OpenCL Implementation of Fluid Simulation

Eric Blanchard*
COMP599 - Fundamentals of Computer Animation

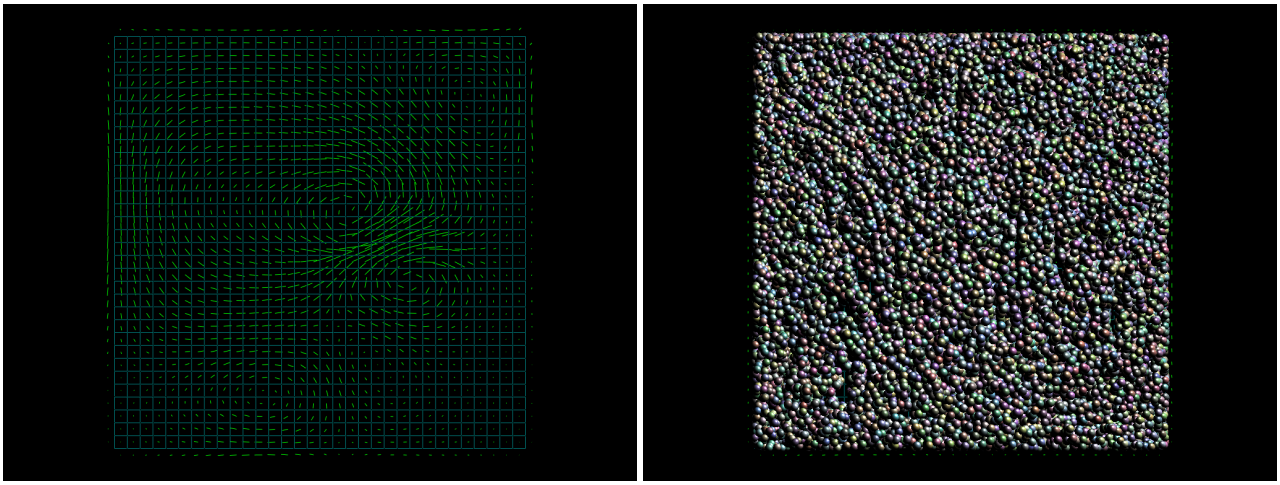

Figure 1: (a) *GPU rendered* 32×32 *velocity field grid (204 fps),* (b) *40000 particles GPU traced on the same* 32×32 *grid (62 fps)*

Keywords: Fluid Simulation, GPU architecture, Parallel Computing

1 Introduction

The goal of the assignment four was to implements a stable Eulerian fluid simulation based on the approach taken by [Stam 2003]. In [Stam 2003], the author describes a stable and efficient way to solve the Navier-Stokes equation describing fluid dynamics. The algorithm proposed is not strictly physically accurate but allow fluid simulation to be solved in a real-time context while maintaining the believability of the resulting motion, whether it be smoke, water or other types of fluids.

[Stam 2003] provides a sample implementation based on a 2D grid where each cell encode a discrete snapshot of the velocity field. For small grid size, good performance can be achieved, but the simulation does not scale well when a finer grid is required. We could also want to extends the algorithm to 3D space, but adding another dimension would also be too costly to achieve real-time results without further optimizing each step of the approach.

One solution to the performance problem is to use the GPU to help accelerate the equations solving by leveraging the power of parallels shaders to solve the fluids equation. Many authors proposed new way to parallelize fluid simulation using GPGPU techniques (like the ones described in [Tariq and Llamas 2007]). Unfortunately, those techniques are tied to the graphics architecture, often using fragment shader combined with velocity fluid encoded in texture memory [Tariq and Llamas 2007] giving efficient solution but cumbersome and difficult to maintain for non-graphics savvy developer.

In this mini-project, I propose to implements the [Stam 2003] approach on the GPU using the OpenCL library. OpenCL is a multiplatform library giving a simple interface to perform parallel computation (on the GPU or CPU).

*e-mail: eric.blanchard@polymtl.ca

2 OpenCL Overview

In this section, we will a provide a quick overview of the OpenCL library. For a more in-depth description of the inner working of GPU architecture see [Fatahalian 2009] and [Kirk and Hwu 2010]. For OpenCL specific information, the Khronos OpenCL specification is the one giving the most detailed informations [OpenCL 2009].

2.1 The OpenCL Execution Model

Computations with OpenCL are made through the use of compute *kernels*. Compute kernels are executed in parallel[1] on the device whether it be the CPU or the GPU depending on the created context target.

Before sending kernels to the device, the number of parallel workitems (*instance of kernels*) must be determined (in our case, for a $n \times n$ grid, we will queue n^2 instances). This size is known as the global dimension. The global domain needs to be splited in local workgroup before being sent to the device. Those local workgroups size are device dependent and represents the core layout on the device.

To make this clear, let's take the NVIDIA GeForce 8800 GTS GPU (one of the GPU used in this project) as an example. The NVIDIA specification tells us that this specific GPU have 96 CUDA cores while OpenCL reports 12 cores. The 8800 specification [Corporation 2005] shows that each group of 8 scalar core are grouped together and share the instruction decode stage. So OpenCL reports the number independent cores while that NVIDIA reports the total number of scalar processor (CUDA cores, marketing wise 96 is a bigger number than 12). Of course we don't need too delve deep into hardware specification for using OpenCL, this exercise was to explain the source of the workgroup concept.

Once we have determined the size of our global and local work-

[1] OpenCL also allow a *task* mode where each kernel are executed on a single core

OpenCL	Graphics
Kernel/Work-item	Shader Program
Local Memory	Constant/Uniform memory
Global Memory	Texture memory

Table 1: *OpenCL vs. Graphics*

group size, we must enqueue the compute kernels through the OpenCL work queue. OpenCL support the standard FIFO model but also an out-of-order execution mode. In this project, only the FIFO type is used. For kernels that are executed in the same local workgroup, it is possible to use synchronization elements to synchronize execution. Unfortunately, there's is no such thing at the global scale.

2.2 The OpenCL Memory Model

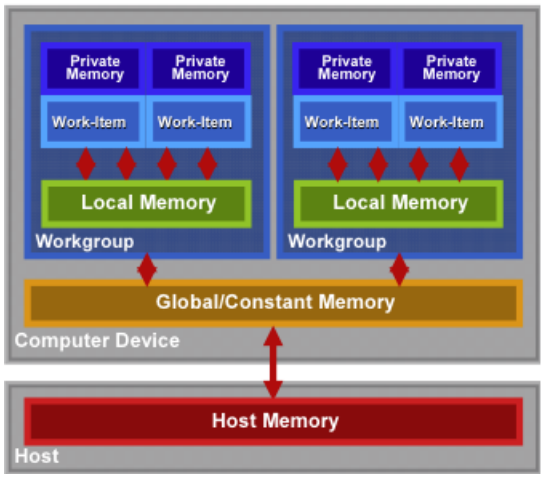

Figure 2: *OpenCL Memory Model*

Before sending our kernel for execution we must have the data ready to source the computations. OpenCL kernels can't access host memory, we must then move data to the device global memory. Figure 2 (taken from [OpenCL 2009]) shows an overview of the OpenCL memory model.

2.2.1 OpenCL vs. Graphics API

Since OpenCL can be seen as abstraction of the GPU computation model, table 1 shows a little side by side comparison of the technical terms used in OpenCL and the graphics jargon.

3 Implementation

The implementation is based on fluid simulation described in [Stam 2003] where each operation has been modified to run as parallel kernels on the GPU. Figure 3 shows the GPU accelerated execution pipeline. The next sections will elaborate on the steps shown on figure 3.

3.1 Step 1/7: Host/GPU Memory Interface

As can be seen in figure 2, the data on which the OpenCL kernels are executed need to reside in the device local memory. Two copy of the velocity field buffers are kept: one in host memory and the other in the device memory. On each velocity field update, we must transfer the content of the velocity field from $host \rightarrow device$ and $device \rightarrow host$. Note that this bus transfer is costly and have a great impact on performance with small grid where the cost of the bus transfer is much higher than the cost of the computation.

3.2 Step 2: The Diffuse Step

The diffuse step implies the exchange of velocity value with the surrounding cells. The kernel for this step is coded as the view from a single grid cell (the execution can be seen as each grid cell computed in parallel). Local memory fetch is made to access the neighboring velocities and the boundary cases are included in this kernel to avoid scheduling an exclusive pass for boundary checking.

Since the diffuse step is solved with the iterative Gauss-Seidel method, the diffuse kernel is queued 30 times (the default value in [Stam 2003]) on an in order command queue. We then obtain the same results as the CPU iterative method but with each step executed in parallel.

3.3 Step 3/5: The Projection Step

The projection step ensure mass conservation after the diffuse and transport step. This step is separated in three distinct kernel:

1. Parallel initialization of the div and p buffers.
2. Gauss-Seidel solving for p.
3. Parallel adjustments for the velocity field.

Once again, boundary checking have been merge in each of these 3 kernels, and each of them is executed in order giving us: $Initialization \rightarrow 30 \times GSsolving \rightarrow Vel.Adjustments$.

3.4 Step 4: The Transport Step

The transport step move the velocity field based on his own velocity values. Since the velocity field is moving itself, we need two distinct buffers for the velocity field, one for the source, the other for the destination.

To solve the transport step, one kernel is executed per grid cell. On each grid cell, the velocity is computed by back-tracing the current grid position with the grid cell velocity. The new position is then use to do a bi-linear interpolation with the surrounding velocities.

3.5 Step 9-10: The Trace Step

The trace step is somewhat similar to the transport step where, instead of transporting velocity, we move particle positions. Since we use OpenGL for rendering, the particle positions are encoded in a vertex buffer object in world space coordinates. The vertex buffer object is mapped to host memory and then transfer to an OpenCL buffer.

The trace kernel is executed once per particle giving us a nice performance boost when using a high number of particles. The trace kernel first convert the particle position from world coordinates to a grid position before computing the new position in the same way described in the transport step. The resulting position is then convert back to world coordinates, and transferred to the OpenGL VBO.

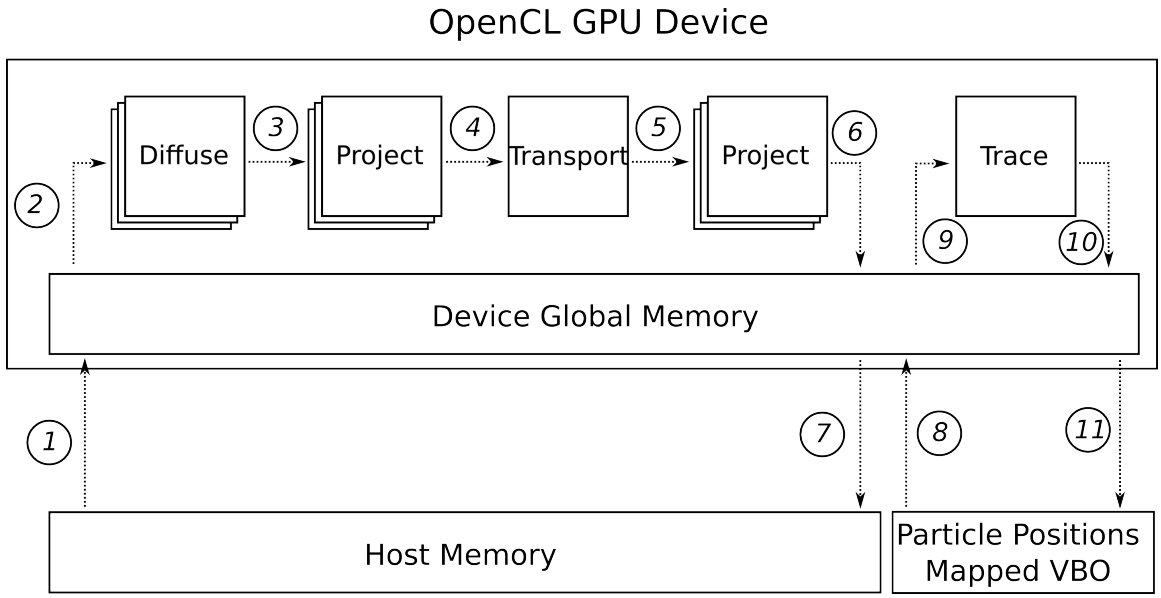

Figure 3: *Project pipeline*

4 Results

In this section, we will compare the performance of the OpenCL implementation with the CPU reference implementation. To analyze the effect of the GPU core count on the fluid parallel pipeline, we used three different CPUs/GPUs combo with grid of different sizes[2]:

- Figure 4: Results obtained with Athlon XP 4200+/GeForce 8800 GTS (96 cores) hardware on the 2.6.32 Linux kernel.
- Figure 5: Results obtained with a laptop Core 2 Duo 2.16GHz/GeForce 8600M (32 cores) hardware on the 2.6.30 Linux kernel.
- Figure 6: Results obtained with a Mac Mini Core 2 Duo 2.26GHz/GeForce 9200M (16 cores) hardware on the Mac OS X 10.6.3.

As we expected, the number of cores is a major factor to enhance the solving speed of the fluid system. With the 96 cores GPU (figure 4), only the 8×8 grid is faster on the CPU and only with a small particle count.

An interesting trend can also be observed (and this independently of the GPU core number), the performance is dropping more slowly on the GPU that on the CPU with an increasing number of computation elements (whether it be an increase grid size or particle count). This trend can probably be explained by the memory bandwidth advantage that the GPU have over his CPU counterpart.

The GPU solution is not perfect though, the setup time to perform a step of the fluid system equation is from being negligible. As we saw in the implementation section, at each time step, the velocity field data need to be transferred from the host memory to the device memory (with a discrete GPU, this means a large transfer on the system bus). For a small grid size and a low particle count, using the GPU is not beneficial since the setup time alone is often greater than the CPU solving time.

[2]The entire results can be seen in the `results.ods` calc sheet

4.1 Possible Optimization

4.1.1 Remove the host transfer for VBO particle tracing

The trace step of our fluid system solving (figure 3) is used to update the position of a vertex buffer object. One weird behavior is that the vertex buffer object probably resides in video memory and that to update it, we mapped it to host memory to transfer the data between the OpenCL and OpenGL objects (the CPU is used to transfer data between two video memory region !). The OpenCL library allow a nice integration with OpenGL where the buffers created through the OpenGL library can be used as is in the OpenCL part. This means that we can build our vertex buffer on the GPU with OpenCL and use the updated buffer with OpenGL without the extraneous CPU copy part and without a second copy of the same data in video memory.

4.1.2 Always stay on the GPU

The second optimization would be to remove the constant penalty suffered at each time step to transfer data from the host to the GPU memory. For now, the velocity field on the host side is only used to display the velocity field through OpenGL immediate mode. We could port the immediate mode part to a vertex buffer and use the same optimization describe in the last point.

4.1.3 Change Gauss-Seidel solver

The Gauss-Seidel method is not really a parallel friendly approach. We used a constant 30 iterations to obtain the resulting approximation, this means that we enqueued 30 parallel executions of $n \times n$ size and this 5 time per step (4 time for the projection and 1 for the diffuse step). Using a method with faster convergence (like the conjugate gradient method) could boost the performance significantly.

4.1.4 Take advantage of the local memory

As we see in figure 2, a small portion of memory is kept locally for each workgroup (16 KByte for the GeForce8/9 family). We could use this local memory as a fast cache to access velocity memory

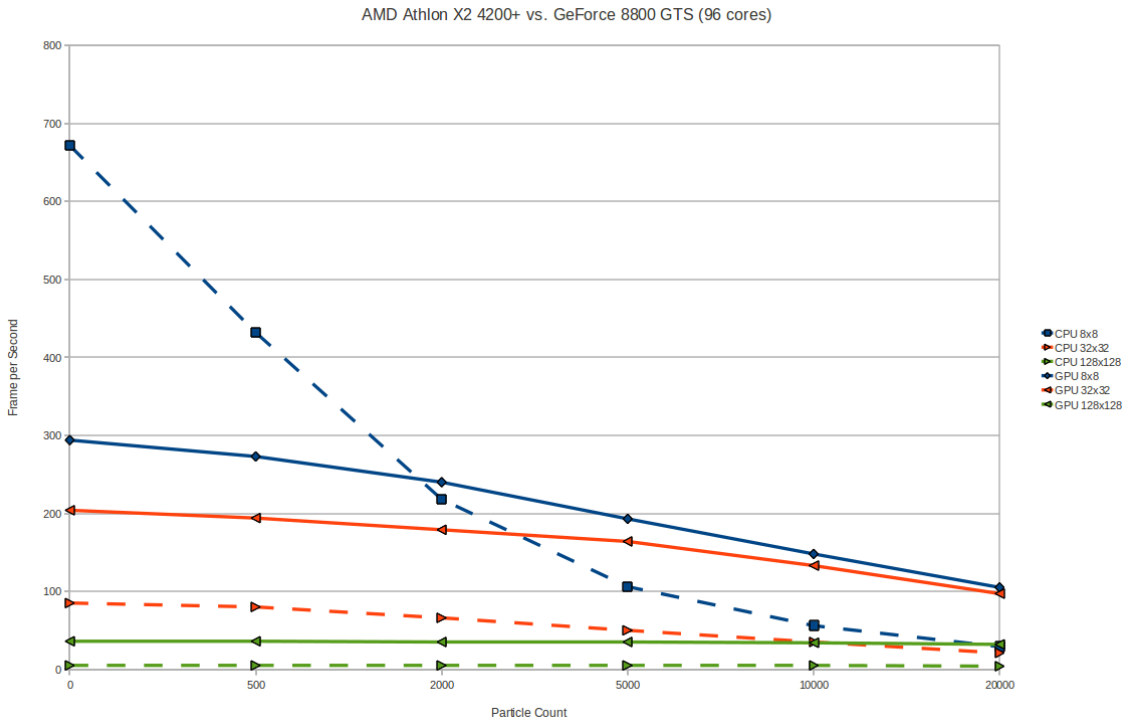

Figure 4: *Athlon XP 4200+ vs. GeForce 8800 GTS (96 cores)*

elements instead of always doing memory fetch from to the device global memory. Since many of our kernels are doing multiple memory access per execution, this could save some bandwidth and accelerate the kernels execution.

4.1.5 Use the Hardware Texture Unit Interpolator

For the transport and trace step, bi-linear interpolation is computed from the velocity field elements. The OpenCL library possess a graphics operations subset and sampler object can be created to access picture element. Using the `CLK_FILTER_LINEAR` filter mode on our 2D velocity array, we could then use the texture unit interpolator instead of our custom made *software* bi-linear interpolation.

5 Conclusion

As we saw in the results section, using the GPU can accelerates significantly the performance of the fluid dynamics solver, especially on large grid size or with a high number of particles. OpenCL provides a clean interface with semantic similar to the OpenGL library, easing the learning process of the library.

Converting sequential algorithm to parallel algorithm offer lot of challenge especially on the optimization side. Even if OpenCL is an hardware independent library, a good knowledge of the underlying hardware help to get the most of the targeted device.

Debugging the compute kernels can also be difficult since there's no debugger that is widely available. NVIDIA is selling a debugging solution, but this is restrain to Microsoft Visual Studio on the Windows platform. It really helps to have a working version on the CPU, not only to compare performance, but also to help in the debugging process. The CPU version can then be used as reference to validate the OpenCL behavior.

References

CORPORATION, N., 2005. Nvidia geforce 8800 gpu architecture overview, November.

FATAHALIAN, K. 2009. From shader code to a teraflop: How shader cores work. In *Siggraph 2009*.

KIRK, D., AND HWU, W.-M. 2010. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann.

OPENCL, K. G., 2009. The opencl specification, October.

STAM, J. 2003. Real-time fluids dynamics for games. In *Game Developers Conference 2003*.

TARIQ, S., AND LLAMAS, I. 2007. Real-time fluids. In *Game Developers Conference 2007*.

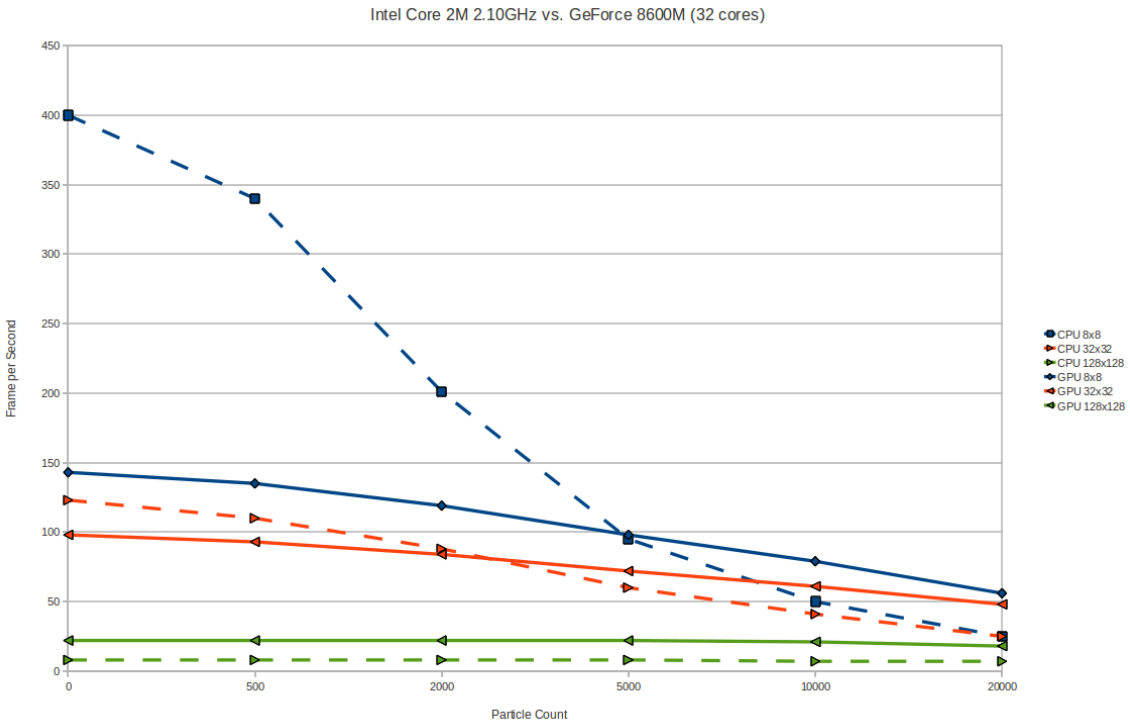

Figure 5: *Intel Core 2 Duo 2.10GHz vs. GeForce 8600M (32 cores)*

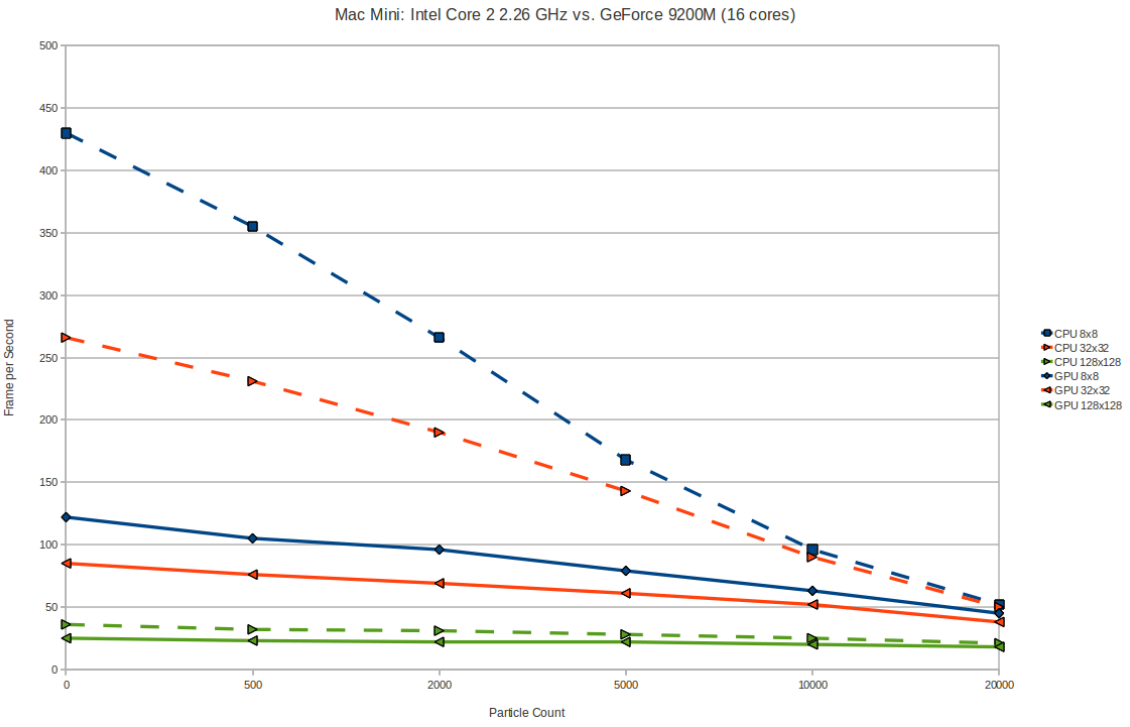

Figure 6: *Mac Mini: Intel Core 2 Duo 2.26GHz vs. GeForce 9200M (16 cores)*