Cracks in the Sky: Abelian-Higgs Cosmic String Evolution with CUDA

J. R. C. C. Correia\textsuperscript{1,2,3}\textsuperscript{*} and C. J. A. P. Martins\textsuperscript{1,2,\dagger}

\textsuperscript{1}Centro de Astrofísica, Universidade do Porto, Rua das Estrelas, 4150-762 Porto, Portugal
\textsuperscript{2}Instituto de Astrofísica e Ciências do Espaço, Universidade do Porto, Rua das Estrelas, 4150-762 Porto, Portugal
\textsuperscript{3}Faculdade de Ciências, Universidade do Porto, Rua do Campo Alegre 687, 4169-007 Porto, Portugal

(Dated: 17 August 2018)

Topological defects form at cosmological phase transitions by the Kibble mechanism, with cosmic strings and superstrings having the most interesting phenomenology. A rigorous analysis of their astrophysical consequences is limited by the availability of accurate numerical simulations, and therefore by hardware resources and computation time. Improving the speed and efficiency of existing codes is therefore essential. All current cosmic string simulations were performed on Central Processing Units. In previous work we presented a General Purpose Graphics Processing Unit implementation of the evolution of cosmological domain wall networks. Here we continue this paradigm shift and discuss an analogous implementation for local Abelian-Higgs strings networks. We discuss the implementation algorithm (including the discretization used and how to calculate network averaged quantities) and then showcase its performance and current bottlenecks. We validate the code by directly comparing our results for the canonical scaling properties of the networks in the radiation and matter eras with those in the literature, finding very good agreement. We finally highlight possible directions for improving the scalability of the code.

INTRODUCTION

A generic cosmological prediction of many theories beyond the Standard Model is the formation of objects known as topological defects, by means of the Kibble mechanism \cite{Kibble1976}. Since properties of these objects and their astrophysical consequences are intrinsically linked to the symmetry breaking patterns which produce them, one can think of them as fossil relics of the physical conditions in the early Universe.

Recent constraints on these objects using cosmic microwave background and gravitational wave data \cite{Planck2013, Graham2017} are mainly limited by the existence of accurate high-resolution simulations of defect networks with a large dynamic range, as well as full sky maps of the backgrounds produced by these networks; the approximations currently being used to mitigate the absence of such data clearly introduce systematic uncertainties that are comparable to the quoted statistical uncertainties. This problem is even more severe for next-generation facilities such as CORE \cite{CORE} or LISA \cite{LISA}. On the other hand, analytic studies of realistic defect networks (with more degrees of freedom) will too be constrained by the lack of high-resolution simulations. Resolving this issue by traditional means would imply unrealistic hardware/compute time needs.

To alleviate this problem, one can attempt to exploit differing hardware architectures with the onus of optimization falling to the developers of the tool in question. In the literature there are several examples of defect simulations optimized for Central Processing Units (CPUs), either assuming shared or distributed memory architectures. Examples of Goto-Nambu cosmic string simulations can be found at \cite{Goto2004, Nambu2010}, while examples of field theory simulations of the two simplest types of defect are the WAVES code \cite{Waves} for domain walls (also optimized for Intel Xeon Phi co-processors \cite{Intel}) and the cosmic string evolution codes of \cite{SCHWARZ2012}. Other more complex examples of field theory defect simulations also include monopoles \cite{Monopoles}, semilocal strings \cite{Semilocal}, dual-higgsed strings \cite{Dualhiggsed, DualhiggsedC} and hybrid defect simulations \cite{Hybrid, Hybrid2}.

Simulations which use Graphics Processing Units are far more scarce, with the only known instance being \cite{CUDA} by the authors. This paper is a continuation of the same paradigm shift of our previous study, which seeks to simulate local Abelian-Higgs strings. We first introduce the algorithm used to simulate field theory cosmic strings (including the discretization used and how to calculate network averaged quantities), then showcase the performance of the implementation, and finally validate it by directly comparing with results found on the literature. We conclude by highlighting tentative directions to augment the scalability of this code.

DISCRETIZATION SCHEME

A $U(1)$ local Abelian-Higgs string corresponds to a topological soliton that arises as a solution to the equations of motion of the Lagrangian

\[
\mathcal{L} = |D_\mu \phi|^2 - \frac{1}{4} F_{\mu\nu} F^{\mu\nu} - \frac{\lambda}{4} (|\phi|^2 - \sigma^2)^2 ,
\]

where $\phi$ is a complex scalar field, $F_{\mu\nu} = \partial_\mu A_\nu - \partial_\nu A_\mu$ a gauge field strength ($A_\mu$ corresponds to the gauge field), $D_\mu = \partial_\mu - ie A_\mu$ indicates a covariant derivative and $\lambda$ and $e$ are two constants which set the values of the scalar and vector masses $m_\sigma = \sqrt{2\lambda}$ and $m_e = e\sigma$.

We follow the same discretization procedure as \cite{SCHWARZ2012},

\[
\]
which requires first writing the discrete Lagrangian,
\[
\mathcal{L} = \frac{1}{2e^2(\eta)a^2(\eta)}\Delta x^4 \left( \sum_i (E_i^{x,\eta})^2 \right) - \frac{1}{2} \sum_i \sum_j \left[ 1 - \cos(\Xi_{ij}) \right] + \| \Pi^{x,\eta} \|^2
\]
\[
- \frac{1}{\Delta x^2} \sum_i (e^{-iA_i^{x,\eta}} - \phi^{x+k_i,\eta} - \phi^{x,\eta})^2 - \frac{1}{4} a^2(\eta) \lambda(\eta)(|\phi^{x,\eta}|^2 - \sigma^2)^2
\]
(2)
where \( \phi \) and \( A \) represent the complex scalar field at conformal time-step \( \eta \) and \( \Pi \) and \( \epsilon \) are conjugates of the aforementioned fields at half-steps. Note that vector fields are rescaled as \( A_i^x \rightarrow e^\Delta x A_i^x \) and \( E_i^x \rightarrow e^\Delta \eta E_i^x \). Both scalar fields reside at lattice sites \( i, j, k \rightarrow x \), and the vector fields at half-sites (for convenience however site \( A_i^{x+1/2k_i} \), where \( k_i \) is some unit vector, is written \( A_i^x \)). Note that the gauge field strength \( F_{ij} \) is written as \( \Xi_{ij} \). There is a subtlety in most field theory defect simulations: since the physical defect width is constant, the true equations of motion result in strings that eventually in comoving coordinates it shrinks. This means that the scale factor, a positive constant and the gauge coupling at times \( \eta \), respectively. \( \Delta x \) and \( \Delta \eta \) are the lattice spacing and time-step value.

Through variational principles, one can write the equations of motion as a staggered leap-frog scheme
\[
\Pi_i^{x,\eta+1/2} = \left( \frac{a_{\eta-1/2}}{a_{\eta+1/2}} \right)^2 \Pi_i^{x,\eta-1/2} - \Delta \eta \left( \frac{a_\eta}{a_{\eta+1/2}} \right)^2 \left( \phi^{x+k_i,\eta} + \phi^{x-k_i,\eta} \right) e^{iA_{\eta}^{x+1/2k_i,\eta}}
\]
\[
+ \frac{\Delta \eta}{(\Delta x)^2} \left( \frac{a_\eta}{a_{\eta+1/2}} \right)^2 \sum_j \left( \phi^{x+k_j,\eta} e^{iA_{\eta}^{x+1/2k_j,\eta}} - 2\phi^{x,\eta} + \phi^{x-k_j,\eta} e^{-iA_{\eta}^{x+1/2k_j,\eta}} \right)
\]
(3)
\[
E_i^{x,\eta+1/2} = \left( \frac{e_{\eta+1/2}}{e_{\eta-1/2}} \right)^2 E_i^{x,\eta-1/2} + 2\Delta \eta e_{\eta+1/2} Im \left[ (\phi^{x+k_i,\eta})^* e^{iA_{\eta}^{x+1/2k_i,\eta}} \phi^{x,\eta} \right]
\]
\[
- \frac{\Delta \eta}{(\Delta x)^2} \left( \frac{e_{\eta+1/2}}{e_{\eta}} \right)^2 \sum_{j \neq i} \left[ \sin(\Xi_{ij}) - \sin(\Xi_{ij+1}) \right]
\]
(4)
\[
\phi^{x,\eta+1} = \phi^{x,\eta} + \Delta \eta \Pi_i^{x,\eta+1/2}
\]
(5)
\[
A_i^{x,\eta+1} = A_i^{x,\eta} + \Delta \eta E_i^{x+1/2,\eta}
\]
(6)
which tells us how to update field variables at each time-step, up until half-a-light-crossing time (due to the periodic boundary conditions, evolving any further would be acasual). There is a subtlety in most field theory defect simulations: since the physical defect width is constant, in comoving coordinates it shrinks. This means that the true equations of motion result in strings that eventually fall through the lattice and can no longer be resolved. A way to bypass this problem is to fix the comoving width as seen in [2] or to first apply a comoving core growth period [13], such that by the end of the simulation defects can still be resolved. Since the defect width is inversely proportional to the scalar and vector masses, this means one must change the way \( e_\eta \) and \( \lambda_\eta \) behave as
\[
e(\eta) = e_0 \phi^{s-1}(\eta) \quad \lambda(\eta) = \lambda_0 \phi^{2(s-1)}(\eta)
\]
(7)
where \( s \) is a parameter such that \( s = 0 \) will force constant comoving core width and \( s = 1 \) recovers the original equations of motion (negative values would imply core growth).

For our simulations scalar and vector masses are equal by choosing \( e_0 = 1 \) and \( \lambda_0 = 2 \) to be set to 1 and 2 respectively. For the initial conditions, one chooses to mimic a field configuration after a phase transition, that at the same time obeys the discretized form of Gauss’s law. As such, the scalar field \( \phi \) is set to have a random phase, its norm is set to unity and all other field variables are set to zero. For the random phase, the library cuRAND by NVIDIA is used.

In order to validate the simulations, two diagnostics are calculated, a correlation length and a potential weighted velocity (taken from local gradient and rate of change of the scalar field),
\[
\xi_L = \sqrt{\frac{\mu}{L}} \quad \nabla^2 > \nu = \frac{2R}{1 + R}
\]
(8)
where
\[
R = \frac{\int |\phi|^2 V(\phi) d^3 x}{\int |D\phi|^2 V(\phi) d^3 x}
\]
(9)
The first estimator is taken from [13], and the second
matches the corrected definition present in [22], with a small difference on the weight function, which instead of being given by the Lagrangian is given by \((|\phi|^2 - \sigma^2)^2\) (akin to the potential, but unity at center of the string core). As a cross-check we use a further correlation length estimator, \(\xi_V\), where the total length of string is computed by finding all lattice plaquettes pierced by a string. The way to do this is to consider a gauge invariant winding around a plaquette

\[ W_{ij} = \frac{(Y_{i,x}^x + Y_{x,i}^y - Y_{i,x}^y - Y_{x,i}^x)}{2\pi} \]  

where \(Y_{i,x} = [(\phi^{x+k})_{\text{arg}} - (\phi^x)_{\text{arg}} + A_i^x]_\pi - A_i^x\), as first seen in [23]. Note that the argument of the scalar field is assumed to be between \([-\pi, \pi]\) and the term in \(\ldots\pi\) has \(\pi\) factors added or subtracted in order to bind the result to the interval \([-\pi, \pi]\). If \(W_{ij}\) is different from zero then a string is present, and a length of \(\Delta x\) is associated with each winding found.

**IMPLEMENTATION AND PERFORMANCE**

We now describe how our implementation utilizes the Compute Unified Device Architecture (CUDA) to evolve a network of Abelian-Higgs cosmic strings. Development and all benchmarks were done on a Nvidia Quadro P5000, with 2560 CUDA cores, a core clock of 1607 MHz and 16384 MB of memory, clocked at 1216 MHz. There are three stencil kernels that evolve the field configurations at every time-step: the first one corresponds to Eq. 3, the second to Eq. 4, and the third one to Eqs. 5–6.

For the first two kernels one loads relevant field quantities from global memory to 2D shared memory XY padded-tiles and streams through the \(z\)-direction [24–27]. The main advantage of doing so is to enable software pre-fetching: load only the field at the next \(z\)-position (into temporary variables/registers) and when streaming up the \(z\)-direction this register value is loaded into the current shared memory tile. Similarly the previously current shared memory tile is loaded into the bottom registers (see Fig. 1). In stepB, and in the kernels which calculate average network quantities, instead of registers above and below, we use shared memory tiles for the top and bottom (complete with halos). There are two reasons to do so: for convenience (some calculations may require values on the top/bottom tile’s halos) and to reduce register pressure.

There is another advantage to loading field values into shared memory titles and/or registers. The field variables are given by the aligned vector types defined in CUDA (float2 and float4) and while the vector loads ensure coalesced memory reads, some computations which require specific components of each field would cause un-coalesced reads. This bottleneck is circumvented by using shared memory and registers. The third kernel is more straightforward, since software pre-fetching cannot be implemented. It simply reads the fields and their conjugates from global memory and writes the updated field values again.

The three kernels are limited by memory bandwidth, in particular when reading from global memory, as indicated by the NVIDIA Visual Profiler. As such, the most relevant performance metric is the effective bandwidth (bytes loaded and stored from/into global memory per second) and how it compares to the peak bandwidth of the GDDR5X memory present in the test-bench graphics card. The average bandwidth reached for each kernel (together with additional metrics) can be found in Table I, for box size 256\(^3\), in the radiation era and for constant comoving width. It is seen that that we are close to peak bandwidth (288.5 GB/s).

The three kernels are limited by memory bandwidth, in particular when reading from global memory, as indicated by the NVIDIA Visual Profiler. As such, the most relevant performance metric is the effective bandwidth (bytes loaded and stored from/into global memory per second) and how it compares to the peak bandwidth of the GDDR5X memory present in the test-bench graphics card. The average bandwidth reached for each kernel (together with additional metrics) can be found in Table I, for box size 256\(^3\), in the radiation era and for constant comoving width. It is seen that that we are close to peak bandwidth (288.5 GB/s).

An important detail is the chosen size of thread block, in particular for the first two kernels. In general stencil kernels prefer a larger thread block size in order to mitigate the performance hit from loading tile halos. However, in our case, due to the data-reuse pattern above one must also consider if the thread-block size will not result in register/shared memory overuse. With the help of the online CUDA Occupancy Calculator [28], the thread-block size of 288.5 GB/s.

Table I. The effective Global Load and Store bandwidth (in units of GB/s), the number of Floating Point Operations per second and the achieved occupancy, for a 256\(^3\) simulation in the radiation era and for constant comoving width.

| Kernel | GLS (GB/s) | GFLOPs | Occupancy (%) |
|--------|-----------|--------|---------------|
| stepA  | 245.92    | 5.78   | 48.0%         |
| stepB  | 271.60    | 3.90   | 47.8%         |
| stepC  | 264.83    | 0.21   | 90.8%         |
| LagVel | 121.15    | 14.45  | 47.8%         |
| WindVel| 112.66    | 14.47  | 48.8%         |

Figure 1. Schematic representation of the stepA kernel: the tile in the middle represents a 2D shared memory tiles where current values in the z-direction (site \(k\)) are loaded together with halos and register values (blue pinheads) hold field values directly above and below (\(k - 1\) and \(k + 1\)).
size that seemed to yield best performance was (32,4) at 256³ box size. The main limiting factor for the occupancy per Streaming Multiprocessor seems to be register pressure, as shown by the NVIDIA Visual Profiler. In all cases the occupancy is large enough that increasing it might not yield better performance (as previously stated, the Visual profiler doesn’t indicate latency as the main performance bottleneck).

The kernel which calculates the correlation length $\xi_L$ and the velocities $\langle v^2 \rangle^V$ (hereinafter named LagVel) operates by using the memory pattern described above to load data into shared memory titles, computing the Lagrangian and the numerator and denominator of $R$ for each thread (cf. Eq. 9, result stored in a register) and using the CUDA Unbound library [29] to compute a block-wide sum. Since each block computes a partial sum, we then transfer these back to the host and after summing we write to disk. The partial sums are calculated on the GPU in order to avoid becoming IO-bound (PCI-E buses could be easily saturated by transferring the values of each field variable to the host). A similar scheme is employed on the alternative kernel WindVel, where instead of the Lagrangian based length estimate, we use the winding based one.

The first calculation kernel is bottlenecked by both compute and memory requirements. The memory requirements are in part explained by the excessive register spilling that occurs (this can be avoided by not limiting the maximum number of registers to 64 per thread with compiler option --maxrregcount, but the side-effect is that it significantly reduces the occupancy, and this heavily impacts performance). The impact of spilling is mitigated by turning on the compiler flag --Xptxas dlcm=ca which caches these spills in L1. Improving the compute part however is more challenging: many of the compiler flags which attempt utilization of hardware intrinsics, or reduce the precision of certain operations often affect the quality of the diagnostics, either changing the asymptotic quantities themselves or increasing uncertainties. (The WindVel kernel, being compute bound also suffers from this.) Still there is one simple optimization that reduces runtimes: avoid executing this kernel at every timestep. In other words, we calculate useful quantities every $n$ timesteps (hereinafter we take $n = 5$), and reduce statistical uncertainties by doing multiple runs. This effectively reduces the time spent in the calculation kernels, as can be seen in Table II for an example run using LagVel (WindVel executes in an equivalent amount of time).

Table II. Total elapsed time of the three evolution kernel plus LagVel (which calculates two averaged network quantities) and the total runtime of a single 256³ and one 512³ run. The first three kernels are executed every timestep, while the last kernel is executed only every 5 timesteps.

| Kernel  | 256³ Time (s) % time | 512³ Time (s) % time |
|---------|----------------------|----------------------|
| stepA   | 2.29 24.52           | 36.89 24.88          |
| stepB   | 3.10 33.17           | 49.64 33.48          |
| stepC   | 2.89 30.87           | 46.54 31.39          |
| LagVel  | 1.05 11.24           | 14.87 10.03          |
| Runtime | 10.72                | 150.42               |

One final remark about the time spent in Input/Output operations (transferring partial sums, computing the final sum on the host, cf. Table I) is that we can speed up the simulation further by overlapping compute on the GPU with the aforementioned operations; however, how much can be gained in terms of speed will also depend on how often we choose to calculate useful quantities. For now, given that a reduced number of calls to LagVel or WindVel kernels diminishes the need for such an optimization and that we can venture into multi-GPU territory, we keep everything non-overlapped.

VALIDATION

We have checked that the discretized form of Gauss’s law is preserved to machine precision and, inspecting iso-surfaces of the scalar field provides visual confirmation that a network of strings is formed and evolves as expected—an example is in Fig. 2.

In contrast to the domain walls GPU code [20], where we had the serial version of the simulation (which was tested and validated before [11, 21, 30] and could directly compare with it. In the present strings case both the serial and parallel versions are completely new to Figure 2. Isosurfaces of the absolute value of the complex scalar field with the value of 0.5, showing a network of Abelian-Higgs cosmic strings in the radiation era, at timestep 60, with box size 512³.
the authors, so we will validate them by evaluating the asymptotic scaling values and comparing them with the results in the literature (which come from CPU codes). This comparison is summarized in Table III.

Comparing directly with the values of $\xi_L$ in [13] we find excellent agreement for both matter and radiation era simulations. Our other length estimator, $\xi_W$, is also in agreement with the results of the first, but in mild disagreement (one to two standard deviations) with in [31]. This is a consequence of the equations of motion containing a term in \( \dot{a}/a \) which acts like a damping term. The extended dynamic range and initial conditions (that yield scaling as quickly as possible) from [31], then reveal a slow drift in the $d\xi/d\eta$ value (changing the $\xi$ from the 0.3 value).

Our velocity estimator seems to slightly underestimate literature results, which is expected, as in [22] the equation of state parameter velocity estimator yields a better agreement with standing string velocities (specifically, see Figure 6 in [22]). Note that in the matter era we cannot evolve the true equations of motion (one needs a larger dynamic range in order to successfully use core-growth), though this can be resolved with a larger dynamic range. As such, we elect to compare our $s = 0$ to the $s = 1$ value of [22], for matter velocities. Plots of all the above-mentioned quantities throughout the duration of the simulation in both radiation and matter era (with and without core growth, where applicable) can be found in Figure 3.

**CONCLUSIONS AND OUTLOOK**

We have implemented field theory cosmic string evolution for the $U(1)$ model using the Compute Unified Device Architecture, such that it uses Graphics Pro-

| Epoch | $s$ | $\xi_L$ | $\xi_W$ | $< v^2 >_\nu$ | Literature |
|-------|-----|---------|---------|---------------|------------|
| Radiation | 1 | $0.33 \pm 0.02$ | - | - | 13 |
| Radiation | 1 | - | - | $0.37 \pm 0.01$ | [22] @4096³ |
| Radiation | 1 | $0.32 \pm 0.01$ | $0.32 \pm 0.03$ | $0.34 \pm 0.01$ | This work |
| Radiation | 0 | $0.31 \pm 0.02$ | - | - | 13 |
| Radiation | 0 | - | $0.26 \pm 0.02$ | - | [31] @1024³ |
| Radiation | 0 | $0.30 \pm 0.02$ | $0.32 \pm 0.03$ | $0.34 \pm 0.01$ | This work |
| Matter | 0 | $0.30 \pm 0.01$ | - | - | 13 |
| Matter | 0 | - | $0.28 \pm 0.01$ | - | [31] @1024³ |
| Matter | 0 | $0.29 \pm 0.01$ | $0.29 \pm 0.02$ | $0.26 \pm 0.01$ | This work |
| Matter | 1 | - | - | $0.31 \pm 0.01$ | [22] @4096³ |

Table III. Numerical results for asymptotic scaling quantities $\xi$ (calculated using the Lagragian or the winding estimator) and the weighted velocity $< v^2 >_\nu$ for $s = 0$ and $s = 1$ (where applicable) from our simulations and from the literature. All quantities assume a box size of $512^3$, except where otherwise noted.

![Figure 3](image-url)
cessing Units as accelerators. We summarized the main implementation steps in terms of the performance of each kernel and showcased the achievable performance. In addition we compared the key physical diagnostic parameters for the correlation length and the mean velocity squared to those previously reported in the literature, finding very good agreement and thus providing a preliminary validation of the code.

Compared to our previous GPGPU application (GPUsWall, [20]), the main bottlenecks in the present one are the evolution kernels, since one can force the calculation of useful quantities to occur every few timesteps and not every single timestep. This means that in contrast to the previous code we evade being compute-bound completely. The walls code also doesn’t use software pre-fetching as was used here. Implementing these strategies is a task left for subsequent work. The main challenges regarding the scalability of this code lie not only in being memory-bound but also in its memory requirements: given that two vector fields (float4’s) and two complex scalar fields (float2’s) are stored in 48 bytes per lattice site, the largest box one could possibly simulate with one GPU is, at the time of writing, 512^3 floating point operations per lattice site. This brings us to our next step: to extend this simulation with multi-GPU support. In principle, given the large necessary number of GPU’s required, the most natural way to implement multi-node, multi-GPU support would be through the Message-Passing-Interface.

Overall we conclude that there is a tangible performance benefit to using GPUs in field theory defect simulations, enabling the possibility of running thousands or tens of thousands of high-resolution field theory simulations in acceptable amounts of time. These simulations can then be used to yield accurate full-sky maps of cosmic microwave or gravitational wave backgrounds which can be used in the data analysis of forthcoming experiments, leading to more robust as well as more stringent constraints. We thus foresee that GPU-based defect codes will in the medium term become the gold standard in the field.

This work was financed by FEDER—Fundo Europeu de Desenvolvimento Regional funds through the COMPETE 2020—Operational Programme for Competitiveness and Internationalisation (POCI), and by Portuguese funds through FCT - Fundação para a Ciência e a Tecnologia in the framework of the project POCI-01-0145-FEDER-028987. J.R.C. is supported by an FCT fellowship (SFRH/BD/130445/2017). We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Quadro P5000 GPU used for this research.

[1] T. W. B. Kibble, J. Phys. A9, 1387 (1976).
[2] A. R. Ade et al. (Planck), Astron. Astrophys. 571, A25 (2014) [arXiv:1303.5085 [astro-ph.CO]].
[3] B. P. Abbott et al. (Virgo, LIGO Scientific), Phys. Rev. D97, 102002 (2018) [arXiv:1712.01168 [gr-qc]].
[4] F. Finelli et al. (CORE), JCAP 1804, 016 (2018) [arXiv:1612.08270 [astro-ph.CO]].
[5] P. Binetruy, A. Bohe, C. Caprini, and J.-F. Dufaux, JCAP 1206, 027 (2012) [arXiv:1201.0983 [gr-qc]].
[6] D. P. Bennett and F. R. Bouchet, Phys. Rev. D41, 2408 (1990).
[7] B. Allen and E. P. S. Shellard, Phys. Rev. Lett. 64, 119 (1990).
[8] C. J. A. P. Martins and E. P. S. Shellard, Phys. Rev. D73, 043515 (2006) [astro-ph/0511792].
[9] K. D. Olum and Y. Vanchurin, Phys. Rev. D75, 063521 (2007) [astro-ph/0604119].
[10] J. J. Blanco-Pillado, K. D. Olum, and B. Schlaer, Phys. Rev. D83, 083514 (2011) [arXiv:1101.5173 [astro-ph.CO]].
[11] C. J. A. P. Martins, I. Y. Rybak, A. Avgoustidis, and E. P. S. Shellard, Phys. Rev. D93, 043534 (2016) [arXiv:1602.01322 [hep-ph]].
[12] J. Briggs, S. J. Pennycook, E. P. S. Shellard, C. J. A. P. Martins, M. Woodacre, and K. Feind, Unveiling the Early Universe: Optimizing Cosmology Workloads for Intel Xeon Phi Coprocessors in an SGI UV 2000 System, Tech. Rep. (SGI/Intel White Paper, 2014).
[13] N. Bevis, M. Hindmarsh, M. Kunz, and J. Urrestilla, Phys. Rev. D75, 065015 (2007) [astro-ph/0605018].
[14] A. Lopez-Eiguren, J. Urrestilla, and A. Achucarro, JCAP 1701, 020 (2017) [arXiv:1611.09628 [hep-ph]].
[15] A. Achucarro, A. Avgoustidis, A. M. M. Leite, A. Lopez-Eiguren, C. J. A. P. Martins, A. S. Nunes, and J. Urrestilla, Phys. Rev. D89, 065033 (2014) [arXiv:1312.2123 [hep-ph]].
[16] N. Bevis and P. M. Saffin, Phys. Rev. D78, 023503 (2008) [arXiv:0804.0280 [hep-th]].
[17] J. Lizarraga and J. Urrestilla, JCAP 1604, 053 (2016) [arXiv:1602.08014 [astro-ph.CO]].
[18] M. Hindmarsh, K. Rummukainen, and D. J. Weir, Phys. Rev. D95, 063520 (2017) [arXiv:1611.08456 [astro-ph.CO]].
[19] P. McGraw, Phys. Rev. D57, 3317 (1998) [astro-ph/9706182].
[20] J. R. C. C. Correia and C. J. A. P. Martins, Phys. Rev. E96, 043310 (2017).
[21] W. H. Press, B. S. Ryden, and D. N. Saffin, Astrophys. J. 347, 590 (1989).
[22] M. Hindmarsh, J. Lizarraga, J. Urrestilla, D. Daverio, and M. Kunz, Phys. Rev. D96, 023525 (2017) [arXiv:1703.06690 [astro-ph.CO]].
[23] K. Kajantie, M. Karjalainen, M. Laine, J. Pisa, and A. Rajantie, Phys. Lett. B428, 334 (1998) [arXiv:hep-ph/9803367].
[24] Y. Zhang and F. Mueller, in Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO ’12 (ACM, New York, NY, USA, 2012) pp. 155–164.
[25] P. Micikevicius, in Proceedings of 2Nd Workshop on General Purpose Processing on Graphics Processing Units, GPGPU-2 (ACM, New York, NY, USA, 2009) pp. 79–
[26] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey, in 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (2010) pp. 1–13.

[27] E. H. Phillips and M. Fatica, in 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS) (2010) pp. 1–10.

[28] Xmartlabs, “Cuda occupancy calculator,” https://github.com/xmartlabs/cuda-calculator (2012).

[29] N. R. NVLabs, “Cub - cuda unbound v1.8.0,” https://nvlabs.github.io/cub/ (2018).

[30] C. J. A. P. Martins, I. Yu. Rybak, A. Avgoustidis, and E. P. S. Shellard, Phys. Rev. D94, 116017 (2016) [Erratum: Phys. Rev.D95,no.3,039902(2017)], arXiv:1612.08863 [hep-ph].

[31] N. Bevis, M. Hindmarsh, M. Kunz, and J. Urrestilla, Phys. Rev. D82, 065004 (2010) arXiv:1005.2663 [astro-ph.CO].