Lab 4 - GPGPU

Ignacio Scarinci

Características de la PC

Características

- Procesador: Intel i7-6700HQ
- Memoria RAM: 16 GB (2x8GB - DDR4-2133)
- NVIDIA GEFORCE GTX950M (MAXWELL)
- S.O. UBUNTU 20.04.2 LTS
- Kernel: 5.8.0-53-generic

Características de la PC

Características de ZX81

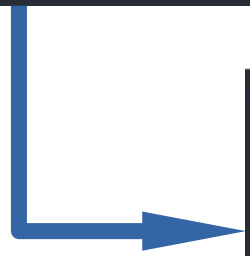
- Procesador: Intel Xeon E5-2680 v4 @ 2.40 GHz
- Memoria RAM: 128 GB
- NVIDIA GEFORCE RTX3070
- S.O. Debian 5.10.40-1
- Kernel: 5.10.0-7-amd64

1er. Intento

```
int main(int argc, char *argv[])
{
  unsigned long PHOTONS;
  unsigned int NUM_BLOCKS;
  unsigned int THREADS_POR_BLOCK;
  sscanf(argv[1], "%lu", &PHOTONS);
  sscanf(argv[2], "%iu", &NUM_BLOCKS);
  sscanf(argv[3], "%iu", &THREADS_POR_BLOCK);

  printf("# Scattering = %8.3f/cm\n", MU_S);
  printf("# Absorption = %8.3f/cm\n", MU_A);
  printf("# Photons    = %8lu\n#\n", PHOTONS);

  // Paso al esapacio de memoria constante las constantes del problema
  const float albedo = MU_S / (MU_S + MU_A);
  const float shells_per_mfp = 1e4 / MICRONS_PER_SHELL / (MU_A + MU_S);
  const int cascaras = SHELLS;
  unsigned long int n_fotones = PHOTONS;
  unsigned int hilos_tot = THREADS_POR_BLOCK * NUM_BLOCKS;
  checkCudaCall( cudaMemcpyToSymbol(num_fotones_cd, &n_fotones, sizeof(unsigned long int)));
  checkCudaCall( cudaMemcpyToSymbol(albedo_cd, &albedo, sizeof(float)));
  checkCudaCall( cudaMemcpyToSymbol(shells_per_mfp_cd, &shells_per_mfp, sizeof(float)));
  checkCudaCall( cudaMemcpyToSymbol(shells_cd, &cascaras, sizeof(int)));
  checkCudaCall( cudaMemcpyToSymbol(hilos_tot_cd, &hilos_tot, sizeof(int)));

  unsigned long long* fotones_simulados;
  fotones_simulados = 0;

  float* heat;
  float* heat2;

  checkCudaCall(cudaMallocManaged(&fotones_simulados, sizeof(unsigned long long)));
  checkCudaCall(cudaMallocManaged(&heat, cascaras*sizeof(float)));
  checkCudaCall(cudaMallocManaged(&heat2, cascaras*sizeof(float)));
  checkCudaCall(cudaMemset(heat, 0, cascaras));
  checkCudaCall(cudaMemset(heat2, 0, cascaras));

  curandStatePhilox4_32_10_t *devPHILOXStates;

  checkCudaCall(cudaMalloc((void **)&devPHILOXStates,
                           hilos_tot * sizeof(curandStatePhilox4_32_10_t)));
```

1er. Intento

```
double start = wtime();
dim3 dimBlock(THREADS_POR_BLOCK);
dim3 dimGrid(NUM_BLOCKS);
//inicializo el generado de números aleatorios
init_rng<<<dimGrid, dimBlock>>>(devPHILOXStates);
checkCudaCall( cudaDeviceSynchronize() );
```



```
__global__ void init_rng(curandStatePhilox4_32_10_t *state)
{
    int id = threadIdx.x + blockIdx.x * blockDim.x;
    // Cada hilo se inicializa con la misma semilla pero con distinta secuencia
    clock_t clock();
    long long int clock64();
    curand_init(clock64(), id, 0, &state[id]);
}
```

1er. Intento

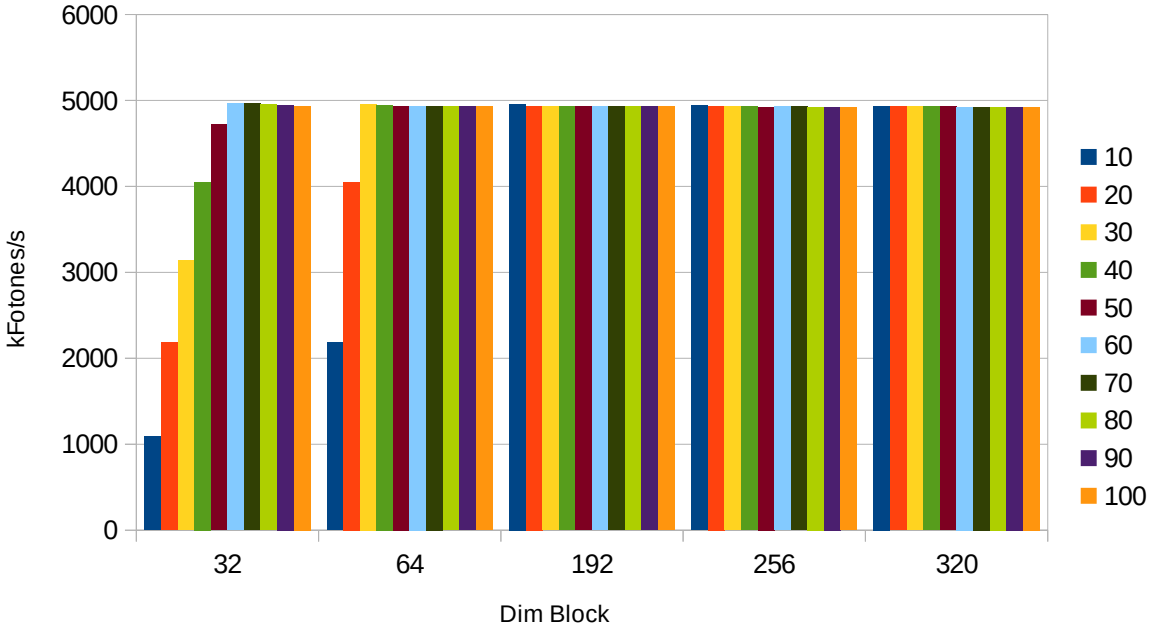
```
  simulador<<<dimGrid,dimBlock>>>(heat, heat2, devPHILOXStates, fotones_simulados);
  checkCudaCall( cudaDeviceSynchronize() );

   // stop timer
   double end = wtime();
   assert(start <= end);
   double elapsed = end - start;

   printf("# %lf seconds\n", elapsed);
   printf("# %lf K photons per second\n", 1e-3 * PHOTONS / elapsed);
   printf("# Fotones simulados    = %llu\n#\n", fotones_simulados[0]);
   printf("# Radius\tHeat\n");
   printf("# [microns]\t[W/cm^3]\tError\n");
   float t = 4.0f * M_PI * powf(MICRONS_PER_SHELL, 3.0f) * PHOTONS / 1e12;
   for (unsigned int i = 0; i < SHELLS - 1; ++i) {
       printf("%6.0f\t%12.5f\t%12.5f\n", i * (float)MICRONS_PER_SHELL,
              heat[i] / t / (i * i + i + 1.0 / 3.0),
              sqrt(heat2[i] - heat[i] * heat[i] / PHOTONS) / t / (i * i + i + 1.0f / 3.0f));
   }
   printf("# extra\t%12.5f\n", heat[SHELLS - 1] / PHOTONS);
//   checkCudaCall( cudaFree(heat));
//   checkCudaCall( cudaFree(heat2));
  return 0;
}
```

1er. Intento

```
__global__ void simulador(float* heat, float* heat2, curandStatePhilox4_32_10_t *state, unsigned long long *fotones_simulados)
{
  int id = threadIdx.x + blockIdx.x * blockDim.x;
  curandStatePhilox4_32_10_t localState = state[id];
  float t;
  FotonStruct p;
  InicializoFoton(&p);
  int termino = 0;
  while(termino == 0)
  {
    p.t = -__logf(curand_uniform(&localState));
    p.x += p.u*t;
    p.y += p.u*t;
    p.z += p.w*t;
    p.cascaron = __float2uint_rz(sqrtf(p.x * p.x + p.y * p.y + p.z * p.z) * shells_per_mfp_cd);
    if (p.cascaron > shells_cd - 1){
      p.cascaron = shells_cd - 1;
    }
    p.weight *= albedo_cd ;
    atomicAdd(&heat[p.cascaron], (1.0f - albedo_cd) * p.weight);
    atomicAdd(&heat2[p.cascaron], (1.0f - albedo_cd) * (1.0f - albedo_cd) * p.weight * p.weight);
    if (p.weight < 0.001f){
      if(curand_uniform(&localState) > 0.1f)
      {
        if(atomicAdd(fotones_simulados,1ul) < (num_fotones_cd-hilos_tot_cd))
        { //  quedan fotones por lanzar
              InicializoFoton(&p);//Lanzo foton
              continue;
        }
        termino = 1;
      }
      p.weight *= 10.0f ;
    }
    float xi1, xi2;
    do {
        xi1 = 2.0f * curand_uniform(&localState) -1.0f;
        xi2 = 2.0f * curand_uniform(&localState) -1.0f;
        t = xi1 * xi1 + xi2 * xi2;
    }while(1.0f < t);
    p.u = 2.0f * t - 1.0f;
    p.v = xi1 * sqrtf(__fdividef(1.0f - p.u * p.u, t));
    p.w = xi2 * sqrtf(__fdividef(1.0f - p.v * p.v, t));
    state[id] = localState;
  }
}
```

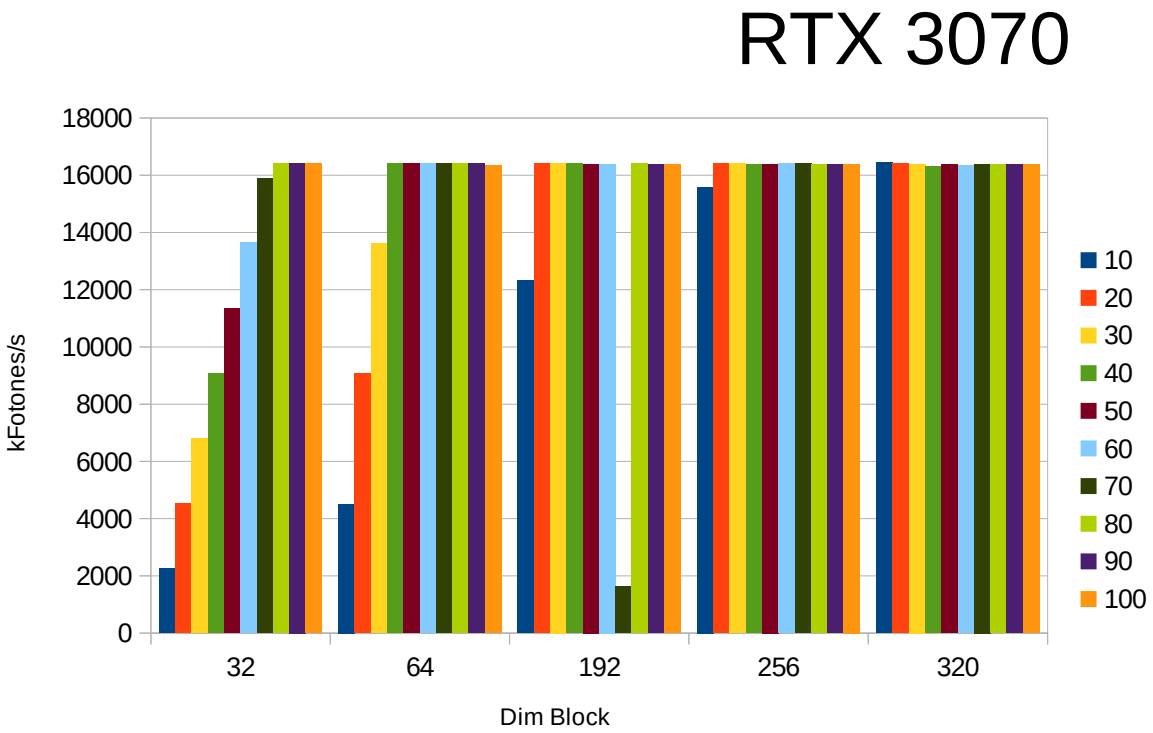
Resultados

GTX 950M



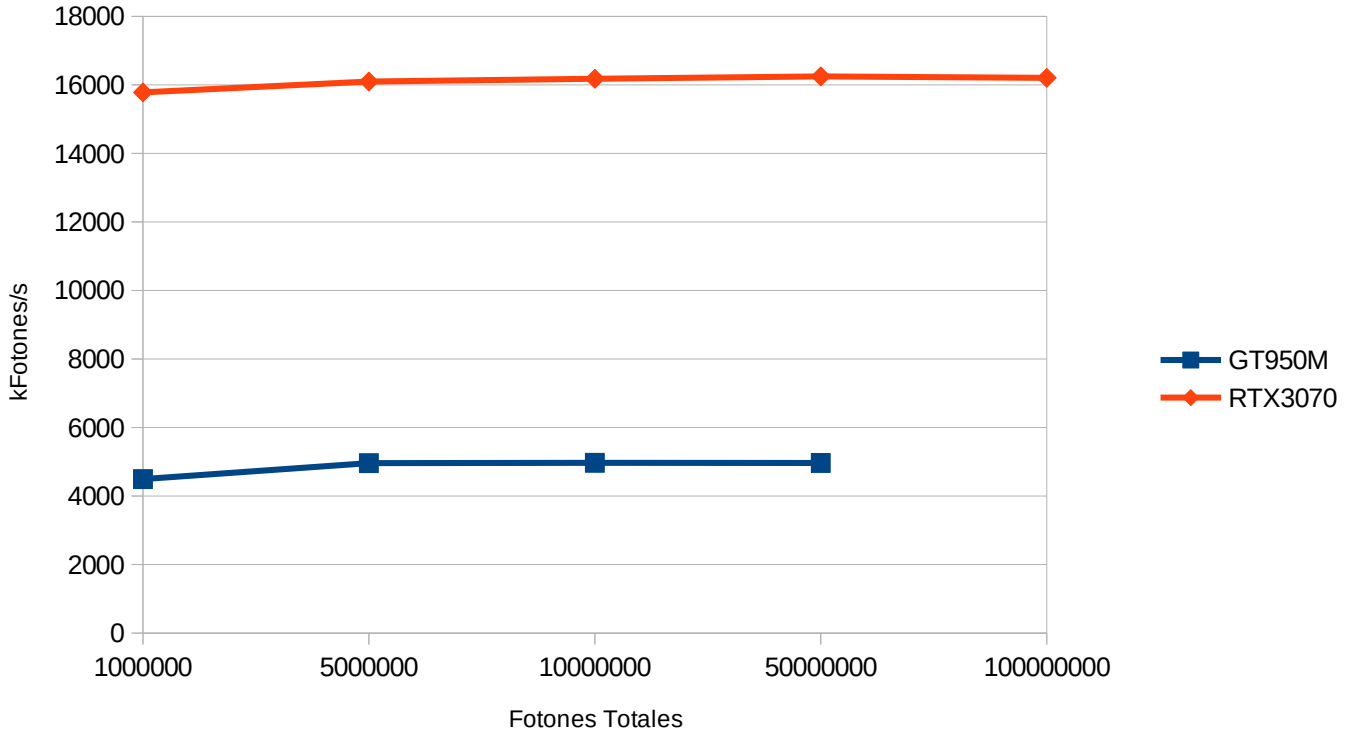
El Mejor resultado se obtuvo con Bloque de 32 y Grid de 60. La velocidad fue de 4965 kfotones/s

Resultados



El Mejor resultado se obtuvo con Bloque de 320 y Grid de 10. La velocidad fue de 16440 kfotones/s

Resultados

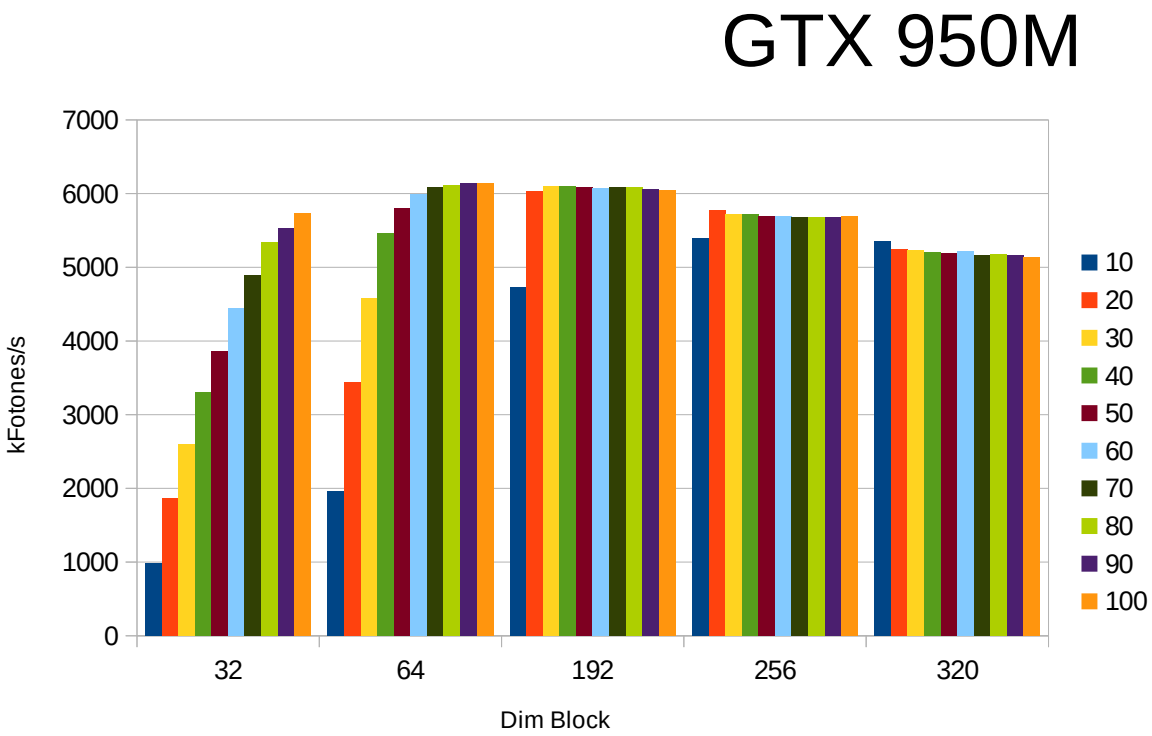

18000
16000
14000
12000
10000
8000
6000
4000
2000
0
kFotones/s
1000000
5000000
10000000
50000000
100000000
Fotones Totales
GT950M
RTX3070

2do. Intento

```
__global__ void simulador(float* heat, float* heat2, curandStatePhilox4_32_10_t *state, unsigned long long *fotones_simulados)
{
  __shared__ float heat_b[SHELLS];
  __shared__ float heat2_b[SHELLS];
  int id = threadIdx.x + blockIdx.x * blockDim.x;
  curandStatePhilox4_32_10_t localState = state[id];
  float t;
  FotonStruct p;

   for(int i=threadIdx.x;i<SHELLS; i += blockDim.x)
      {
        heat_b[i] = 0.0f;
        heat2_b[i]= 0.0f;
      }
   __syncthreads();
```

```
    __syncthreads();
    for(int i=threadIdx.x; i<SHELLS;i+=blockDim.x){
      atomicAdd(&heat[i],heat_b[i]);
      atomicAdd(&heat2[i],heat2_b[i]);
    }
    __syncthreads();
```

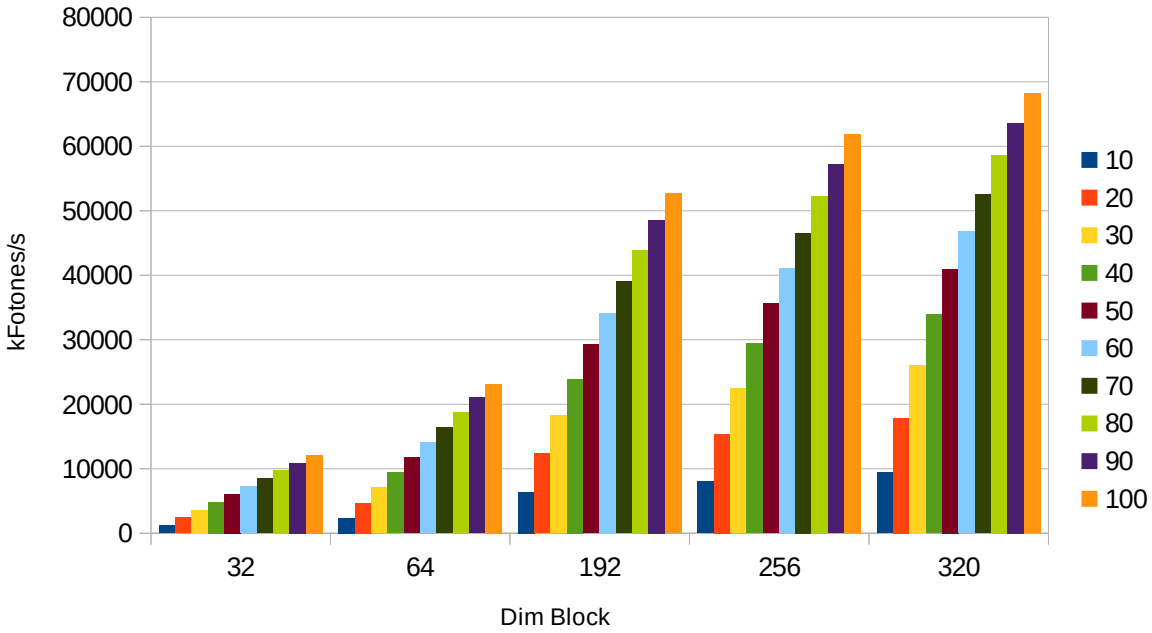
Resultados



El Mejor resultado se obtuvo con Bloque de 64 y Grid de 100. La velocidad fue de 6140 kfotones/s

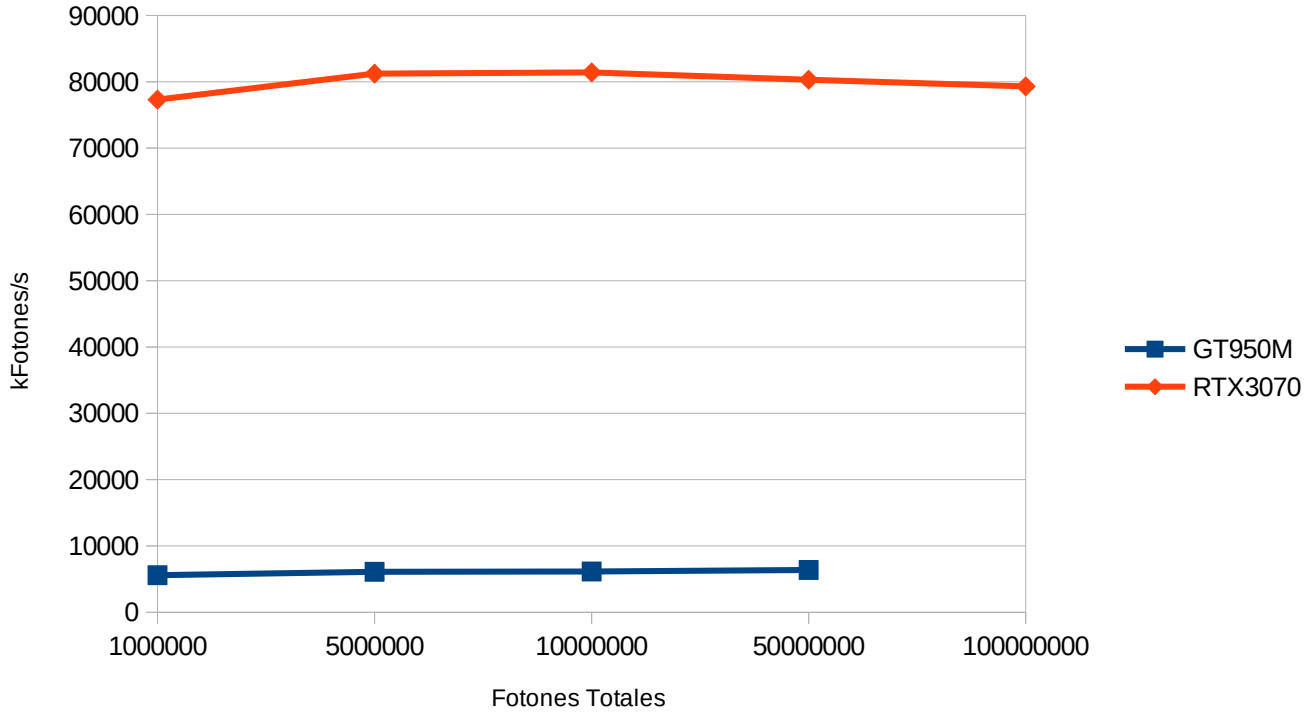
Resultados

RTX 3070

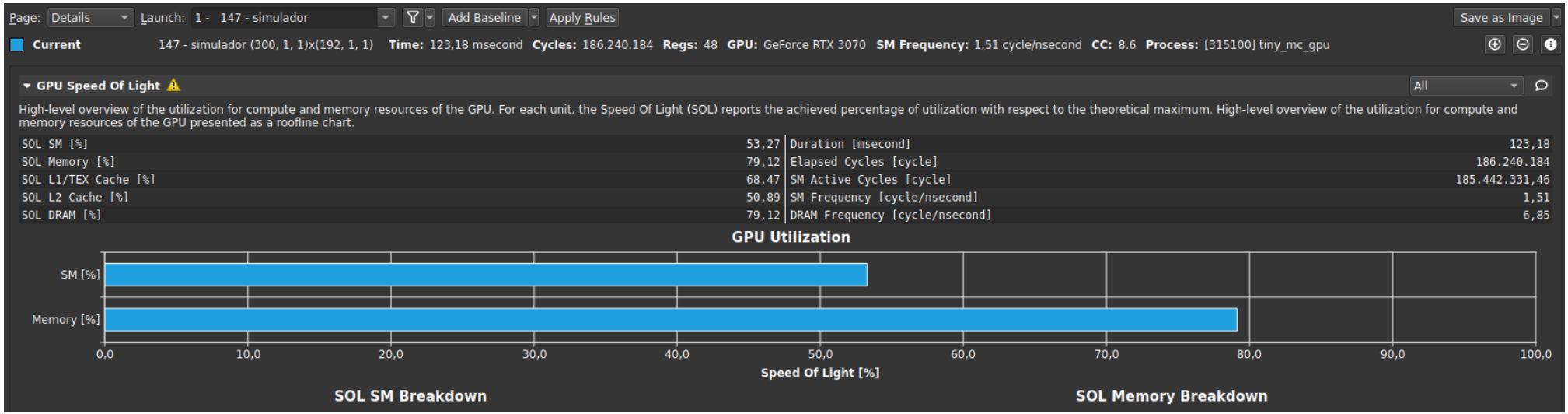


El Mejor resultado se obtuvo con Bloque de 192 y Grid de 300. La velocidad fue de 81383 kfotones/s

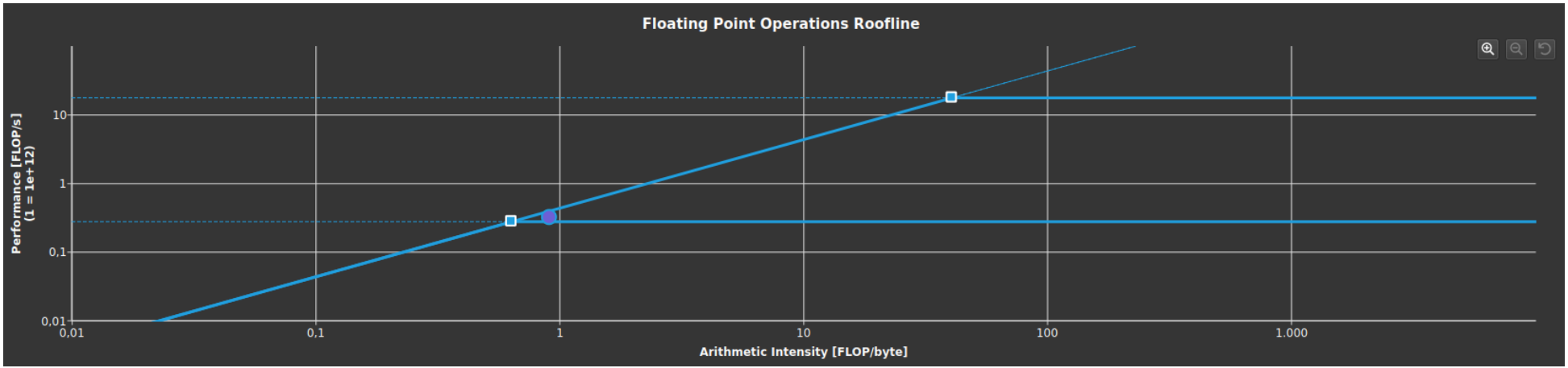
Resultados


90000
80000
70000
60000
50000
40000
30000
20000
10000
0
kFotones/s
1000000
5000000
10000000
50000000
100000000
Fotones Totales
GT950M
RTX3070

Roofline


Page: Details
Launch: 1 - 147 - simulador
Add Baseline
Apply Rules
Save as Image
Current
147 - simulador (300, 1, 1)x(192, 1, 1)
Time: 123,18 msecond
Cycles: 186.240.184
Regs: 48
GPU: GeForce RTX 3070
SM Frequency: 1,51 cycle/nsecond
CC: 8.6
Process: [315100] tiny_mc_gpu
GPU Speed Of Light
All
High-level overview of the utilization for compute and memory resources of the GPU. For each unit, the Speed Of Light (SOL) reports the achieved percentage of utilization with respect to the theoretical maximum. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.
SOL SM [%] 53,27
SOL Memory [%] 79,12
SOL L1/TEX Cache [%] 68,47
SOL L2 Cache [%] 50,89
SOL DRAM [%] 79,12
Duration [msecond] 123,18
Elapsed Cycles [cycle] 186.240.184
SM Active Cycles [cycle] 185.442.331,46
SM Frequency [cycle/nsecond] 1,51
DRAM Frequency [cycle/nsecond] 6,85
GPU Utilization
SM [%]
Memory [%]
0,0
10,0
20,0
30,0
40,0
50,0
60,0
70,0
80,0
90,0
100,0
Speed Of Light [%]
SOL SM Breakdown
SOL Memory Breakdown

Roofline

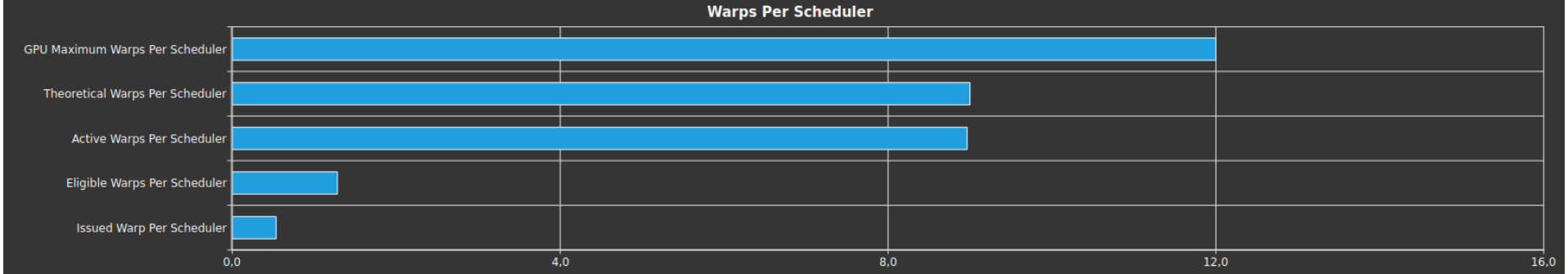

Floating Point Operations Roofline
Performance [FLOP/s]
(1 = 1e+12)
10
1
0,1
0,01
0,01
0,1
1
10
100
1.000
Arithmetic Intensity [FLOP/byte]

Roofline

Scheduler Statistics

Summary of the activity of the schedulers issuing instructions. Each scheduler maintains a pool of warps that it can issue instructions for. The upper bound of warps in the pool (Theoretical Warps) is limited by the launch configuration. On every cycle each scheduler checks the state of the allocated warps in the pool (Active Warps). Active warps that are not stalled (Eligible Warps) are ready to issue their next instruction. From the set of eligible warps the scheduler selects a single warp from which to issue one or more instructions (Issued Warp). On cycles with no eligible warps, the issue slot is skipped and no instruction is issued. Having many skipped issue slots indicates poor latency hiding.

Active Warps Per Scheduler [warp]	8,97	No Eligible [%]	46,52
Eligible Warps Per Scheduler [warp]	1,28	One or More Eligible [%]	53,48
Issued Warp Per Scheduler	0,53		



Recommendations

Issue Slot Utilization

[Warning] Every scheduler is capable of issuing one instruction per cycle, but for this kernel each scheduler only issues an instruction every 1.9 cycles. This might leave hardware resources underutilized and may lead to less optimal performance. Out of the maximum of 12 warps per scheduler, this kernel allocates an average of 8.97 active warps per scheduler, but only an average of 1.28 warps were eligible per cycle. Eligible warps are the subset of active warps that are ready to issue their next instruction. Every cycle with no eligible warp results in no instruction being issued and the issue slot remains unused. To increase the number of eligible warps either increase the number of active warps or reduce the time the active warps are stalled.

Todos contra todos

	Lab 1	Lab 2	Lab 3	Lab 4
kfotones/s	282,46	371,51	6982,59	81383,96
Speedup	1	1,32	24,72	288,13

Conclusiones

- Es relativamente fácil trasladar un código de CPU a GPU
- Resulta complicado (al menos al principio) entender y manejar de forma optima los distintos niveles de memoria