AAS 20-570

GPGPU IMPLEMENTATION OF PINES' SPHERICAL HARMONIC GRAVITY MODEL

J. R. Martin,* and H. Schaub†

Efficient, analytic gravity algorithms are of the upmost importance for astrodynamics research. Such algorithms underlie all satellite simulation software to which researchers and missions have grown increasingly dependent. As these software grow increasingly high-fidelity – accommodating more complex environments, higher density constellations, and more realistic perturbations – there grows a need to revisit their core gravity algorithm to ensure it does not become a bottleneck. Namely as gravity field models continue to improve, simulations will become increasingly burdened by traditional, serial algorithms – requiring astrodynamicists to trade computation speed for model accuracy. In efforts to bypass this tradeoff, this paper investigates a general purpose graphics processing unit (GPGPU) implementation of Pines' spherical harmonic gravity algorithm using Vulkan — an emergent, cross-platform graphics and compute API.

INTRODUCTION

As the age of large, highly-coordinated satellite constellations grows closer to reality, the need for fast, analytic orbit propagation is paramount to efficient satellite simulation and planning. To achieve efficient propagation, however, simulations are often burdened by the fidelity of the gravity model used. With a coarse gravity model, the simulation may run efficiently, but trajectories are only valid over short time scales. Alternatively, with a high-fidelity gravity model, trajectories will become more accurate, but at the cost of slow runtimes – inhibiting larger sensitivity studies or Monte Carlo analysis.

Explicitly, analytic calculation of the gravitational acceleration imparted by a heterogenous mass is a computationally expensive task when using high-fidelity gravity field models. Traditionally this calculation is done by first representing the gravitational potential as a spherical harmonic series expanded to a finite degree l and order m, converting this expansion to non-singular representation, and then taking the gradient to compute the gravitational acceleration. A popular implementation of this process is Pines' formulation.[1]

Despite its popularity within the astrodynamics community, Pines' formulation has a high computational cost – scaling as $\mathcal{O}(l^2)$ where l maximum degree of the gravity model (see Figure 2).[2] For low-fidelity gravity models, this computational inefficiency is negligible, and the ability to propagate orbits for one or many spacecraft is unaffected. However, when scientists and engineers use high-fidelity representations of the gravity field like Earth's EGM2008 model (reaching degree and order

*NSF Graduate Research Fellow, Ann and H.J. Smead Department of Aerospace Engineering Sciences, University of Colorado, Boulder, 431 UCB, Colorado Center for Astrodynamics Research, Boulder, CO, 80309.

†Glenn L. Murphy Chair of Engineering, Ann and H.J. Smead Department of Aerospace Engineering Sciences, University of Colorado, Boulder, 431 UCB, Colorado Center for Astrodynamics Research, Boulder, CO, 80309. AAS Fellow, AIAA Fellow.

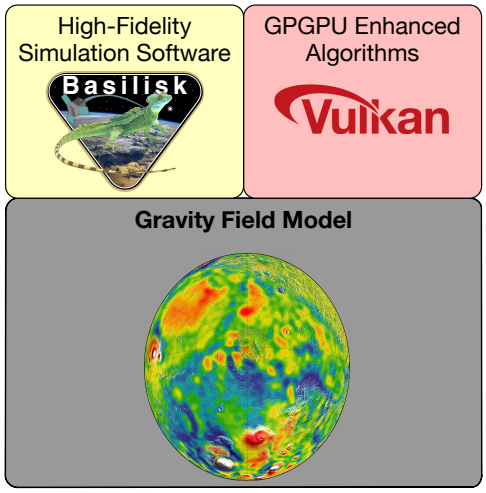


Figure 1: GPGPU enhanced gravity algorithms for high-fidelity astrodynamics software enables support for high accuracy orbit propagation with lower runtimes.

2160) or the Moon's model produced by GRAIL (degree 900) – Pines' formulation can demand millions of computations per timestep to produce the corresponding acceleration.[3,4] Consequently, high-fidelity gravity models impose a large computational bottleneck for astrodynamics simulation – often requiring trajectory designers and researchers to choose either simulation speed or accuracy.

A traditional solution to this computational bottleneck is to use a truncated gravity model – one that provides sufficiently many terms in the spherical harmonic expansion to capture the coarsest gravitational perturbations but few enough to prevent exorbitant amounts of compute time. Such a solution is often acceptable when generating trajectories over short time intervals or for missions that do not require precise orbits. In the case of longer simulations, however, this solution is untenable as the effect of unaccounted gravitational perturbations will accumulate through the dynamics and negatively impact the trajectory (Figure 2). This paper proposes that, in principle, given the state of modern day computing, astrodynamicists need not compromise between simulation speed and accuracy. Explicitly, the computational overhead of Pines' formulation might be significantly reduced if transitioned off of traditional CPUs and onto alternative hardware like Graphics Processing Units (GPUs).

GPUs are designed to solve problems in parallel, unlike CPUs which demand serial execution. As such, if a problem can be properly decomposed into parallelized pieces, GPU algorithms can offer order-of-magnitude performance gains over their serial CPU counterpart.[5] Performance gains of this magnitude have opened up entirely new domains physical modeling within the scientific community* – but their application for gravity modeling remains relatively unsaturated.[6,7]

Kefan et. al. presented a CUDA implementation of a spherical harmonic gravity model claiming positive speed up ratios on the GPU, but fail to provide repeatable or verifiable results.[8] Moreover, the algorithm presented requires vendor specific hardware and does not provide details into the model used. Hupca et. al. demonstrated that inverse spherical harmonic transforms can be evaluated rapidly on multi-core systems or GPUs by but only when evaluating the transform across a 2D grid rather than a single point location.[9] Atallah et. al. propose a GPGPU implementation of the

*https://www.nvidia.com/content/gpu-applications/PDF/gpu-applications-catalog.pdf

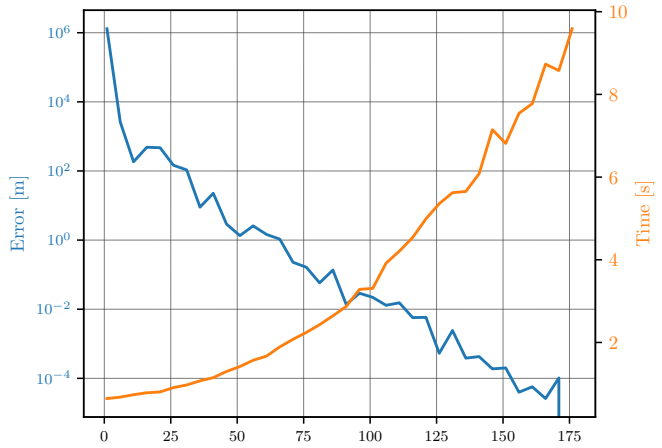


Figure 2: Compute time and final error associated with simulating a spacecraft at 600km altitude orbiting for four hours real-time using Pines' formulation as a function of spherical harmonic degree.

Chebyshev Picard Method which is quick to evaluate, but is only an approximation of the field and only valid over a finite domain.[10] None of these paper provide an explicit, analytic computation of the gravitational force experienced at a single point in a cross-platform, GPGPU compatible manner. This paper attempts to fill this hole by providing an implementation of Pines' formulation on a GPU using Vulkan, a modern GPGPU compute and graphics API.

Vulkan is the only API that is simultaneously cross-platform, officially supported, and does not have hardware specific stipulations [†]. Alternative GPU APIs like CUDA, OpenGL/CL, Metal, and DirectX each fail in at least one of these three categories as of 2020. In addition to its broader applicability, Vulkan is also considered a low-level GPU API providing developers with direct access and control of the GPU, opening opportunities for powerful optimization. Developers are given near-complete control of the graphics and compute pipelines allowing for careful design of command buffers and their dispatch. Coupling these features with traditional tuning of dispatch calls, memory transfer, and thread barriers allows developers to accumulate maximum speed gains. Discovering the optimal permutations of these features and design choices require extensive testing and careful formulations of the underlying algorithm at hand. This paper discusses various ways to decompose Pines' formulation and quantitatively explores how these optimization affect performance.

PINES' FORMULATION

Exploiting GPU hardware to efficiently evaluate Pines' formulation requires refactoring the equations from the original algorithm. As such, it is advantageous to provide the unperturbed algorithm before investigating specific optimization strategies.

Gravitational Potential

Pines' formulation provides an analytic formula that computes the acceleration imparted by a non-homogenous, massive body. This is done by first expressing the potential as a series expansion of spherical harmonics.

[†] https://www.khronos.org/vulkan/

$$U(\boldsymbol{r}) = \frac{\mu}{r} \sum_{l=0}^{\infty} \sum_{m=0}^{l} \left(\frac{1}{r}\right)^l P_{l,m}[\sin(\phi)] \left[C'_{l,m} \cos(m\lambda) + S'_{l,m} \sin(m\lambda)\right] \tag{1}$$

where r is the magnitude of the position vector with respect to the center of mass of the gravitational body, μ is the gravitational parameter of the body, $P_{l,m}$ are the associated legendre polynomials, ϕ is the geodetic latitude, λ is the geodetic longitude, and $C'_{l,m}$ and $S'_{l,m}$ are the Stokes' coefficients. While the potential can technically remain in this form, it is more commonly expressed with non-dimensional coefficients

$$C_{l,m} = \frac{C'_{l,m}}{R_{\text{ref}}^l m} \tag{2}$$

$$S_{l,m} = \frac{S'_{l,m}}{R_{\text{ref}}^l m} \tag{3}$$

where R_{ref} is a chosen reference radius, typically defined as the radius of the sphere which encloses all mass elements of the body (the Brillouin sphere).[11] Simplifying the expression provides

$$U(\boldsymbol{r}) = \frac{\mu}{r} \sum_{l=0}^{\infty} \sum_{m=0}^{l} \left(\frac{R_{\text{ref}}}{r}\right)^l P_{l,m}[\sin(\phi)] \left[C_{l,m} \cos(m\lambda) + S_{l,m} \sin(m\lambda)\right] \tag{4}$$

Again, the equation can remain in this form, however the terms in the series grow exponentially with the degree l. To retain numerical stability, a normalization factor is introduced by Lundberg and Schutz[12]

$$N_{l,m} = \sqrt{\frac{(l-m)!(2-\delta_m)(2l+1)}{(l+m)!}} \tag{5}$$

such that the coefficients and the associated legendre polynomials become

$$\bar{C}_{l,m} = \frac{C_{l,m}}{N_{l,m}} \tag{6}$$

$$\bar{S}_{l,m} = \frac{S_{l,m}}{N_{l,m}} \tag{7}$$

$$\bar{P}_{l,m}[x] = P_{l,m}[x] N_{l,m} \tag{8}$$

altogether providing

$$U(\boldsymbol{r}) = \frac{\mu}{r} \sum_{l=0}^{\infty} \sum_{m=0}^{l} \left(\frac{R_{\text{ref}}}{r}\right)^l \bar{P}_{l,m}[\sin(\phi)] \left[\bar{C}_{l,m} \cos(m\lambda) + \bar{S}_{l,m} \sin(m\lambda)\right] \tag{9}$$

Gravitational Acceleration

To compute the gravitational acceleration, the gradient of Equation (9) must be taken. However, in the case of $\phi = -\frac{\pi}{2}$ or $\frac{\pi}{2}$, the gradient diverges. As such, Pines introduced an alternative formulation which bypasses this numerical instability by changing to dimensionless coordinates within the cartesian coordinate frame where

$$\boldsymbol{r} = r \begin{pmatrix} s \\ t \\ u \end{pmatrix} \qquad \hat{\boldsymbol{\imath}} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \qquad \hat{\boldsymbol{\jmath}} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \qquad \hat{\boldsymbol{k}} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \tag{10}$$

such that

$$r = \sqrt{x^2 + y^2 + z^2} \tag{11}$$
$$s = \frac{x}{r} \tag{12}$$
$$t = \frac{y}{r} \tag{13}$$
$$u = \frac{z}{r} \tag{14}$$

Using these alternative coordinates, the associate Legendre polynomials can be rewritten as

$$P_{l,m}[\sin(\phi)] = P_{l,m}[u] = (1 - u^2)^{\frac{m}{2}} A_{l,m}[u] \tag{15}$$

where

$$A_{l,m}[u] = \frac{d^m}{du^m} P_l[u] = \frac{1}{2^l l!} \frac{d^{l+m}}{du^{l+m}} (u^2 - 1)^l \tag{16}$$

Moreover, if one defines ξ as

$$\xi = \cos(\phi)\cos(\lambda) + j\cos(\phi)\sin(\lambda) = \frac{x}{r} + j\frac{y}{r} = s + jt \tag{17}$$

then

$$\xi^m = \cos^m(\phi) e^{jm\lambda} = (s + jt)^m \tag{18}$$

such that

$$R_m[s,t] = Re\{\xi^m\} \tag{19}$$
$$I_m[s,t] = Im\{\xi^m\} \tag{20}$$

then the potential can be rewritten as

$$U(\boldsymbol{r}) = \frac{\mu}{r} \sum_{l=0}^{\infty} \sum_{m=0}^{l} \left(\frac{R_{\text{ref}}}{r}\right)^l \bar{A}_{l,m}[u]\{\bar{C}_{l,m} R_m[s,t] + \bar{S}_{l,m} I_m[s,t]\} \tag{21}$$

Defining

$$D_{l,m}[s,t] = \bar{C}_{l,m} R_m[s,t] + \bar{S}_{l,m} I_m[s,t] \tag{22}$$
$$\rho_l[r] = \frac{\mu}{r}\left(\frac{R_{\text{ref}}}{r}\right)^l \tag{23}$$

the potential simplifies further to

$$U(\boldsymbol{r}) = \sum_{l=0}^{\infty} \sum_{m=0}^{l} \rho_l[r] \bar{A}_{l,m}[u] D_{l,m}[s,t] \tag{24}$$

where the constituent terms abide by the following recursion relationships

$$R_m[s,t] = sR_{m-1}[s,t] - tI_{m-1}[s,t] \tag{25}$$
$$I_m[s,t] = sI_{m-1}[s,t] + tR_{m-1}[s,t] \tag{26}$$
$$\bar{A}_{l,l}[u] = \sqrt{\frac{(2l+1)(2-\delta_l)}{(2l)(2-\delta_{l-1})}}\bar{A}_{l-1,l-1}[u] \tag{27}$$
$$\bar{A}_{l,l-1}[u] = u\sqrt{\frac{(2l)(2-\delta_{l-1})}{2-\delta_l}}\bar{A}_{l,l}[u] \tag{28}$$
$$\bar{A}_{l,m}[u] = \frac{N_{l,m}}{l-m}((2l-1)uA_{l-1,m}[u] - \tag{29}$$
$$(l+m-1)A_{l-2,m}[u]) \tag{30}$$

with the following initial conditions

$$R_0[s,t] = 1 \tag{31}$$
$$I_0[s,t] = 0 \tag{32}$$
$$\bar{A}_{0,0}[u] = 1 \tag{33}$$

Equation (30) can be further simplified by expanding $N_{l,m}$ for cases where $l \geq (m+2)$, into $N_{1_{l,m}}$ and $N_{2_{l,m}}$ such that

$$\bar{A}_{l,m}[u] = N_{1_{l,m}}uA_{l-1,m}[u] - N_{2_{l,m}}A_{l-2,m}[u] \tag{34}$$

where

$$N_{1_{l,m}} = \sqrt{\frac{(2l+1)(2l-1)}{(l-m)(l+m)}} \tag{35}$$
$$N_{2_{l,m}} = \sqrt{\frac{(l+m-1)(2l+1)(l-m-1)}{(l-m)(l+m)(2l-3)}} \tag{36}$$

To compute the acceleration, the gradient of the potential must be taken with respect to the non-dimensional coordinates u, t, s, and r.

$$\nabla U(\boldsymbol{r}) = \frac{\partial U}{\partial r}\frac{\partial r}{\partial \boldsymbol{r}} + \frac{\partial U}{\partial s}\frac{\partial s}{\partial \boldsymbol{r}} + \frac{\partial U}{\partial t}\frac{\partial t}{\partial \boldsymbol{r}} + \frac{\partial U}{\partial u}\frac{\partial u}{\partial \boldsymbol{r}} \tag{37}$$

$$\frac{\partial r}{\partial \boldsymbol{r}} = \frac{1}{r}\hat{\boldsymbol{r}} \tag{38}$$
$$\frac{\partial s}{\partial \boldsymbol{r}} = \frac{1}{r}\hat{\boldsymbol{\imath}} - \frac{s}{r}\hat{\boldsymbol{r}} \tag{39}$$
$$\frac{\partial t}{\partial \boldsymbol{r}} = \frac{1}{r}\hat{\boldsymbol{\jmath}} - \frac{t}{r}\hat{\boldsymbol{r}} \tag{40}$$
$$\frac{\partial u}{\partial \boldsymbol{r}} = \frac{1}{r}\hat{\boldsymbol{k}} - \frac{u}{r}\hat{\boldsymbol{r}} \tag{41}$$

$$\boldsymbol{g} = \left(\frac{\partial U}{\partial r} - \frac{s}{r}\frac{\partial U}{\partial s} - \frac{t}{r}\frac{\partial U}{\partial t} - \frac{u}{r}\frac{\partial U}{\partial u}\right)\hat{\boldsymbol{r}} + \frac{1}{r}\frac{\partial U}{\partial s}\hat{\boldsymbol{\imath}} + \frac{1}{r}\frac{\partial U}{\partial t}\hat{\boldsymbol{\jmath}} + \frac{1}{r}\frac{\partial U}{\partial u}\hat{\boldsymbol{k}} \tag{42}$$

The partials of the potential can be applied directly to their interior variables

$$\frac{\partial U}{\partial r} = \sum_{l=0}^{\infty}\sum_{m=0}^{l} \frac{\partial \rho_l[r]}{\partial r} \bar{A}_{l,m}[u] D_{l,m}[s,t] \tag{43}$$

$$\frac{\partial U}{\partial u} = \sum_{l=0}^{\infty}\sum_{m=0}^{l} \rho_l[r] \frac{\partial \bar{A}_{l,m}[u]}{\partial u} D_{l,m}[s,t] \tag{44}$$

$$\frac{\partial U}{\partial s} = \sum_{l=0}^{\infty}\sum_{m=0}^{l} \rho_l[r] \bar{A}_{l,m}[u] \frac{\partial D_{l,m}[s,t]}{\partial s} \tag{45}$$

$$\frac{\partial U}{\partial t} = \sum_{l=0}^{\infty}\sum_{m=0}^{l} \rho_l[r] \bar{A}_{l,m}[u] \frac{\partial D_{l,m}[s,t]}{\partial t} \tag{46}$$

$$\tag{47}$$

Evaluating the partials:

$$\frac{\partial \rho_l[r]}{\partial r} = -\frac{(l+1)}{R_{\text{ref}}} \rho_{l+1}[r] \tag{48}$$

$$\frac{\partial \bar{A}_{l,m}[u]}{\partial u} = \frac{N_{l,m}}{N_{l,m+1}} \bar{A}_{l,m+1}[u] \tag{49}$$

$$\frac{\partial D_{l,m}[s,t]}{\partial s} = m(\bar{C}_{l,m} R_{m-1}[s,t] + \bar{S}_{l,m} I_{m-1}[s,t]) \tag{50}$$

$$\frac{\partial D_{l,m}[s,t]}{\partial t} = m(\bar{S}_{l,m} R_{m-1}[s,t] - \bar{C}_{l,m} I_{m-1}[s,t]) \tag{51}$$

Inserting Equations (48) - (51) into Equations (43) - (46)

$$\begin{aligned}
\frac{\partial U}{\partial r} &= \sum_{l=0}^{\infty}\sum_{m=0}^{l} -\frac{(l+1)}{R_{\text{ref}}} \rho_{l+1}[r] \bar{A}_{l,m}[u] D_{l,m}[s,t] \\
\frac{\partial U}{\partial u} &= \sum_{l=0}^{\infty}\sum_{m=0}^{l} \rho_l[r] \frac{N_{l,m}}{N_{l,m+1}} \bar{A}_{l,m+1}[u] D_{l,m}[s,t] \\
\frac{\partial U}{\partial s} &= \sum_{l=0}^{\infty}\sum_{m=0}^{l} \rho_l[r] \bar{A}_{l,m}[u] m(\bar{C}_{l,m} R_{m-1}[s,t] + \bar{S}_{l,m} I_{m-1}[s,t]) \\
\frac{\partial U}{\partial t} &= \sum_{l=0}^{\infty}\sum_{m=0}^{l} \rho_l[r] \bar{A}_{l,m}[u] m(\bar{S}_{l,m} R_{m-1}[s,t] - \bar{C}_{l,m} I_{m-1}[s,t])
\end{aligned} \tag{52}$$

Setting $a_1 = \frac{1}{r}\frac{\partial U}{\partial s}$, $a_2 = \frac{1}{r}\frac{\partial U}{\partial t}$, $a_3 = \frac{1}{r}\frac{\partial U}{\partial u}$, and $a_4 = \left(\frac{\partial U}{\partial r} - \frac{s}{r}\frac{\partial U}{\partial s} - \frac{t}{r}\frac{\partial U}{\partial t} - \frac{u}{r}\frac{\partial U}{\partial u}\right)$, refactoring

$\rho_l[r]$, and simplifying yields

$$a_1[r,s,t,u] = \sum_{l=0}^{\infty}\sum_{m=0}^{l} \frac{\rho_{l+1}[r]}{R_{\text{ref}}} m\bar{A}_{l,m}[u](\bar{C}_{l,m}R_{m-1}[s,t] + \bar{S}_{l,m}I_{m-1}[s,t]) \tag{53}$$

$$a_2[r,s,t,u] = \sum_{l=0}^{\infty}\sum_{m=0}^{l} \frac{\rho_{l+1}[r]}{R_{\text{ref}}} m\bar{A}_{l,m}[u](\bar{S}_{l,m}R_{m-1}[s,t] - \bar{C}_{l,m}I_{m-1}[s,t]) \tag{54}$$

$$a_3[r,s,t,u] = \sum_{l=0}^{\infty}\sum_{m=0}^{l} \frac{\rho_{l+1}[r]}{R_{\text{ref}}} \frac{N_{l,m}}{N_{l,m+1}} \bar{A}_{l,m+1}[u]D_{l,m}[s,t] \tag{55}$$

$$a_4[r,s,t,u] = \sum_{l=0}^{\infty}\sum_{m=0}^{l} \frac{\rho_{l+1}[r]}{R_{\text{ref}}} \frac{N_{l,m}}{N_{l+1,m+1}} \bar{A}_{l+1,m+1}[u]D_{l,m}[s,t] \tag{56}$$

where

$$\frac{N_{l,m}}{N_{l,m+1}} = \sqrt{\frac{(l-m)(2-\delta_m)(l+m+1)}{2-\delta_{m+1}}} \tag{57}$$

$$\frac{N_{l,m}}{N_{l+1,m+1}} = \sqrt{\frac{(l+m+2)(l+m+1)(2l+1)(2-\delta_m)}{(2l+3)(2-\delta_{m+1})}} \tag{58}$$

The final acceleration can then be expressed as:

$$\begin{aligned} \boldsymbol{g} =& (a_1[r,s,t,u] + s \cdot a_4[r,s,t,u])\hat{\boldsymbol{\imath}} + \\ & (a_2[r,s,t,u] + t \cdot a_4[r,s,t,u])\hat{\boldsymbol{\jmath}} + \\ & (a_3[r,s,t,u] + u \cdot a_4[r,s,t,u])\hat{\mathbf{k}} \end{aligned} \tag{59}$$

GPU ALGORITHM

This paper aims to develop an algorithm that evaluates Equation (59) as efficiently as possible on GPU hardware. This requires first understanding the underlying software and hardware of GPUs.

GPGPU Software

Foremost, GPU programs operate on the kernel scale. A kernel is the specific algorithm dispatched to the GPU to be computed asynchronously. Each kernel invocation is assigned a unique thread, an ID, and local memory. These kernel invocations are typically dispatched in work-groups of a fixed, user-defined size of (x, y, z) (`WorkGroupSize`) up to some limit specified by the hardware. Each work-group has a unique shared memory space where kernel invocations within that work-group can exchange data with one another at faster rate than typical global access memory.[13] A GPU program begins execution when the CPU dispatches one or many work-groups – also of user-specified dimensions (x, y, z) (`NumWorkGroups`). As such, if a user defines the `WorkGroupSize` as (16, 8, 4) there will be 512 kernel invocations within that work-group, and if `NumWorkGroups` is defined as (128, 256, 64) for a total of 2,097,152 work-groups, there will ultimately be 1,073,741,824 total kernel invocations sent to the GPU for execution.

GPGPU Hardware

GPU venders like NVIDIA and AMD use similar microarchitectures[‡§¶‖]. Every GPU will have a collection of Streaming Multiprocessors (SM) or Compute Units (CU) (NVIDIA and AMD language respectively) which each house hardware designed to schedule and execute a work-group. The threads (kernel invocations) within a work-group are then divided into batches called warps (NVIDIA) or wavefronts (AMD). A warp is defined as a batch of 32 threads, and a wavefront is defined as a batch of 64 threads. The remainder of this paper will exclusively use AMD language and sizes as the default. Continuing with the earlier example, if a work-group is of size (16, 8, 4), or 512 threads and is sent to a CU, 8 wavefronts will be executed on that CU. By extension, if there are (128, 256, 64) work-groups distributed across 32 CUs, a total of $16,777,216/32 = 524,288$ wavefronts must be executed per CU. This number is still large, however, each CU can also manage multiple wavefronts simultaneously to hide memory latency. I.e. if a single wavefront is waiting from a result in memory, a different wavefront within that work-group can simultaneously run on the momentarily unutilized arithmetic hardware within the same CU. On modern AMD GPUs, up to 40 wavefronts can be scheduled per CU. This ultimately brings the total "tasks" per CU to $524,288/40 \approx 13,108$ – a much more approachable number than the individual 1,073,741,824 invocations that needed to be completed.[14]

GPGPU General Optimization Strategies

GPU optimization is a nuanced endeavour which demands attention to both GPU hardware and software. This subsection presents the optimizations considered in the development of this algorithm thus far, but should not be considered extensive. Further discussion and additional optimization strategies can be found in resources like the CUDA C++ Best Practices Guide, hardware whitepapers, and other online forums.[15,16]

The first consideration for GPU programming is that all threads within each wavefront are executed in lockstep, meaning all threads within the wavefront are expected to perform the same instruction. If there is branching between different threads in the same wavefront, the GPU will halt all threads that do not meet the branch condition and leave them idle until the branch is complete. Consequently, branches impart a performance penalty that grows in proportion to the number of inactive threads and cycles to complete the branch. As such it is strongly recommended to minimize the number of branches in the kernel whenever possible. Common examples of branching include `if-else` statements or conditional `for` loops.

In a similar vein, bank conflicts should be minimized. If multiple kernel invocations need to access to the same shared memory bank, the store and load operations must be executed sequentially. This synchronization imparts a performance penalty as GPU cycles are often much slower than their CPU cycle counterpart. It is therefore recommended that serial work be left to the CPU whenever possible. To avoid the synchronization caused by bank conflicts, the programmer is can ensure that memory accesses per thread are separated by the width of the bank through proper striding and padding.

[‡] https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[§] https://www.amd.com/system/files/documents/rdna-whitepaper.pdf

[¶] https://www.techpowerup.com/gpu-specs/docs/amd-gcn1-architecture.pdf

[‖] https://www.nvidia.com/en-us/data-center/resources/pascal-architecture-whitepaper/

Another optimization strategy is to use Single Instruction Multiple Data (SIMD) operations. Both CPUs and GPUs have specialized instructions that allow for the same operation to be performed on multiple data simultaneously. In the case of a 4D vector, SIMD operations allow all elements to be simultaneously loaded, stored, or operated on. E.g. it costs the same amount of cycles to compute `a=4*6` as it does to compute `a=(1,2,3,4)*(5,6,7,8)`. It is encouraged to maximize the number of SIMD operations per kernel wherever possible.

It is also necessary to maximize occupancy on a GPU. This means ensuring that most, if not all, threads remain active, and all compute units are adequately supplied with many wavefronts. If either of these criteria are not met, the GPU will not be operating at full capacity, and can become slower than if the task were executed on the CPU.

Finally, its important to consider if a kernel is compute bound (algorithm efficiency limited by how many operations need to be performed) or memory bound (algorithm efficiency limited by bottlenecks in memory transfer and overhead). To determine which regime a compute kernel operates within one must first compute the arithmetic intensity of the algorithm, which equates to the number of operations per byte of data transferred to the GPU. If the arithmetic intensity is sufficiently large, the kernel will always be compute bound and the programmer should prioritize minimizing the number of computation cycles. Alternatively, if the arithmetic intensity is low, the programmer should prioritize throughput – optimizing efficient memory load and store requests.[17]

Pines' Formulation Core Routines

To optimize Pines' formulation for a GPU, the algorithm must be broken down into its constituent parts. This paper decomposes the algorithm into two primary routines. Working backwards, the first routine is the computing the double summation for a_1-a_4.

Core Routine 1: Data Reduction Assuming the addends of the series are computed a priori, Equations (53) - (56) simply represent the summation of all terms within a lower-triangle 2D matrix (this assumes the gravity model used is of equal degree and order such that $N = l_{\text{max}} = m$). If that matrix is then flattened into a single 1D array, there exist GPU data reduction techniques that substantially decrease the total number of cycles needed to compute the sum. Explicitly, summing all terms within a lower triangular matrix with a total of $N(N+1)/2$ entries requires $\mathcal{O}(n^2)$ cycles on a CPU. The same reduction algorithm takes as few as $\mathcal{O}(\log_2 n)$ cycles on a GPU by using sequential memory access patterns and shared memory across all threads in a work-group. For brevity, the core routine is expressed in Algorithm 1 and visualized in Figure 3. Additional optimization techniques exist beyond those expressed in Algorithm 1 like loop unrolling and removing instruction overhead, though a deeper discussion of such techniques is left to the many resources available online**.

Core Routine 2: Legendre Matrix The more challenging routine to put on a GPU is computing the series addends prior to data reduction. Redefining the addends from Equations (53) - (56) as

**http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

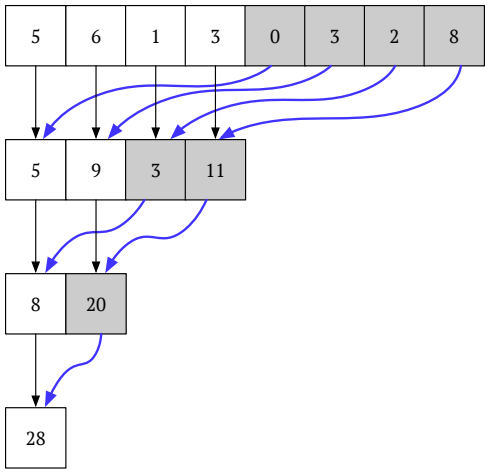


Figure 3: Data Reduction Technique

Algorithm 1 Data Reduction Loop

```
localIdx = LocalInvocationIndex
finalValue[localIdx] = a_{i_{l,m}}
barrier()
for s = WorkGroupSize.x/2; s > 0; s >>= 1) do
    if (localIdx < s) then
        finalValue[localIdx] += finalValue[localIdx + s]
    end if
    barrier()
end for
```

$a_{i_{l,m}}$ and expanding yields:

$$a_{1_{l,m}}[r,s,t,u] = \frac{\rho_{l+1}[r]}{R_{\text{ref}}} m\bar{A}_{l,m}[u](\bar{C}_{l,m}R_{m-1}[s,t] + \bar{S}_{l,m}I_{m-1}[s,t]) \tag{60}$$

$$a_{2_{l,m}}[r,s,t,u] = \frac{\rho_{l+1}[r]}{R_{\text{ref}}} m\bar{A}_{l,m}[u](\bar{S}_{l,m}R_{m-1}[s,t] - \bar{C}_{l,m}I_{m-1}[s,t]) \tag{61}$$

$$a_{3_{l,m}}[r,s,t,u] = \frac{\rho_{l+1}[r]}{R_{\text{ref}}} \frac{N_{l,m}}{N_{l,m+1}} \bar{A}_{l,m+1}[u](\bar{C}_{l,m}R_{m}[s,t] + \bar{S}_{l,m}I_{m}[s,t]) \tag{62}$$

$$a_{4_{l,m}}[r,s,t,u] = \frac{\rho_{l+1}[r]}{R_{\text{ref}}} \frac{N_{l,m}}{N_{l+1,m+1}} \bar{A}_{l+1,m+1}[u](\bar{C}_{l,m}R_{m}[s,t] + \bar{S}_{l,m}I_{m}[s,t]) \tag{63}$$

These expressions are not immediately amenable for GPU optimization as they are idiosyncratic, each with different variables and indices. The expressions can be homogenized however by defining

the following constants:

$$Q = \frac{\rho_{l+1}[r]}{R_{\text{ref}}} \tag{64}$$

$$c_1 = \frac{N_{l,m}}{N_{l,m+1}} \tag{65}$$

$$c_2 = \frac{N_{l,m}}{N_{l+1,m+1}} \tag{66}$$

$$d_1 = s\bar{C}_{l,m} + t\bar{S}_{l,m} \tag{67}$$

$$d_2 = s\bar{S}_{l,m} - t\bar{C}_{l,m} \tag{68}$$

$$\tag{69}$$

and expanding R_m and I_m in terms of R_{m-1} and I_{m-1}. Simplifying, the expressions for all $a_{i_{l,m}}$ become significantly more consistent:

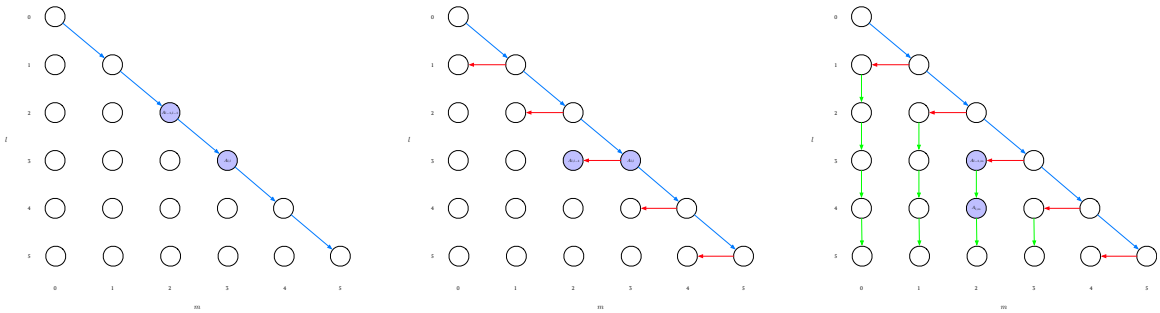
$$\begin{bmatrix} a_{1_{l,m}} \\ a_{2_{l,m}} \\ a_{3_{l,m}} \\ a_{4_{l,m}} \end{bmatrix} = \begin{bmatrix} Q \\ Q \\ Q \\ Q \end{bmatrix} \begin{bmatrix} m \\ m \\ c_1 \\ c_2 \end{bmatrix} \begin{bmatrix} A_{l,m}[u] \\ A_{l,m}[u] \\ A_{l,m+1}[u] \\ A_{l+1,m+1}[u] \end{bmatrix} \left(\begin{bmatrix} \bar{C}_{l,m} \\ \bar{S}_{l,m} \\ d_1 \\ d_1 \end{bmatrix} \begin{bmatrix} R_{m-1} \\ R_{m-1} \\ R_{m-1} \\ R_{m-1} \end{bmatrix} + \begin{bmatrix} \bar{S}_{l,m} \\ -\bar{C}_{l,m} \\ d_2 \\ d_2 \end{bmatrix} \begin{bmatrix} I_{m-1} \\ I_{m-1} \\ I_{m-1} \\ I_{m-1} \end{bmatrix} \right) \tag{70}$$

Equation 70 is advantageous for GPU computing for two reason. The first is SIMD compatibility. The expressions for $a_{1_{l,m}} - a_{4_{l,m}}$ share many common terms which can be loaded into memory simultaneously via SIMD operations. Moreover, each addend uses the same the same six multiplications and one addition – common instructions that can be shared across all expressions. By converting the equations into vectorized form, the shared multiply and addition instructions alone reduce the number of cycles needed to compute all addends by an additional factor of four.

The second advantage of Equation (70) is that $Q, c_1, c_2, d_1, d_2, \bar{C}_{l,m}, \bar{S}_{l,m}$ are all independent variables that can be computed without any knowledge of prior operations within the algorithm. This means there is no recursion or unique branch criteria to calculate their values, and thereby no synchronization is imposed on the GPU for these variables.

Beyond these terms, there is the challenge of computing the remaining variables $A_{l,m}$, $A_{l,m+1}$, $A_{l+1,m+1}$, R_{m-1}, I_{m-1} which *do* require knowledge of prior terms due to their recursion formulas. Two possible solutions exist for evaluating these terms. The first is a memory-centric approach where these terms are only computed once, stored in memory, and then loaded back into registers as needed. The alternative is a compute-centric approach where the values are computed as needed. The former demands more memory operations and fewer arithmetic operations while the later requires less memory but at the cost of redundant calculations.

Memory Bound In the memory bound case, it is assumed that all terms within Equation (70) are computed individually and saved to local memory on the GPU. This requires careful sequencing to satisfy the recursion relationships of $A_{l,m}$, R_m, and I_m. There is little advantage to solving R_m and I_m on the GPU as their formula have no elements of parallelization. As such those terms are solved on the CPU and transferred upon dispatch to the GPU. The $A_{l,m}$ terms, however, have a mix of strict, serial recursion but also opportunities for parallelization. Specifically Eqs. (27) is explicitly recursive as seen in Figure 4a, however Equations (28) and (30) (seen in Figures 4b and 4c) can be evaluated asynchronously across the columns. Each kernel invocation can be assigned to a different column to be computed independently. Despite this asynchronous opportunity, such



(a) First CPU Command (b) Second CPU Command (c) Third CPU Command

Figure 4: CPU Data Flow

approach suffers from loop divergence. With every cycle another thread / column will have been completed and the thread within the wavefront will go idle. Nevertheless the order of computation to the kernel invocation scales as $\mathcal{O}(n)$ rather than the entire $\mathcal{O}(n^2)$ algorithm necessary to compute the same terms on a CPU.

Beyond the asynchronous challenges of this strategy, there are also hardware limits to consider – namely available VRAM on the GPU. Assuming $N_{S/C}$ total spacecraft are being simulated on the GPU, additional memory will need to be allocated. Table 1 show the total number of buffers that must be stored on the GPU as well as their size. Constraining these parameters to the available memory limits of the hardware, Figure 5 shows the maximum number of spacecraft that can be simultaneously simulated on a GPU given a chosen spherical harmonic model fidelity.

Table 1: The buffers transferred to the GPU across algorithm lifetime

Struct Name	Variables	Type	Structs per Buffer	Buffers Per Sim
N_{Params}	N_1, N_2, N_{q1}, N_{q2}	float	$(l+1)(l+2)/2$	1
A	$a_{l,m}$	float	$(l+1)(l+2)/2$	$N_{S/C}$
Coef	$C_{l,m}, S_{l,m}$	float	$(l+1)(l+2)/2$	1
Misc	μ, R_0	float	1	1
	l_{max}	int	1	1
Location	r, u	float	1	$N_{S/C}$
Acceleration	a_1, a_2, a_3, a_4	float	$(l+1)(l+2)/2$	$N_{S/C}$
Euler	R_m, I_m	float	$(l+1)$	$N_{S/C}$

Algorithm 2 Legendre-Matrix Memory Bound Algorithm

```
Define l as unique thread ID
Compute lower diagonal, A_{l,l-1}
Thread barrier
for m = l, 1 ... N + 2 do
    Recursively solve for A_{l,m}
end for
```

Compute Bound The alternative formulation to the memory bound approach centers on computing all variables on the GPU directly on an as needed basis. This prevents shaders from needing to interface with global memory for which write and read operations are particularly slow. This compute bound method leans on the fact that even in the memory bound case, there is still divergence

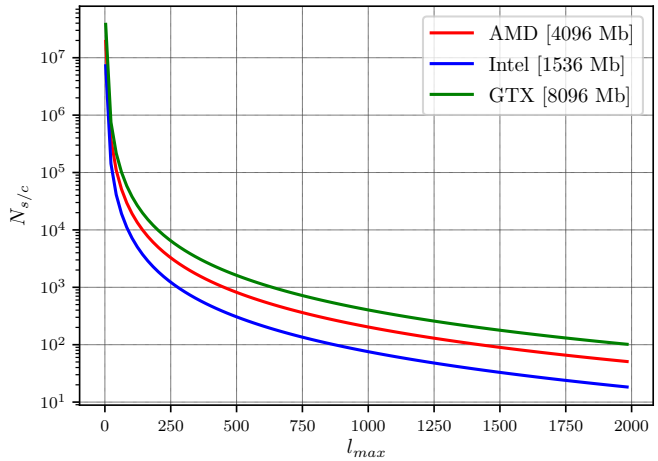


Figure 5: Video memory consumed as a function of l_{max} and number of spacecraft, $N_{S/C}$

at the column level such that the algorithm complexity will always be $\mathcal{O}(n)$. This is also true of the compute bound case exhibited in Figure 6 which uses $m + 2$ computations for traversing the diagonal, 2 computations to reach the off-diagonal terms, and $2(l - m - 1)$ computations to acquire the necessary values in the $A_{l,m}$ matrix – yielding the same $\mathcal{O}(n)$ algorithm without the need for global memory access.

The added benefit of the compute bound approach is that there is not any functional limit to the total number of spacecraft that can be simulated at once. The only data that needs to be transferred to the GPU are the Stokes' coefficients ($4 * l(l + 1)$ bytes sent once), the normalization parameters ($16 * l^2$ bytes sent once) and the location of each spacecraft ($16 * N_{S/C}$ bytes sent at each timestep). The disadvantage of using the compute bound approach is the redundant computation of intermediate terms. Each shader invocation will always need to traverse the diagonal of the matrix, and compute intermediate $A_{i,j}$ on their way to the $A_{l,m}$. Despite this disadvantage, the cumulative advantages outweigh the cost of redundancy, so the compute bound approach is ultimately prioritized in subsequent discussion and benchmarking.

Scalability to Constellations

In both the compute and memory bound approaches, there is no clear way to circumvent thread divergence when computing the Legendre matrix in the single spacecraft case. Some indices within the matrix will inevitably require more cycles to compute than their adjacent terms due to the recursion. This ultimately compromises the lockstep nature of the wavefronts and incurs a performance penalty. Despite this challenge for the single spacecraft case, there is a workaround when simulating multiple spacecraft with the same kernel. Namely, if the Legendre matrix of each spacecraft is stacked along the z work-group dimension, and the local size of the z dimension in each work-group is sufficiently large, each wavefront can be assigned a specific index, l, m, such that all threads within that wavefront execute the exact same instructions with no divergence. Such approach requires sufficiently many satellite in the simulation to maximize occupancy on the GPU, but should avoid the divergence penalty.

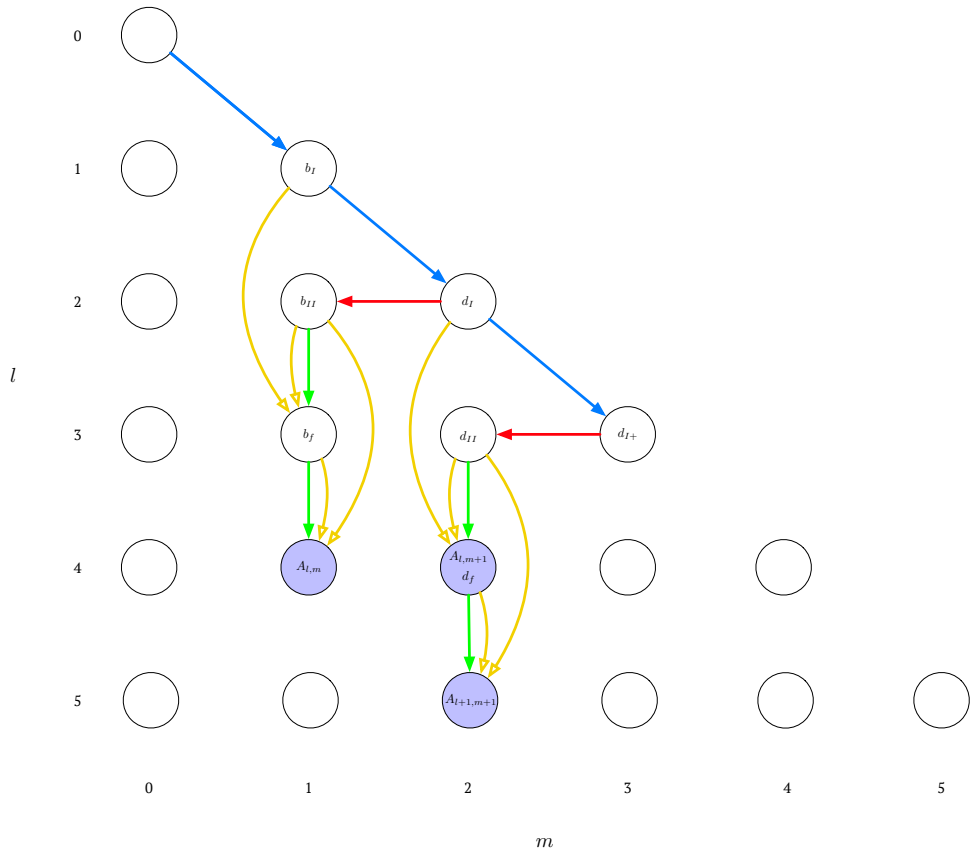
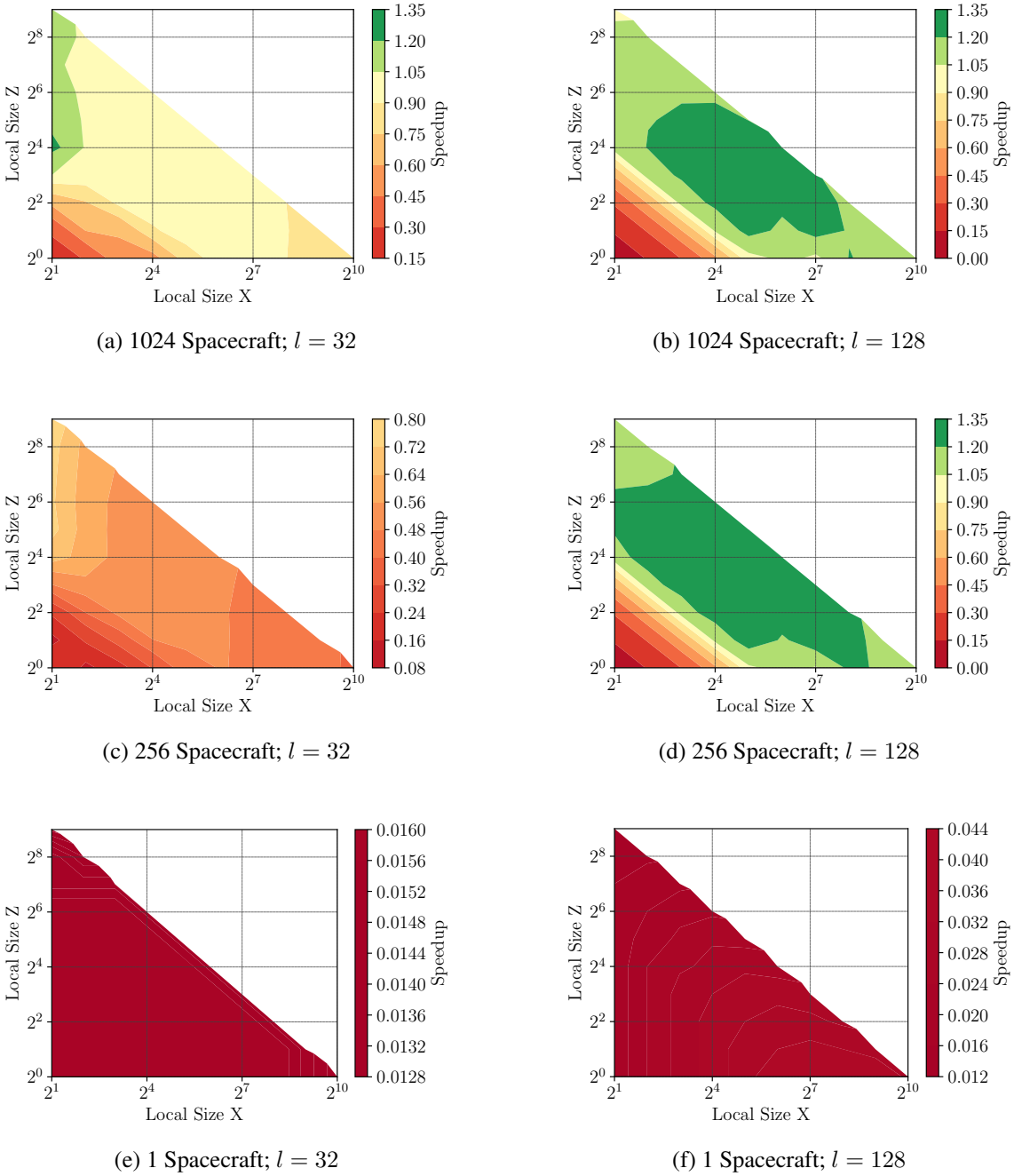


Figure 6: Data Flow of GPU Kernel Invocation

BENCHMARKS

Optimization of this algorithm is ongoing; however preliminary results and benchmarking methodology are presented to demonstrate how this work is being verified. Foremost this algorithm is tested on two different GPUs. The first is the Intel HD Graphics 630 with 1536 MB of VRAM, the second is a AMD Radeon Pro 560 with 4096 MB of VRAM. The former is a consumer grade integrated GPU available on modern CPU processors, while the latter is a more performant discrete graphics card at the high-end of the consumer spectrum. Both GPUs are run on a 15" 2017 Macbook Pro with a 3.1 GHz Quad-Core Intel Core i7 CPU with 16 Gb of 2133 MHz LPDDR3 RAM. The performance of the GPGPU implementation is tested by varying the number of spacecraft simulated simultaneously as well as changing the dimensions of the work-group size by factors of 2. The maximum number of kernel invocations that can be spawned on the Intel integrated graphics card is 256 whereas the AMD card allows for as many as 1024. As such the dimensions of the work-group must always multiply such that they remain less than or equal to 256 or 1024 respectively.

Such benchmarks are important to coarsely characterize GPU occupancy and thread divergence. As discussed, the Legendre matrix computation has unavoidable thread divergence along the columns, but otherwise has coalesced memory access and shares many of the same instructions for intermediate computations. Therefore, when decreasing the work-group size x-dimension (which corresponds to how many elements of $A_{l,m}$ are evaluated in the work-group) thread divergence is reduced as fewer columns are seen (performance gain), but the number of shared intermediate instructions



(a) 1024 Spacecraft; $l = 32$

(b) 1024 Spacecraft; $l = 128$

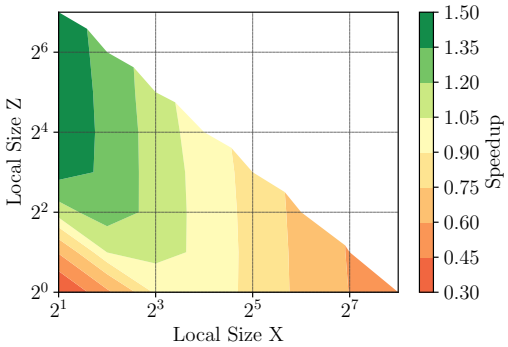
(c) 256 Spacecraft; $l = 32$

(d) 256 Spacecraft; $l = 128$

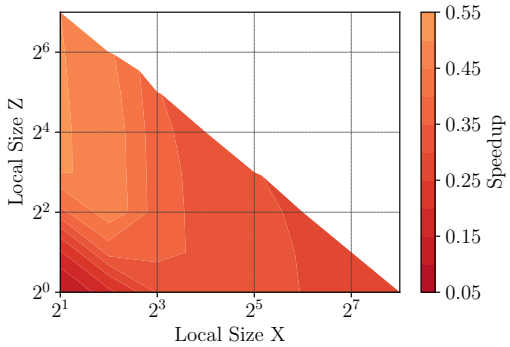
(e) 1 Spacecraft; $l = 32$

(f) 1 Spacecraft; $l = 128$

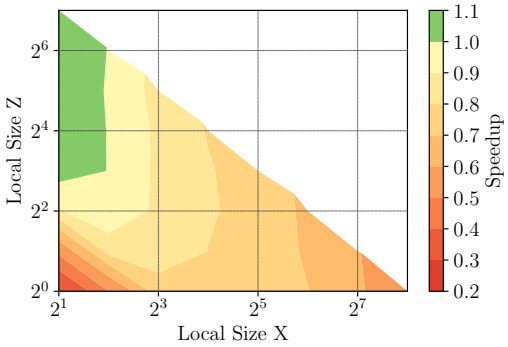
Figure 7: Current speed up ratios for the Legendre matrix computation on AMD Radeon Pro 560 GPU using the compute bound method.



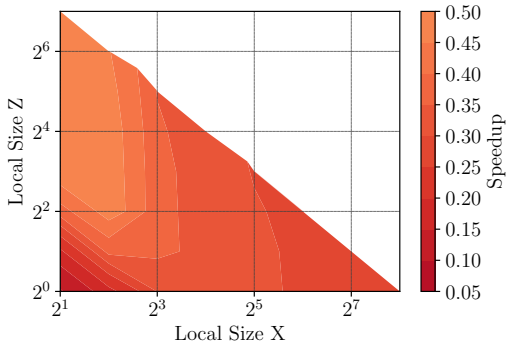
(a) 1024 Spacecraft; $l = 32$



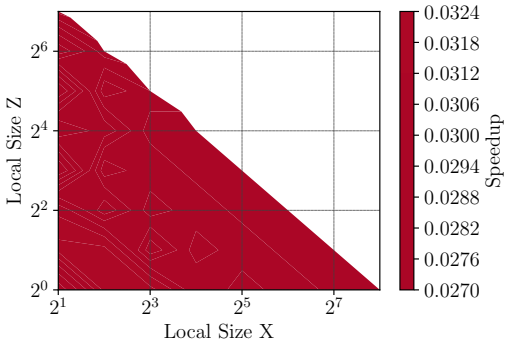
(b) 1024 Spacecraft; $l = 128$



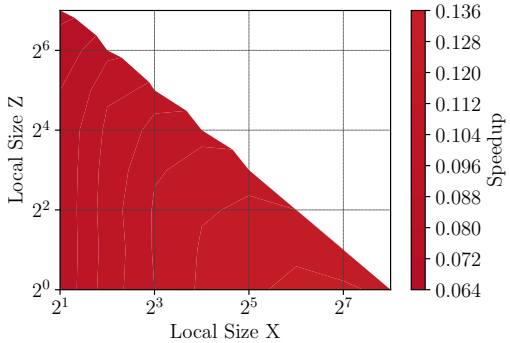
(c) 256 Spacecraft; $l = 32$



(d) 256 Spacecraft; $l = 128$



(e) 1 Spacecraft; $l = 32$



(f) 1 Spacecraft; $l = 128$

Figure 8: Current speed up ratios for the Legendre matrix computation on Intel HD Graphics 630 GPU using the compute bound method.

among the work-group decrease (performance penalty). By extension, when varying the local group size in z-dimension (corresponding to the number of simultaneously computed Legendre matrices), these tradeoffs grow more exaggerated though the general wavefront efficiency should increase for reasons mentioned in the prior section. By searching across local work-group size permutations, empirically optimal work-group dimensions can be found which maximize shared instructions and occupancy while minimizing thread divergence.

The current results for the Legendre matrix calculation are presented as speed-up ratios. These ratios are measured by averaging the time taken to submit and complete the Legendre matrix calculation on the GPU during a Basilisk scenario. The Basilisk scenario simulated thirty minutes of real-time orbit propagation, stepping with 10 second intervals, and using an RK4 integrator. The same simulation was performed on a CPU and the time to compute the Legendre matrix was averaged. The corresponding speed-up ratio is defined as the ratio between these two time average:

$$\tau = \frac{t_{\text{CPU}}}{t_{\text{GPU}}} \tag{71}$$

The speed-up ratios for the AMD GPU are found in Figure 7 and for the Intel GPU in Figure 8.

Currently the GPU Legendre matrix algorithm demonstrates consistent > 1 speed-up ratios on the AMD card when there are sufficiently many spacecraft (> 256) operating in a sufficiently a high-fidelity gravity field ($l = 128$). While speed-up ratios were occasionally realized in the lower-fidelity case ($l = 32$), the ratios were consistently less performant than their high-fidelity counterpart. It is assumed that this is due to the relatively small dimension of the Legendre matrix in the low fidelity case. The computational overhead to transition the initial memory onto the GPU, dispatch threads, and return a result is sufficiently high on discrete GPUs such that there exists a minimum degree model that must be used before measurable speed-up ratios can be achieved.

The narrative differs when looking at the performance on the Intel GPU in Figure 8. Here there are no GPU performance gains when using the high-fidelity models, but there appreciable speed-ups (as high as 40%) when using the lower-fidelity models. This appears to be an artifact of the shared DRAM of the integrated graphics card with the CPU. Discrete GPUs like the AMD card require that data be transferred from the CPU to the GPU explicitly. This is often an expensive process and best performed in a single, large data transfer. Integrated graphics processors do not have this limitation and can share memory share memory directly with the CPU resulting in much faster transfers. Consequently, the overhead for dispatching work to the Intel card is lower than the overhead associated with the discrete card, ultimately allowing observable speed-up ratios in the low-fidelity gravity field. Despite this, the integrated graphics card does suffer when evaluating high-fidelity models due to its lower clock-rate. This is seen through the narrowness of the speed-up window in Figures 8a and 8c. Given that the highest performance is observed when the work-group x-dimension is particularly small, the thread divergence of solving multiple columns in the Legendre matrix is significantly more costly to the integrated card than the discrete card.

Together, the two GPUs offer unique advantages that span most operational conditions. The discrete graphics card is able to model high-fidelity gravity fields with measurable speed-ups, and the integrated card is able to model low-fidelity gravity fields with measurable speed-ups. The one condition for which neither GPU succeeds is in the case of too few spacecraft. When modeling a single spacecraft in either a low- or high-fidelity gravity field, it is always faster to model on a CPU. Heuristically this is intuitive – despite the integrated card having a lower overhead cost than the discrete card, an overhead still exists. The dimensions of the single satellite problem are sufficiently

small that the overhead is always too large, regardless of GPU type.

CONCLUSION

This paper provides a discussion of general GPGPU best practices and attempts implementation of such practices to construct an alternative form of Pines algorithm to compute the accelerations generated by high-fidelity gravity fields. Such implementation decreases the algorithmic complexity from $\mathcal{O}(n^2)$ on a CPU into $\mathcal{O}(n)$ for the Legendre matrix computation and to $\mathcal{O}(\log_2 n)$ for the data reduction computation on the GPU. Moreover the GPGPU implementation that has no functional limitations on GPU memory making it advantageous for modeling large satellite constellations. While implementation efforts are still ongoing, the intermediate results do match heuristic expectation and fully-realized speed-up ratios are expected in the near future.

ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. 2040434.

REFERENCES

[1] S. Pines, "Uniform Representation of the Gravitational Potential and its Derivatives," *AIAA Journal*, vol. 11, no. 11, pp. 1508–1511, 1973.

[2] B. A. Jones, *Efficient Models for the Evaluation and Estimation of the Gravity Field*. PhD thesis, CU Boulder, 2010.

[3] N. K. Pavlis, S. A. Holmes, S. C. Kenyon, D. Schmidt, and R. Trimmer, "A preliminary gravitational model to degree 2160," *International Association of Geodesy Symposia*, vol. 129, pp. 18–23, 2005.

[4] F. G. Lemoine, S. Goossens, T. J. Sabaka, J. B. Nicholas, E. Mazarico, D. D. Rowlands, B. D. Loomis, D. S. Chinn, G. A. Neumann, D. E. Smith, and M. T. Zuber, "GRGM900C: A degree 900 lunar gravity model from GRAIL primary and extended mission data," *Geophysical Research Letters*, vol. 41, no. 10, pp. 3382–3389, 2014.

[5] V. Volkov, "Understanding Latency Hiding on GPUs — EECS at UC Berkeley," tech. rep., UC Berkeley Electrical Engineering and Computer Sciences, Berekely, CA, 2016.

[6] P. W. Kenneally, *Faster than Real-Time GPGPU Radiation Pressure Modeling Methods*. Ph.d. thesis, University of Colorado at Boulder, 2019.

[7] D. Negrut, A. Tasora, M. Anitescu, H. Mazhar, T. Heyn, and A. Pazouki, *Chapter 20 - Solving Large Multibody Dynamics Problems on the GPU*. No. Dvi, Elsevier Inc., 2012.

[8] W. Kefan and L. Ge, "The gravity parallel computation based on GPU," *2017 3rd IEEE International Conference on Computer and Communications, ICCC 2017*, vol. 2018-Janua, pp. 2409–2413, 2018.

[9] I. O. Hupca, J. Falcou, L. Grigori, and R. Stompor, "Spherical harmonic transform with GPUs," vol. 7155 LNCS, no. PART 1, pp. 355–366, 2012.

[10] A. Atallah and A. Bani Younes, "Parallel Chebyshev Picard Method," in *AIAA SciTech Forum*, no. January, (Orlando, FL), AIAA, 2020.

[11] M. Brillouin, "Équations aux dérivées partielles du 2e ordre. Domaines à connexion multiple. Fonctions sphériques non antipodes," vol. 4, pp. 173–206, 1933.

[12] J. B. Lundberg and B. E. Schutz, "Recursion formulas of Legendre functions for use with nonsingular geopotential models," *Journal of Guidance, Control, and Dynamics*, vol. 11, no. 1, pp. 31–38, 1988.

[13] R. Kessenich, John; Baldwin, Dave; Ros, "The OpenGL Shading Language Version 4.60.7," tech. rep., 2019.

[14] M. Mantor and M. Houston, "AMD Graphics Core Next," *AMD Fusion Developer Summit*, 2011.

[15] NVIDIA, "CUDA C Best Practices Guide (v11.0)," Tech. Rep. July, 2020.

[16] S. Jones, "Cuda Optimization Tips, Tricks and Techniques," in *GPU Technology Conference*, (Sillicon Valley), NVIDIA, 2017.

[17] A. Ilic, F. Pratas, and L. Sousa, "Cache-aware roofline model: Upgrading the loft," *IEEE Computer Architecture Letters*, vol. 13, no. 1, pp. 21–24, 2014.