed */
```

```
184 void Laplace_Jacobi_memEstatica_acc(unsigned char* data, int MaxIter, float tol) {
185     float u_kp0[imageH][imageW];
186     float u_kp1[imageH][imageW];
187     memcpy(u_kp0, data, imageH * imageW * sizeof(float));
188     int k;
189     float mse = 1e7;//error cuadrático medio
190     long int tam_imag = imageW * imageH;
191 #pragma acc data copy(u_kp0), create(u_kp1)
192     for (k = 0; k < MaxIter && mse>tol; k++) {
193         mse = 0.0f;
194 #pragma acc kernels
195         for (int i = 1; i < imageH - 1; i++) {//Vamos a procesar dejando un margen de 1 pixel
196             for (int j = 1; j < imageW - 1; j++) {
197                 u_kp1[i][j] = (u_kp0[i - 1][j] + u_kp0[i][j - 1] + u_kp0[i + 1][j] + u_kp0[i][j + 1]) / 4.0f;
198 
199                 float resta = u_kp1[i][j] - u_kp0[i][j];
200                 mse += resta * resta;
201             }
202         }
203         mse /= tam_imag;
204         if (k % 10000 == 0) {
205             printf("\niter: %d, mse: %.10f", k, mse);
206         }
207         //actualizamos u_kp0
208 #pragma acc kernels
209         for (int i = 1; i < imageH - 1; i++) {
210             for (int j = 1; j < imageW - 1; j++) {
211                 u_kp0[i][j] = u_kp1[i][j];
212             }
213         }
214     }
215     memcpy(data,u_kp0, imageH * imageW * sizeof(float));
216     printf("\nNúmero de Iteraciones: %d, mse: %e", k, mse);
217 }
```

-MINFO=ACCEL

- nvc++ -acc -Minfo=acc `pkg-config --cflags opencv4` LaplaceEq_MemEstaticaDinamica.cpp -o LaplaceEq_par2 `pkg-config --libs opencv4`

```
Laplace_Jacobi_memEstatica_acc(unsigned char *, int, float):
    192, Generating copy(u_kp0[:][:]) [if not already present]
         Generating create(u_kp1[:][:]) [if not already present]
    193, Generating implicit copy(mse) [if not already present]
    195, Loop is parallelizable
    196, Loop is parallelizable
         Generating Tesla code
        195, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
             Generating implicit reduction(+:mse)
        196,   /* blockIdx.x threadIdx.x auto-collapsed */
    209, Loop is parallelizable
    210, Loop is parallelizable
         Generating Tesla code
        209, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
        210,   /* blockIdx.x threadIdx.x auto-collapsed */
```

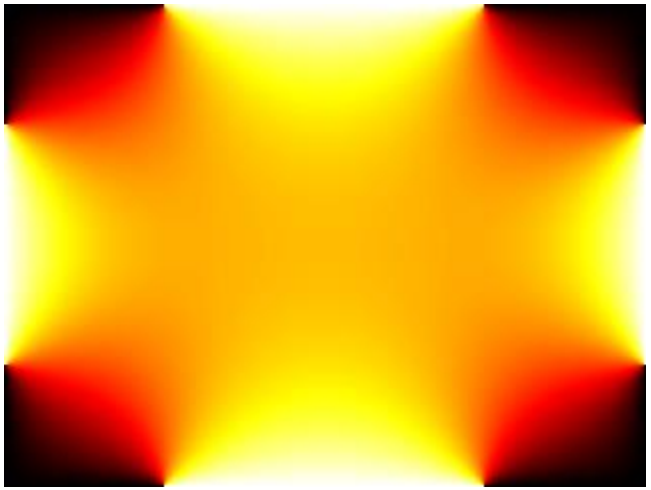
EJECUTANDO EL CÓDIGO PARALELO

- nvc++ -acc -Minfo=acc `pkg-config --cflags opencv4` LaplaceEq_MemEstaticaDinamica.cpp -o LaplaceEq_par2 `pkg-config --libs opencv4`

./LaplaceEq_par2

iter: 0, mse: 18.0969734192
iter: 10000, mse: 0.0000156186
iter: 20000, mse: 0.0000040187
iter: 30000, mse: 0.0000010534
iter: 40000, mse: 0.0000002763
Número de Iteraciones: 47605,
mse: 9.999439e-08
Tiempo de procesamiento: **2.90 segundos**.
Speedup = **7.6x**

Resultado



USANDO MEMORIA DINÁMICA

```
221 void Laplace_Jacobi_memDinamica(unsigned char *data,int MaxIter,float tol) {
222     float* u_kp0 = reinterpret_cast<float*>(data);
223     float* u_kp1;
224     int k;
225     float mse = 1e7;//error cuadrático medio
226     long int tam_imag = imageW * imageH;
227     size_t size_imag_bytes = tam_imag * sizeof(float);
228     u_kp1 = (float*)malloc(size_imag_bytes);
229     //memcpy(u_kp1, u_kp0, size_imag_bytes);
230     for (k = 0; k < MaxIter && mse>tol;k++) {
231         mse = 0.0f;
232         for (int i = 1; i < imageH-1; i++) {//Vamos a procesar dejando un margen de 1 pixel
233             for (int j = 1; j < imageW-1; j++) {
234                 long int idx_11 = (long int)i * imageW + j;
235                 long int idx_01 = (long int)(i-1) * imageW + j;
236                 long int idx_10 = (long int)i * imageW + j-1;
237                 long int idx_21 = (long int)(i + 1) * imageW + j;
238                 long int idx_12 = (long int)i * imageW + j + 1;
239                 u_kp1[idx_11] = (u_kp0[idx_01] + u_kp0[idx_10] + u_kp0[idx_21] + u_kp0[idx_12]) / 4.0f;
240
241                 float resta = u_kp1[idx_11] - u_kp0[idx_11];
242                 mse += resta*resta;
243             }
244         }
245         mse /= tam_imag;
246         if (k % 10000 == 0) {
247             printf("\niter: %d, mse: %.10f",k,mse);
248         }
249         //actualizamos u_kp0
250         //memcpy(u_kp0, u_kp1, size_imag_bytes);
251         for (int i = 1; i < imageH - 1; i++) {
252             for (int j = 1; j < imageW - 1; j++) {
253                 long int idx_11 = (long int)i * imageW + j;
254                 u_kp0[idx_11] = u_kp1[idx_11];
255             }
256         }
257     }
258     printf("\nNúmero de Iteraciones: %d, mse: %e", k, mse);
259     free(u_kp1);
260 }
```

EJECUTANDO EL CÓDIGO SECUENCIAL

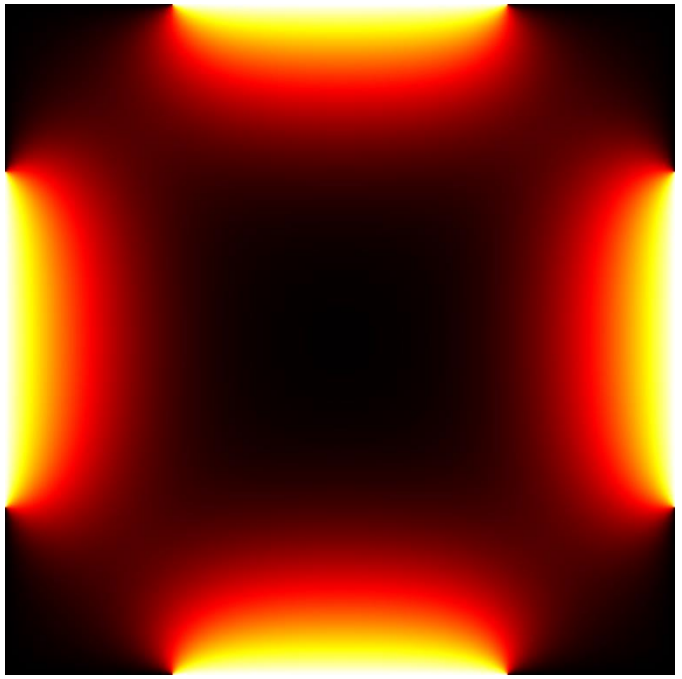
```
const int imageW = 1024; //Probando con una imagen más grande
const int imageH = 1024;
```

- nvc++ `pkg-config --cflags opencv4` LaplaceEq_MemEstaticaDinamica.cpp -o LaplaceEq_sec_memdin `pkg-config --libs opencv4`

./LaplaceEq_sec_memdin

iter: 0, mse: 4.8828125000
iter: 10000, mse: 0.0000032874
iter: 20000, mse: 0.0000010965
iter: 30000, mse: 0.0000005932
iter: 40000, mse: 0.0000003945
Número de Iteraciones: **50000**,
mse: 2.925066e-07
Tiempo de procesamiento: **363.99 segundos**.

Resultado



```
266 void Laplace_Jacobi_memDinamica_acc(unsigned char* data, int MaxIter, float tol) {
267     float *u_kp0 = reinterpret_cast<float*>(data);
268     float *u_kp1;
269
270     int k;
271     float mse=1e7;//error cuadrático medio
272     long int tam_imag = imageW * imageH;
273     size_t size_imag_bytes = tam_imag * sizeof(float);
274     u_kp1 = (float*)malloc(size_imag_bytes);
275 #pragma acc data copy(u_kp0[0:tam_imag]), create(u_kp1[0:tam_imag])
276     for (k = 0; k < MaxIter && mse>tol; k++) {
277         mse = 0.0f;
278 #pragma acc kernels
279             for (int i = 1; i < imageH - 1; i++) {//Vamos a procesar dejando un margen de 1 pixel
280                 for (int j = 1; j < imageW - 1; j++) {
281                     u_kp1[(long int)i * imageW + j] = (u_kp0[(long int)(i - 1) * imageW + j] +
282                                                       u_kp0[(long int)i * imageW + j - 1] +
283                                                       u_kp0[(long int)(i + 1) * imageW + j] +
284                                                       u_kp0[(long int)i * imageW + j + 1]) / 4.0f;
285
286                     float resta = u_kp1[(long int)i * imageW + j] - u_kp0[(long int)i * imageW + j];
287                     mse += resta * resta;
288                 }
289             }
290             mse /= tam_imag;
291             if (k % 10000 == 0) {
292                 printf("\niter: %d, mse: %.10f", k, mse);
293             }
294             //actualizamos u_kp0
295 #pragma acc kernels
296             for (int i = 1; i < imageH - 1; i++) {
297                 for (int j = 1; j < imageW - 1; j++) {
298                     //long int idx_11 = (long int)i * imageW + j;
299                     //u_kp0[idx_11] = u_kp1[idx_11];
300                     u_kp0[(long int)i * imageW + j] = u_kp1[(long int)i * imageW + j];
301                 }
302             }
303     }
304     printf("\nNúmero de Iteraciones: %d, mse: %e", k, mse);
305     free(u_kp1);
306 }
```

-MINFO=ACCEL

- nvc++ -acc -Minfo=accel `pkg-config --cflags opencv4` LaplaceEq_MemEstaticaDinamica.cpp -o LaplaceEq_par_memdin1 `pkg-config --libs opencv4`

```
Laplace_Jacobi_memDinamica_acc(unsigned char *, int, float):
    276, Generating copy(u_kp0[:tam_imag]) [if not already present]
         Generating create(u_kp1[:tam_imag]) [if not already present]
    277, Generating implicit copy(mse) [if not already present]
    279, Complex loop carried dependence of u_kp0->,u_kp1-> prevents parallelization
         Accelerator serial kernel generated
         Generating Tesla code
        279, #pragma acc loop seq
        280, #pragma acc loop seq
    279, Complex loop carried dependence of u_kp0-> prevents parallelization
    280, Complex loop carried dependence of u_kp0->,u_kp1-> prevents parallelization
    296, Complex loop carried dependence of u_kp1->,u_kp0-> prevents parallelization
         Accelerator serial kernel generated
         Generating Tesla code
        296, #pragma acc loop seq
        297, #pragma acc loop seq
    297, Complex loop carried dependence of u_kp1->,u_kp0-> prevents parallelization
```

RESTRICT

- Ayuda al compilador a indicarle que los apuntadores no apuntan a la misma localidad de memoria (not aliased), de esta forma permite que cierta región de código no sea considerada secuencial por problemas de alias

```
263  void Laplace_Jacobi_memDinamica_acc(unsigned char* data, int MaxIter, float tol) {
264      /*float *u_kp0 = reinterpret_cast<float*>(data);
265      float *u_kp1;*/
266      float* restrict u_kp0 = reinterpret_cast<float*>(data);
267      float* restrict u_kp1;
```

-MINFO=ACCEL

- nvc++ -acc -Minfo=accel `pkg-config --cflags opencv4` LaplaceEq_MemEstaticaDinamica.cpp -o LaplaceEq_par_memdin2 `pkg-config --libs opencv4`

```
Laplace_Jacobi_memDinamica_acc(unsigned char *, int, float):
    274, Generating copy(u_kp0[:tam_imag]) [if not already present]
         Generating create(u_kp1[:tam_imag]) [if not already present]
    275, Generating implicit copy(mse) [if not already present]
    278, Loop is parallelizable
    279, Loop is parallelizable
         Generating Tesla code
        278, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
             Generating implicit reduction(+:mse)
        279,   /* blockIdx.x threadIdx.x auto-collapsed */
    296, Loop is parallelizable
    297, Loop is parallelizable
         Generating Tesla code
        296, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
        297,   /* blockIdx.x threadIdx.x auto-collapsed */
```

EJECUTANDO EL CÓDIGO PARALELO

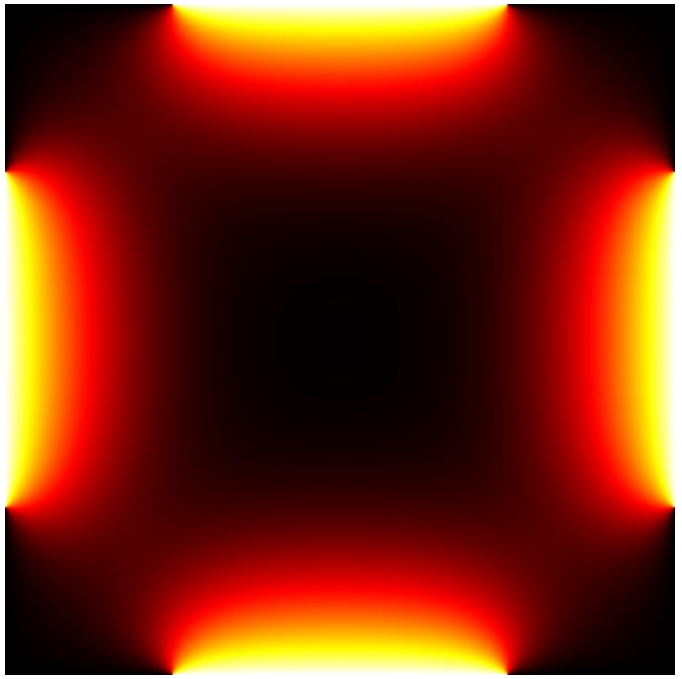
```
const int imageW = 1024; //Probando con una imagen más grande
const int imageH = 1024;
```

- nvc++ -acc `pkg-config --cflags opencv4` LaplaceEq_MemEstaticaDinamica.cpp -o LaplaceEq_par_memdin2 `pkg-config --libs opencv4`

./LaplaceEq_par_memdin2

iter: 0, mse: 4.8828125000
iter: 10000, mse: 0.0000032901
iter: 20000, mse: 0.0000010970
iter: 30000, mse: 0.0000005933
iter: 40000, mse: 0.0000003945
Número de Iteraciones: **50000**,
mse: 2.924465e-07
Tiempo de procesamiento: **11.02 segundos**.
Speedup = **33.03x**

Resultado



GRACIAS POR SU ATENCIÓN

Francisco J. Hernández-López

fcoj23@cimat.mx

WebPage:

www.cimat.mx/~fcoj23