

FASTER DARK MATTER CALCULATIONS USING THE GPU

Bachelor's Thesis

Sebastian Liem

Supervisor: Professor Joakim Edsjö

Department of Physics
Stockholm University

September, 2011

Abstract

We have investigated the use of the graphical processing unit to accelerate the software package DarkSUSY. DarkSUSY is, among other things, used for calculating the dark matter relic density – an measurable quantity – given the supersymmetric neutralino, $\tilde{\chi}$, as a dark matter candidate. Supersymmetric theories have many free parameters and we want to calculate the relic density for large areas of the parameter space. The results can then be compared with observations and to constrain the parameters. A faster DarkSUSY would allow for larger searches in the parameter space. We modified DarkSUSY using Nvidia's CUDA platform and wrote a program that, by using the GPU, calculates the $\tilde{\chi} + \tilde{\chi} \rightarrow W^+ + W^-$ contribution to the annihilation cross-section. Our initial try was only negligible faster than our non-CUDA program due to under-utilization of the GPU, but solving that the program was 47 times faster than the reference program. We also report on difficulties we faced, both solved and unsolved so the reader can make an informed decision on the worth of rewriting so that the heavy calculations in DarkSUSY use the GPU.



Stockholm
University

Sammanfattning

Vi har undersökt om man kan använda grafikkortet för att få mjukvarupaketet DarkSUSY snabbare. DarkSUSY används, bland annat, för att beräkna relikdensiteten av mörk materia – en mätbar kvantitet – användandes den supersymmetriska neutralinon, $\tilde{\chi}$, som mörk materia-kandidat. Supersymmetriska teorier har många fria parametrar och vi vill beräkna relikdensiteten för stora områden av parameterutrymmet. Resultaten kan sedan jämföras med observationer för att begränsa parametrarna. Ett snabbare DarkSUSY skulle tillåta större sökningar i parameterutrymmet. Vi modifierade DarkSUSY med hjälp av Nvidias CUDA-plattform och skrev ett program som, genom att använda grafikkortet, beräknar $\tilde{\chi} + \tilde{\chi} \rightarrow W^+ + W^-$ kanalens bidrag till annihilationstvärsnittet. Vårt första försök var bara försumbart snabbare än vårt icke-CUDA program på grund av underanvändning av grafikkortet. Men med det åtgärdat så var programmet 47 gånger snabbare än referensprogrammet. Vi rapporterar också de problem vi stött på, både de vi löste och de vi inte löste. Detta så att läsaren kan avgöra värdet av att omarbete så att alla de beräkningsintensiva delarna av DarkSUSY använder grafikkortet.

Acknowledgments

I want to thank my parents, all of them, for their love and support, my sister for always putting me in my place, my brother for all our wild games and Hedvig for all the love.

I would also like to thank the Department of Physics at Stockholm University for providing office space, I could not have done it from home.

Finally I would like to thank my supervisor Joakim Edsjö for guiding me on my first tentative steps into the world of physics research.

Contents

1	Introduction	4
1.1	Disposition	4
2	Dark matter	5
2.1	Support for the existence of dark matter	5
2.2	Supersymmetry	7
2.3	Relic density	8
2.4	Searching the parameter space	9
3	GPU computing	9
3.1	CUDA	10
3.2	CUDA's memory model	11
4	CUDAfying DarkSUSY	14
4.1	Where to parallelize?	14
4.2	Reference program	15
4.3	<code>dsandwdcosnmod</code>	16
4.4	Calling C functions from Fortran	17
4.5	Accessing DarkSUSY global variables from C	18
4.6	Transferring DarkSUSY global variables to the GPU	19
4.7	Porting DarkSUSY subroutines to CUDA C	20
4.7.1	<code>dsankinvar()</code>	20
4.7.2	<code>dsan*ff*vv()</code>	21
4.7.3	<code>dsanclearaa()</code> and <code>dsansumaa()</code>	21
4.8	Double vs single precision	21
4.9	Result	21
4.10	Parallelization in p	22
5	Discussion	23
5.1	Is it worth it?	23
5.2	Alternatives	24
5.3	Concluding remarks	25
A	Computer specification	27

1 Introduction

Most people associate graphics cards, or the graphical process unit (GPU), with computer games. However, in recent years the GPU has become more and more used in the context of scientific and engineering computing. This is due to a combination of the GPU's massively parallel architecture, which tends to translate to massive performance gains, and the comparable ease of programming the GPU since Nvidia released their CUDA architecture in 2006.

Hagiwara *et al.* [1, 2] have successfully used the GPU to speed up cross-section calculations by a factor 40-150. Calculating the cross-section is very expensive and lies in the heart of many particle physics computations which is why these performance gains are interesting.

Hagiwara *et al.* use a quite complicated software stack which begins with the MadGraph program [3]. In MadGraph you input your particle physics model and then it, together with the rest of the software stack, can generate code for most particle physics calculations you might want to do.

The code for calculating cross-sections uses a software package called HELAS. Hagiwara *et al.* have been translating the generated HELAS code to their newly developed HEGET (HELAS Evaluation with GPU Enhanced Technology) code.

The aim of this Bachelor's thesis is to try replicating Hagiwara's *et al.* performance gains but for dark matter calculations. These also involve calculating the cross-sections of various interactions. However we will not use the MadGraph-based system, or HEGET, but instead a software package called DarkSUSY.

DarkSUSY [4] is a publicly available [5] Fortran package for numerical supersymmetric dark matter calculations written by Gondolo *et al.* With it we can, for a specified supersymmetric model, calculate the relic neutralino density of the universe or check accelerator bounds on our dark matter candidate.

We have not tried to convert the entire DarkSUSY software package to utilize the GPU, only a small trial program, as the full conversion would be quite an undertaking. Instead this thesis tries to provide the necessary information to decide if such an undertaking would be worthwhile.

1.1 Disposition

We have chosen to divide this report into, excluding this introduction, four sections. Two of these are introductory material to set the stage for the third one, which details our efforts at getting DarkSUSY to utilize the GPU. In the last section we discuss our results.

The first introductory section is a treatment of dark matter. We discuss the need for introducing dark matter to make sense of our universe. We

also discuss supersymmetry, an extension of the standard model of particle physics, as it provides us with a good candidate for what dark matter actually is.

We will place some focus on describing how to calculate the relic density, the amount of dark matter in our universe today, from theory. This is important as it is a prediction that can be compared to observations.

After the discussion of dark matter there is a section introducing the basics of GPU computing. We have based this on documentation [6] published by Nvidia and learning material found on their CUDA website [7].

After these two introductory sections we, as stated earlier, account for our work on using the GPU to accelerate DarkSUSY. This we do in rather deep detail, explaining what problems we encountered and what we did to solve or alleviate them.

We have chosen not to include source code with this report due to its large size. The code is instead available upon request. The technical specification of our computer environment is detailed in appendix A.

2 Dark matter

The WMAP survey of the cosmic microwave background indicate that only around 4.6% [8] of the energy content of the universe is baryonic matter, i.e. everyday matter such as protons and neutrons. These 4.6% includes every planet and star, every nebula and galaxy, we have ever seen.

These fantastic news begs the obvious question, in what form is the the rest of the energy? Combining the WMAP data with supernovae observations we get that around 25% is dark matter and the rest is dark energy [9]. These numbers assume a particular cosmology called Λ -CDM where Λ is the cosmological constant, which is one explanation for what dark energy is, and CDM stands for Cold Dark Matter.

Cold should be understood as non-relativistic and be contrasted with hot, or relativistic, dark matter. The canonical example of hot dark matter is neutrinos.

One might be tempted to say that the all dark matter is neutrinos – no need to postulate new kinds of matter. However if all dark matter is hot it would be hard to explain why our present day universe have so much structure [10].

Much of the following material is based on chapter 1, 7 and 16 in [11].

2.1 Support for the existence of dark matter

We do not know what dark matter actually is, or if it even exists, as we have never directly observed it in the laboratory. However Λ -CDM is a very successful model, often referred to as the standard model of cosmology, and that success in itself is a good indicator for the existence of dark matter.

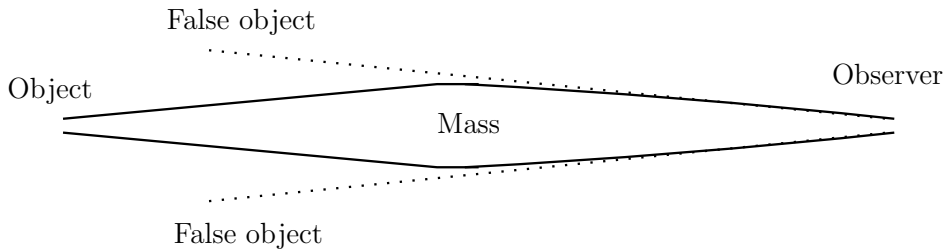


Figure 1: Schematic illustration of gravitational lensing. The mass bend the light so that the observer sees the object multiple times around the mass.

Dark matter is called dark because we cannot see it – it does not interact via the electromagnetic force. It does, however, have mass and thus do interact via the gravitational force. Its gravitational effects actually is the original reason for its supposed existence.

As early as 1937¹ Zwicky [12] estimated the mass of galaxy clusters from their internal movements. The results were drastically larger than estimate based on the luminosity, a discrepancy which might be explained by non-luminous dark matter.

Furthermore Vera Rubin saw in 1970 that the Andromeda galaxy does not spin the way it should given the distribution of the luminous mass. Rubin *at al* has since then been investigating the spin of many galaxies [13] and the discrepancy is almost universal.

The rotational speed of the stars seems to be, at least in large galaxies, almost independent of the distance to the galactic core. Our gravitational theories would have it depend as inverse square root of the distance, just like in our solar system. An explanation for this might be that there are halos of invisible dark matter around the galaxies.

Then we have gravitational lensing. Large masses can bend light so that it is possible to see what's behind massive objects as multiple images and arcs around the objects (see fig. 1). The phenomena is insensitive to the kind of mass, dark or luminous, and again the luminous mass does not seem to be the whole story as we have observed lensing that cannot be completely explained by the mass we see.

Of course all these phenomena could be explained by new physics rather than new matter (or an combination). General relativity is a very successful theory but has not, for obvious reasons, been tested directly at galactic or intergalactic distances. It is possible that an modified gravitational theory could explain these phenomena, but so far there is not any such modification that satisfyingly explains them all.

¹Actually Zwicky wrote a paper in 1933 but it is in German and hard to find: *Die Rotverschiebung von extragalaktischen Nebeln*, Helvetica Physica Acta, vol. 6, p. 110

There is also the famous Bullet Cluster which is made up of two colliding galaxy clusters. Clowe *et al.* [14] studied this cluster carefully, mapping the mass distribution using gravitational lensing and the distribution of the luminous mass using x-rays. Again there was missing mass but far more interesting was that the center of the total mass in the two cluster did not coincide with the center of luminous mass.

The authors call this a direct empirical proof of dark matter because this displacement obviously cannot be explained with modified gravity. It stands to reason that any modification cannot change the direction of the gravitational force and still be consistent with short-distance observations.

2.2 Supersymmetry

One of the most commonly considered dark matter candidate is the supersymmetric neutralino, $\tilde{\chi}$. To understand why that is we will introduce, very briefly, supersymmetry.

Supersymmetry is an extension of the standard model of particle physics with many theoretical motivations and no experimental evidence. It supposedly would solve the hierarchy problem of the standard model, unify of the electroweak and strong force, and give a mechanism for electroweak symmetry breaking. And it also gives us a promising dark matter candidate.

Now supersymmetry is a whole class of theories but the most talked about subclass is simplest², with only one supersymmetric generator, called the Minimal Supersymmetric Standard Model (MSSM). What it does is introduce a symmetry between fermions and bosons.

Each standard model fermion gets paired with a supersymmetric boson and each standard model boson with a supersymmetric fermion. These superpartners would have the same quantum numbers and mass as their standard model equivalents.

Now they cannot possibly have the same mass as then we would have seen them in our accelerators already. The symmetry must therefore also be broken, by some unknown mechanism, to allow for the superpartners to have greater mass than their standard model partners.

Supersymmetry predicts four neutralinos with different masses; the lightest would be our candidate as the others would decay to it. They are fermions and electrically neutral so they would not interact via the electromagnetic force. They do have weak charges through and would interact via the weak force. They are the mass eigenstates of superpositions of the supersymmetric particles: Wino, Bino and the neutral Higgsinos.

For the lightest neutralino to be a good dark matter candidate it has to be stable, otherwise it would decay into normal matter and then we would be able to see the decay products. For it to be stable a discrete symmetry

²Motivated by the paraphrased words of Albert Einstein: “Make everything as simple as possible, but not simpler.”

called R-parity must be introduced. While this might seem arbitrary R-parity also has other uses. One such use is that without R-parity MSSM predict violations of the baryon number and total lepton number in such degrees ruled out by experimental data [15]. Even so R-parity could be very slightly violated and still conform to observations.

2.3 Relic density

With the term relic density we mean the density of dark matter, neutralinos in our case, left in the universe today. These neutralinos would be produced in the earlier hotter universe where there was enough abundant energy for the neutralinos to be produced.

The relic density is a measurable quantity and we want to compare theory with the observed value. So we have to compute the relic density according to theory, and we do this by solving the Boltzmann equation

$$\frac{dn}{dt} = -3Hn - \langle \sigma_{\text{ann}} v \rangle (n^2 - n_{\text{eq}}^2) \quad (2.1)$$

where n is the number density of neutralinos, H is the Hubble parameter, $\langle \sigma_{\text{ann}} v \rangle$ is the thermally averaged annihilation cross-section and n_{eq} is the equilibrium number density of neutralinos.

The first term, $-3Hn$, comes from the expansion of the universe: the volume increases and so the density goes down.

The second term, $-\langle \sigma_{\text{ann}} v \rangle (n^2 - n_{\text{eq}}^2)$, express the chemical equilibrium of the reactions $\tilde{\chi} + \tilde{\chi} \leftrightarrow e^+ + e^-, \mu^+ + \mu^-, W^+ + W^-, \dots$ (any other pair of SM particles.)

Now, as the universe expands the temperature goes down which means n^2 decreases exponentially due to Boltzmann statistics. This means that the neutralino production cease and thus the amount of neutralinos per comoving volume become near constant.

Note that $\langle \sigma_{\text{ann}} v \rangle$ is the most intensive part of the calculation because so many reactions contribute. It becomes more intensive when we take into account that there are other supersymmetric particles than our neutralino that interact and contribute to $\langle \sigma_{\text{ann}} v \rangle$.

These are called co-annihilations and if we have N supersymmetric particles we can write the equivalent of (2.1) for particle i as [16]

$$\begin{aligned} \frac{dn_i}{dt} = & -3Hn_i - \sum_{j=1}^N \langle \sigma_{ij} v_{ij} \rangle (n_i n_j - n_i^{\text{eq}} n_j^{\text{eq}}) \\ & - \sum_{j \neq i} [\langle \sigma_{Xij} v_{ij} \rangle (n_i n_X - n_i^{\text{eq}} n_X^{\text{eq}}) - \langle \sigma_{Xji} v_{ij} \rangle (n_j n_X - n_j^{\text{eq}} n_X^{\text{eq}})] \\ & - \sum_{j \neq i} [\Gamma_{ij} (n_i - n_i^{\text{eq}}) - \Gamma_{ji} (n_j - n_j^{\text{eq}})] \end{aligned} \quad (2.2)$$

The two first right-hand terms is the same as in (2.1) but with the second including annihilations between all pairs of supersymmetric particles. The third term accounts for all $\chi_i \rightarrow \chi_j$ conversions via reactions with standard model particles. The last term accounts for decays.

Now all supersymmetric particles will decay to our stable lightest neutralino (assuming R-parity) so in the end the final density of our dark matter candidate is described by the sum of all densities of all supersymmetric particles:

$$n = \sum_{i=1}^N n_i \quad (2.3)$$

so for n we get

$$\frac{dn}{dt} = -3Hn - \sum_{i,j=1}^N \langle \sigma_{ij} v_{ij} \rangle (n_i n_j - n_i^{\text{eq}} n_j^{\text{eq}}) \quad (2.4)$$

The last two terms in (2.2) cancel. However this sum can be incorporated into a effective cross-section σ_{eff} and we can then rewrite the equation to a form resembling (2.1)

$$\frac{dn}{dt} = -3Hn - \langle \sigma_{\text{eff}} v \rangle (n^2 - n_{\text{eq}}^2) \quad (2.5)$$

2.4 Searching the parameter space

While calculating the relic density is complicated it does not take very long – in the most extreme cases; a couple of hours. But MSSM has many free parameters and we want to run searches through the parameter space. We want to calculate the relic density and checking constraints for each unique set of parameters.

Pure MSSM has around a hundred parameters, a way too large a parameter space to search through so one usually makes a lot of assumptions to reduce the number down to around 5–15 in the simpler models.

5-15 parameters gives a more manageable parameter space, but it is still huge and this is why speeding up the calculations is interesting. Reducing run time by a factor ten would mean an ten-fold increase of the parameter space we could scan.

3 GPU computing

A graphical processing unit (GPU) is a specialized circuit design to quickly and efficiently manipulate computer graphics. The requirement for doing so means that GPUs are highly parallel and optimized for floating point operations.

This puts them in contrast with the central processing unit (CPU) which is often better at integer arithmetic and most often not parallel at all. A traditional CPU can only run one instruction at a time, we say that it is capable of running one thread of computation.

Some high-end modern CPUs today have eight cores, each like a CPU in itself, which enables the execution of eight threads of computation simultaneously. Almost all modern CPU also uses hyper-threading which allows two threads to run on each core, our eight threads then becomes sixteen. However, for modern GPUs the number of threads is counted in hundreds if not thousands.

But it is important to note that CPU and GPU threads are not equivalent. While CPU threads are independent of each other the GPU is a single instruction, multiple data (SIMD) device. That means that each thread execute the same instructions as all others but each with different input data³.

This means that not every problem is suitable for GPU parallelization. The problem has to be amendable to a certain type of parallel formulation. Numerical integration and N-body simulations⁴ are two example problems well suited for the GPU due their inherent parallelism.

3.1 CUDA

CUDA is Nvidia's parallel computing architecture which allows a programmer to write programs for Nvidia GPUs. The programming language for doing so is a variant of the C programming language.

We will go through the CUDA programming model and, to keep the discussion clear and succinct, we will do so using an example. We will use a simple, if somewhat contrived, problem – adding two vectors, a and b, together and storing the result in an third vector r.

The conventional way to do this in C is by representing the vectors as arrays and then loop through them, adding each element together separately. If the vectors are of length 1000 the code would look like this

```
for(int i = 0; i < 1000; i++) {  
    r[i] = a[i] + b[i];  
}
```

This is definitely not parallel and if N is sufficiently large it will take some time. “Some time” here being microseconds on modern hardware but the operation(s) in the loop could easily have been more expensive.

³Actually GPUs are arrays of SIMD devices so it is more like groups of threads and that within each group the threads run the same set of instructions.

⁴Simulation of dynamical system of particle effected by forces. Examples in cosmology include the effect of dark matter on structure formation.

So how would we solve the problem using the GPU? The first thing we must understand is that, in CUDA, we have a *host* subsystem, consisting of CPU, RAM, hard-drive *et cetera*, and a *device* subsystem consisting of the GPU. The relationship between them is asymmetric, the host instructs the device in what is to be done.

We note that GPU threads are cheap so we might partition our problem so that each thread is responsible for taking one element out of *a* and *b*, adding them together and storing the result. In CUDA one specifies what each threads shall do by writing a special function called a *kernel*. Our kernel looks like this

```
--global-- void
addvectors(float *a, float *b, float *result) {
    int index = blockDim.x * blockIdx.x + threadIdx.x;
    result[index] = a[index] + b[index];
}
```

In CUDA you group threads into blocks, and how many threads in each block is specified by the programmer. Threads are numbered in each block, and the kernel can access that number through the variable `threadIdx`.

When one executes a kernel one also specify the desired number of blocks, this is called a grid. Each block is also numbered, accessible through the variable `blockIdx`. The expression `blockDim.x * blockIdx.x + threadIdx.x` evaluates to a unique number for each thread.

Assuming that our arrays are a thousand elements long we could launch our kernel by the following invocation

```
dim3 dimGrid(10);
dim3 dimBlock(100);
addvectors<<<dimGrid, dimBlock>>>(a, b, result);
```

We launch ten blocks, each with a hundred threads. Thread `{0,0}` (`{threadIdx.x, blockIdx.x}`) takes care of the first array elements, thread `{1,1}` takes care of the 102:nd array elements and so on.

You might have guessed from the `.x` following `threadIdx`, `blockIdx` and `blockDim` or the `dim3` type of `dimGrid` and `dimBlock` that these variables are three dimensional vectors. This can be useful depending on the problem one is solving. The maximum dimensions of blocks and grids are device dependent.

3.2 CUDA's memory model

Modern CPUs dedicate much of their transistors to data caching and flow control, GPUs instead use these transistors to achieve its proficiency at parallel floating point operations. This means that the task of proper memory management fall on the programmer.

The device subsystem, the GPU, has a separate memory hierarchy from the host. The host does not have free access to all device memory and the device cannot access host memory at all. This is a separation brought on by the physical separation of host and device subsystems and bandwidth between them is, in comparison, poor.

The host can read and write to *global memory* on the device using analogues of `malloc()`, `memcpy()` and friends. It is the only memory the host can access. Global memory is large, around 1-2 GiB, but accessing it is slow, even from the device. It has the same lifetime as the program.

Each block has *shared memory* associated with it. Each thread in the block can access it, has the lifetime of the block and it is quite fast. It is useful for explicitly caching global memory, enable cooperating multitasking and for facilitating coalesced memory access. Size is measured in KiB.

Finally each thread has a small local store of registers where automatic variables and arguments are stored. This is the fastest memory available.

In fig. 2 we have a useful illustration of CUDA's thread and memory hierarchy and the relationship between them.

Let us get back to our example kernel. We do not use any shared memory, its use is always explicit. `idx` and the pointers `a`, `b` and `result`, our arguments and automatic variables, are stored in local registers.

Where the data that the pointers are pointing to are stored is not explicitly stated in our example. We will remedy that now. Remember that we cannot access host memory from the device. So, we have to copy the values of `a` and `b` to global memory on the device from host memory. And after the computation is done, copy the values of `result` on the device to host memory.

If we ignore the declaration and initialization of the host vectors our kernel launch with memory management may look like this

```
float *a, *b, *result;
int size = 1000*sizeof(float);

/* allocate memory on the device (global memory) */
cudaMalloc((void*)&a, size);
cudaMalloc((void*)&b, size);
cudaMalloc((void*)&result, size);

/* copy our vectors to the device so that the
   kernel can access their values */
cudaMemcpy(a, a_host, size, cudaMemcpyHostToDevice);
cudaMemcpy(b, b_host, size, cudaMemcpyHostToDevice);

/* launch the kernel */
dim3 dimGrid(10);
dim3 dimBlock(100);
addvectors<<<dimGrid, dimBlock>>>(a, b, result);
```

```

/* copy the result back to the host */
cudaMemcpy(result_host, result, size, cudaMemcpyDeviceToHost);

/* deallocate memory on the device */
cudaFree(a);
cudaFree(b);
cudaFree(result);

```

There are two other types of memory available on the device, constant and texture memory. These are read-only for the device and have specific features reflecting the graphical processing nature of the device. For instance, texture memory offers data-filters. We will not be using neither of these memory types in this thesis.

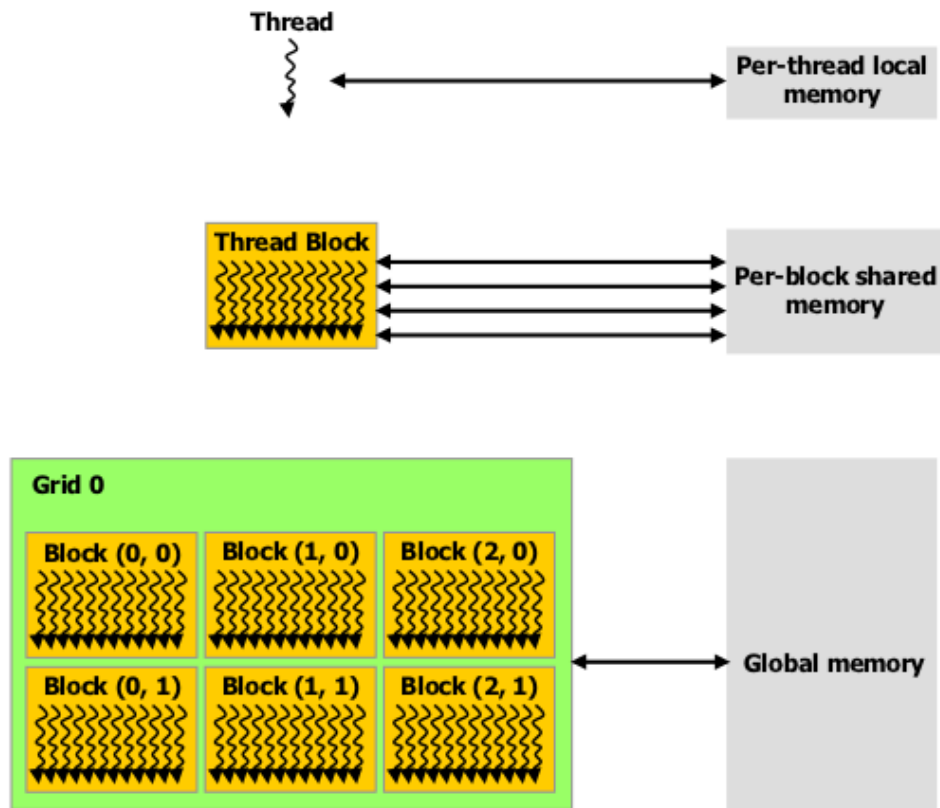


Figure 2: Graphical illustration of CUDA's thread and memory hierarchy. Courtesy of Nvidia [6].

4 CUDAfying DarkSUSY

This section reports on our work; improving DarkSUSY performance by utilizing the GPU. We have organized this section in a chronological fashion, explaining each problems, and our solutions, as we encountered them.

4.1 Where to parallelize?

There are several ways to parallelize DarkSUSY, some better than others. We tried looking for ways that minimize the extent of modification we had to do while maximizing performance gains.

Another constraint to consider is the SIMD nature of the GPU, i.e. the fact that all GPU threads must all run the same code. The problem of running a parameter search certainly matches up with SIMD principle. It involves running the same code, e.g. calculating the relic density, for a large number of parameter sets. We could let each thread calculate for one set of parameters.

Unfortunately this does not work because the non-trivial amount of code needed to calculate the relic density. That amount of code does not fit in the GPU, there is a limit on how large a kernel can be.

Instead we had to identify a sub-problem to parallelize. One of the more time intensive parts of calculating the relic density is computing the cross-section so we focused on that.

We note that DarkSUSY does not compute the cross-section explicitly, it instead computes W which is the Lorentz invariant annihilation rate per unit volume. It is related to the cross-section via $W = 4E_i E_j \sigma v_{ij}$ [16].

DarkSUSY calculates W by numerical integration – the integrand is the following expression

$$\frac{dW}{d \cos \theta} = \sum_{ijkl} \frac{p_{ij} p_{kl}}{32\pi p_{eff} S_{kl} \sqrt{s}} \sum_{\text{helicities}} \left| \sum_{\text{diagrams}} \mathcal{M}(ij \rightarrow kl) \right|^2 \quad (4.1)$$

The variable of integration is $\cos \theta$, and we integrate over $[-1, 1]$. The exact form of the integrand is not that important for our purposes, Edsjö and Gondolo [16] have more detail.

In the equation we sum over the helicities. Helicity is the projection of the spin in the axis of the momentum and it is a useful base for doing cross-section calculations. It allows us to rewrite the summation of the amplitudes from

$$\left| \sum a \right|^2 \quad \text{to} \quad \sum |a'|^2$$

This is advantageous as we then avoid a lot of canceling contributions which is troublesome for floating point numbers.

Numerical integration is a task well suited for CUDA because it involves evaluation of the integrand at many points in the integration area. Instead of doing this in a loop as a traditional DarkSUSY program does we can let the GPU evaluate the integrand at all the desired points simultaneously.

4.2 Reference program

In order to evaluate our efforts and verify that any modified program still produce the correct result we have a reference program that Joakim Edsjö prepared. Due to time constraints the program only takes into account the

$$\tilde{\chi} + \tilde{\chi} \rightarrow W^+ + W^- \quad (4.2)$$

annihilation channel. This channel was chosen because the associated code was neither too trivial nor too complicated, but still quite typical.

We will go through the reference program in an effort to explain how DarkSUSY works. This will hopefully provide the necessary context to understand our modifications.

A DarkSUSY program begins with calling the subroutine `dsinit` which initialize data structures and the like.

Next we call `dsgive_model` and `dssusy` with a set of MSSM parameters as arguments. This computes model specific values such as the masses of supersymmetric particles.

After that we call a function `dsandwdcosnmod` that computes (4.1) for a given value of $\cos\theta$ and momentum p . We call `dsandwdcosnmod` for 61 different values of $\cos\theta$ by looping. The number 61 is typical number of points needed to estimate the integral. The code looks as follows

```

do ij = 1, 50
  do kl = 1, 30
    do i = -30, 30
      dwdcos(i) = dsandwdcosnmod(p, cth(i), kn(1), kn(1))
    enddo
  enddo
enddo

```

The inner loop is what we later replace by calculating (4.1) for all i in parallel. For now you may notice that the outer loops does not do anything. They only run the inner loop $30 \cdot 50 = 1500$ times.

There are two reasons for this. The first reason is that it allows us to actually measure the execution time, the inner loop in itself actually runs quicker than we can measure on our hardware. Secondly it somewhat models calculating the true $\langle\sigma_{\text{eff}}v\rangle$ rather than only the $\tilde{\chi} + \tilde{\chi} \rightarrow W^+ + W^-$ contribution calculated by the reference program.

The loop over `kl` simulates that there could be other end state particles $e^+ + e^-$, $q + \bar{q}$, ... 30 is a nice round, and representative, number. The loop

over ij take into account co-annihilations, we can have other supersymmetric particles than our pair of neutralinos as initial particles. The number 50 is a bit arbitrary as it's very model dependent.

It does not model the proper calculation of $\langle\sigma_{\text{eff}}v\rangle$ perfectly, it does not take into account the cost of loading new kernels into the GPU. We tried to measure that cost by preparing slightly different kernels which we then launch after each other and thus forcing the GPU to load new kernel each time. The cost was slight in comparison to the cost of running the kernels.

Now in the end our reference program stop short of actually does the integration, it only evaluates the integrand for 61 different values of $\cos\theta$. We are not interested in the latter steps of computing W as it would be the same for both the reference program and our parallelized program.

4.3 dsandwdcosnmod

`dsandwdcosnmod` calculates a simplified version of (4.1) that only takes reaction (4.2) into account. As arguments it takes the value of the momentum, $\cos\theta$ and the initial particles⁵.

It does not do all the work itself, it calls a number of DarkSUSY subroutines. The first called is `dsankinvar` which calculates the kinematical variables used by later subroutines.

Next the function calls four different subroutines: `dsantfffv()`, `dsanufffv()`, `dsansffsvv()` and `dsansffvv()`. These each computes one Feynman diagram's contribution to the helicity amplitudes.

The `an` signify that these are annihilation diagrams, the `ff` that the two input particles are fermions and the `vv` that output particles are two vector bosons. The `f` or `v` between `ff` och `vv` signify if the exchange particle is a fermion or a vector boson. Exactly which particles they are is specified as arguments to the subroutines.

The `s`, `t` or `u` designate which kind of Feynman diagram the subroutine implement, the three possible are illustrated in fig 3. The variable s , t or u is the Mandelstam variable for each type of diagram. The Mandelstam variable is the four-momentum squared of the diagram's intermediate particle which is an important quantity in these calculations.

As we said these subroutines add one Feynman diagram's contribution to the helicity amplitudes which is stored in a multidimensional array of complex numbers called `aa`.

When contributions from all Feynman diagram have been added to `aa` `dsandwdcosnmod` calls an function, `dsansumaa`, to sum together all the helicity amplitudes.

⁵We only use the lightest neutralino as initial particles.

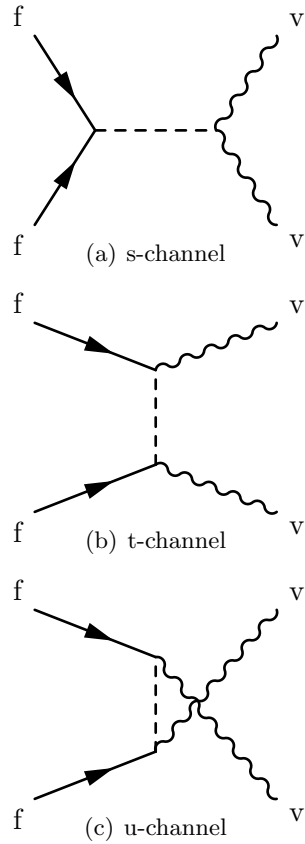


Figure 3: Feynman diagrams for the Mandelstam channels, the three possible events involving one intermediate particle.

4.4 Calling C functions from Fortran

DarkSUSY is written in Fortran 77 while CUDA uses a C dialect, so a DarkSUSY program cannot launch a kernel directly. However, calling C functions from Fortran is easy and such a function could in turn launch kernels, manage device memory and so on.

Calling a C function from Fortran involves writing the function in a separate file, compile it to object code and then link it with the Fortran program. Due to how Fortran handles object code symbols it is necessary to append an underscore to the name of the function.

Another thing to consider is the difference in calling semantics between C and Fortran. Fortran uses call by reference which means that a subroutine can modify the values of its arguments. It has, in other words, access to the memory location of its arguments.

C, on the other hand, is by default call by value which means the value of the argument is passed to the function but not the memory location. A

function therefore cannot modify the values of its arguments, unless one explicitly passes the memory addresses, using pointers, to the function.

For a C function to be callable from Fortran it must match Fortran's call by reference semantics by accepting pointers to its arguments instead of its arguments proper. To be explicit; instead of writing the function definition as

```
void f(double a, int b, double r);
```

we should write

```
void f_(double *a, int *b, double *r);
```

4.5 Accessing DarkSUSY global variables from C

DarkSUSY stores most of its state in common blocks. A common block is a Fortran concept and is, in effect, a named piece of global memory in which one can store any number of variables. To define a common block we could write

```
complex*16 gl(50,50,50),
-          gr(50,50,50)
common /vrtxs/ gl,gr
```

which defines a common block named `vrtxs` with two complex matrices of size 50x50x50 named `gl` and `gr`. This piece of code must be present in every subroutine that needs access to `gl` and `gr`, it defines an interface to that specific piece of memory.

If two subroutines happen to have different definitions of the same common block, the other subroutine might have `gr` before `gl`, the compiler does *not* complain. It instead assumes that you want to interpret that raw memory in a different way. So, in our example, the other subroutine will have the values of `gl` and `gr` exchanged.

The state stored by DarkSUSY in common blocks are numerical values for particle masses, vertices, coupling constants and mixing matrices and so on. The total size of all these common blocks are in the region of several megabytes.

By storing its state in global memory DarkSUSY avoids having to pass the state as arguments to each subroutine that require it. But in order to make any calculation with C code there must be a way to access these common blocks from C code.

The equivalent of common blocks in C are global structs. By writing struct definitions mirroring the common blocks and declaring them `extern`⁶ we provide a C interface to the common blocks. So for our `vrtsx` common block we write

```
extern struct vrtsx_ {
    doublecomplex gl[50][50][50];
    doublecomplex gr[50][50][50];
} vrtsx_;
```

As we stated earlier the compiler does not care if two definitions of the same common block are different as long as they occupy the same amount of memory. This is still true for the struct and the common block.

C compilers are also free to pad out structs to align memory to better match CPU implementations. So we had to be careful when we wrote the structs so both the order of the variables were correct and that we convinced the compiler to not do any padding.

As you might imagine, the C code's access to the common blocks are rather fragile and usually fail silently. We therefore wrote test code to verify that the structs access the common blocks correctly.

4.6 Transferring DarkSUSY global variables to the GPU

We do not simply want to do some of the calculations in C, we want to do them on the GPU. As we said in section 3.2 the GPU and CPU have separate memory hierarchies. We therefore have to copy relevant DarkSUSY data from the host side to the device.

Some of this data are model dependent and have the same value in each thread, they are particle masses, vertices and such. These are only read during the calculation of cross-sections, never written to so these we stored in device global memory.

It happens that the device side does not have a global environment so for our threads to have access to the data we have to pass them pointers to the data as arguments. To simplify the bookkeeping we defined a struct to hold all our global memory variables. This allowed us to only pass around a single pointer.

Other DarkSUSY global variables store intermediate results, e.g. `aa` which stores helicity amplitudes. Storing these in global variables works in a single thread environment but not in a multithreaded one. The threads would just thrash eachothers calculations.

We therefore chose to store these variables in each thread's local stores of registers. This also had the effect of improving performance as register

⁶The `extern` keyword means that the struct is initialized elsewhere, in our case on the Fortran side.

access is much faster than global memory. Again, to simplify bookkeeping, we stored all these variables in a big struct.

To manage these memory transfers we wrote two subroutines callable from Fortran; `dscudainit()`, which sets up and transfer model data to the device, and `dscudacleanup()` which tears down the data structures.

4.7 Porting DarkSUSY subroutines to CUDA C

The main function of interest in the reference program is `dsandwdcosnmod()`. Calls to this function, the inner loop in section 4.2, we replace with a C function called `dscudaandwdcosnmod()`. This new function is responsible for copying over the arguments to the device, launching a kernel and then copy the results back to the host.

The kernel launched by `dscudaandwdcosnmod()` is the real replacement for `dsandwdcosnmod()` in the sense that it does all the computation. The critical difference is that the kernel compute for all values of $\cos\theta$ simultaneously.

As we wrote in section 4.3 the original does the computation by calling a number of DarkSUSY subroutines. We cannot call these from the GPU so we wrote CUDA C versions of these for our kernel to call instead. DarkSUSY subroutine names are prefixed by `ds`, our CUDA version are prefixed by `dscuda`.

In total these subroutines are many hundred lines of Fortran code which we had to convert to C. This took a lot of time and while we spent some of it writing scripts to partially automate the process it was still tiresome and error-prone work.

4.7.1 `dsankinvar()`

As mentioned earlier, `dsankinvar()` computes kinematical variables used by later subroutines. The Fortran original uses static variables⁷ to avoid recomputing the variables if the input data have not changed since last call to `dsankinvar()`.

This type of optimization is not possible in CUDA as static variables are not allowed in device code. Hence our CUDA version does not retain that optimization, however a similar optimization was achieved by separating the calculation of the Wigner d-functions⁸ to its own function `wigner()`. Wigner d-functions only depend on $\cos\theta$ which is a constant in each thread.

⁷Variables who's values persist between function calls

⁸Wigner d-functions, or d-matrix is an irreducible representation of the groups SU(2) and SO(3) and is of use in these calculations.

4.7.2 `dsan*ff*vv()`

The expression `dsan*ff*vv()` abbreviates the list: `dsantfffv()`, `dsanufffv()`, `dsansffsvv()` and `dsansffvvv()`. As stated earlier, these each computes one Feynman diagram's contribution to the helicity amplitudes.

The original subroutines handles if one or more of the vector bosons are massless, i.e. photons. Our CUDA versions of subroutines does not because when when we included the photon code it failed to compile. The resulting device code became too large to fit on the GPU. We therefore decided to remove all photon code as we had no use for it in our reference program.

If we did a full calculation of the relic density we would need the removed photon code, in addition to the other process not included in reference program. One way to solve this would be to provide photon and non-photon variants of the `dsan*ff*vv()` subroutines.

4.7.3 `dsanclearaa()` and `dsansumaa()`

The helicity amplitudes are stored in a variable `aa` and we have two functions directly related to this variable. `dsanclearaa()` sets all elements to $0 + i0$ and `dsansumaa()` which sums the elements together like $\sum |aa_i|^2$. We wrote CUDA version of these two without any difficulties.

4.8 Double vs single precision

DarkSUSY works with double precision floating point numbers rather than single precision as that improved precision is often needed. While the version of CUDA we used supports double precision the GPU is still very much single precision hardware, double precision operations are much slower than single precision operations.

We therefore prepared two versions of our CUDA program, one single and one double precision version. If we compare the computed results from these versions with the reference program we see that the double version differ with $\pm 10^{-10}$ and the single version differ with $\pm 2 \cdot 10^{-9}$. So for our reference program single precision is good enough – in general this is not the case.

4.9 Result

We evaluated performance by measuring execution time using Fortran's `cpu_time()`, this should have an accuracy of 10 ms which is more than enough for our purposes. We do not measure DarkSUSY or model initialization, only the calculation proper. We do however include copying data to and from the GPU. We present our results in table 1.

ref	double	single
1.62 sec	3.45 sec (0.5)	1.28 sec (1.3)

Table 1: Execution time, on our hardware (appendix A), for the reference program, the single precision version and the double precision version of our CUDA program. The number in the parenthesis is how many times faster the CUDA program is compared to the reference program.

As we can see the single precision CUDA program is only slightly faster than the reference and the double precision CUDA program takes twice as long.

0.7 seconds of the CUDA programs execution time is spent copying the DarkSUSY state to the GPU, something which must be done every time one or more model parameters change. This transfer could be minimized because a kernel only use a small fraction of the model data and that fraction could be passed directly via the kernel’s arguments.

However this would also require much restructuring of existing DarkSUSY code and the single precision CUDA program would still, at best, only be three times faster than the reference program.

One explanation for our poor performance gains is that we under-utilize the GPU, we only use 61 threads, while speedups are noted when applications use thousands of threads.

One way to remedy that is launching several kernels in parallel but the hardware available to us, see appendix A, does not support that. Newer graphics cards, of compute capability 2.0 or later, can run up to 16 kernels simultaneously which would make the single precision CUDA program 2.2⁹ times faster than the reference program.

We did not do neither of these optimizations due to time and available hardware but if we did them both we would get performance gains in the region of a factor 40 faster than the reference program which would be more satisfying.

4.10 Parallelization in p

In order to calculate the thermally averaged annihilation cross-section $\langle\sigma_{\text{ann}}v\rangle$ we have to calculate the cross-section, or W via (4.1), for several different momenta of the initial particles.

This is a parallelism we have not exploited in our programs so far, we have only done the calculation for a single value of the momentum. And

⁹The number 2.2 does not account for the overhead associated with managing parallel kernel launches. It does include data transfer.

with our troubles of fully utilizing the GPU it seemed reasonable to extend our CUDA program to use this momentum parallelism.

We chose to do the calculation for a hundred values of the momentum as it is a typical number for calculating $\langle\sigma_{\text{ann}}v\rangle$. Our reference program simply got another loop over the momentum

```

do ij=1,50
  do kl=1,30
    do j=1,100
      do i=-30,30
        dwdcos(i)=dsandwdcosnmod(p(j),cth(i),kn(1),kn(1))
      enddo
    enddo
  enddo
enddo

```

and it takes 100 times longer (162.7 s) to execute than before on the CPU, not a very surprising result.

In our earlier CUDA program we launched one block with 61 threads but in this new version we launch 100 blocks of 61 threads (6100 threads in total) each block taking care of one value of the momentum. This CUDA program took 3.44 seconds to finish, or a factor 47 faster than the reference program.

5 Discussion

DarkSUSY is decidedly not designed with the GPU in mind, it is a strictly single thread program. And so we have, in some sense, fought against DarkSUSY's architecture when we did these modifications. This manifests itself in the 0.7 second cost of transferring data to the GPU or the removal of photon logic from the `dsan*ff*vv` subroutines.

One limitation inherent to the GPU and the CUDA platform, rather than DarkSUSY, is the poor performance of double precision numbers. While our reference program had no need for double precision there are other calculations in DarkSUSY where the extra precision is essential.

For Nvidia's consumer graphics cards, GeForce, working in double precision means 1/8 of the performance we would have if we used single precision. Nvidia's professional hardware, Quadro and Tesla, better, there we would only lose half of the performance but then these GPUs are at least ten times more expensive.

5.1 Is it worth it?

As we stated in the introduction; the purpose of this thesis is to investigate how much effort it would be to make DarkSUSY use the GPU and what performance improvements we can expect.

Estimating the time and effort required for a particular software development project is notoriously difficult and falls somewhat outside the scope of this report. It would very much depend on the people involved and resources available.

What we have done instead is detailed our efforts and the problems we have encountered. This has hopefully made clear the – rather extensive – scope of the changes needed to make DarkSUSY take full advantage of the GPU.

5.2 Alternatives

To fully assess the cost of any venture we have to consider opportunity costs. Are there other, easier and/or better, ways to do faster relic density calculations? If so, it would be wise to focus on those. We will discuss some alternatives but not investigate them in any depth as it would, again, falls outside the scope of this report.

The most straightforward way to make faster calculations is to use more hardware. A parameter search, being inherently parallel, could easily be divided over several instances of DarkSUSY, no modification to DarkSUSY necessary. These instances could even run on the same multi-core computer as DarkSUSY’s performance is very much CPU bound – meaning the CPU is the bottleneck¹⁰.

Assuming that DarkSUSY remains CPU bound when we increase the number of instances, we would need 47 CPU cores to match the performance gains we got by using the GPU. This could be a small computer cluster but there are also professional servers today with 48 CPU cores (4 processors with 12 cores each).

Another alternative would be to take advantage of Hagiwara *et al.* work on HEGET [1, 2] and switch out DarkSUSY’s own cross-section codes with MadGraph-based code. MadGraph [3] is a heavily used software stack for particle physics, with all the advantages that entail. It is also more general than DarkSUSY, it accepts different particle physics models and from that generates the necessary code for the calculations. Of course, this generality, in all probability, comes with some performance loss.

Now the benefits and cost of this alternative is hard to estimate as HEGET not being publicly available yet, being a work in progress. It might be interesting to note that there have been some *very* early work to make MadGraph to calculate the relic density itself[17].

¹⁰Joakim Edsjö have tested this with a 8-core CPU with hyperthreading, allowing 16 instances of DarkSUSY to run simultaneously without slowdown.

5.3 Concluding remarks

In this report we have investigated the use of the GPU to accelerate DarkSUSY. We chose to focus on the calculation of the cross-section, to be more specific the equation (4.1). In the end we were able to gain substantial performance improvement in comparison with our reference program.

However, the effort involved, and limitations of CUDA, temper that success. So before converting the entirety of DarkSUSY one should carefully consider the costs and possible alternatives.

References

- [1] K. Hagiwara, J. Kanzaki, N. Okamura, D. Rainwater, and T. Stelzer, *Calculation of HELAS amplitudes for QCD processes using graphics processing unit (GPU)*, *Eur. Phys. J.* **C70** (2010) 513–524, [arXiv:0909.5257 \[hep-ph\]](#).
- [2] K. Hagiwara, J. Kanzaki, N. Okamura, D. Rainwater, and T. Stelzer, *Fast calculation of HELAS amplitudes using graphics processing unit (GPU)*, *Eur. Phys. J.* (Apr., 2010) 477–492, [arXiv:0908.4403 \[physics.comp-ph\]](#).
- [3] J. Alwall, M. Herquet, F. Maltoni, O. Mattelaer, and T. Stelzer, *MadGraph 5: going beyond*, *Journal of High Energy Physics* **6** (June, 2011) 128–+, [arXiv:1106.0522 \[hep-ph\]](#).
- [4] P. Gondolo, J. Edsjö, P. Ullio, L. Bergström, M. Schelke, E. Baltz, T. Bringmann, and G. Duda, *DarkSUSY: Computing supersymmetric dark matter properties numerically*, *JCAP* **0407** (2004) 008, [arXiv:astro-ph/0406204](#).
- [5] P. Gondolo, J. Edsjö, P. Ullio, L. Bergström, M. Schelke, E. Baltz, T. Bringmann, and G. Duda, *DarkSUSY Homepage*, <http://www.physto.se/~edsjo/darksusy/>.
- [6] Nvidia, *CUDA C Programming Guide v3.2*, http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf.
- [7] Nvidia, *CUDA Zone*, http://www.nvidia.com/object/cuda_home_new.html.
- [8] N. Jarosik *et al.*, *Seven-Year Wilkinson Microwave Anisotropy Probe (WMAP) Observations: Sky Maps, Systematic Errors, and Basic Results*, *Astrophys. J. Suppl.* **192** (2011) 14, [arXiv:1001.4744 \[astro-ph.CO\]](#).

- [9] R. Amanullah *et al.*, *Spectra and Hubble Space Telescope Light Curves of Six Type Ia Supernovae at $0.511 < z < 1.12$ and the Union2 Compilation*, *Astrophys. J.* **716** (June, 2010) 712–738, [arXiv:1004.1711](https://arxiv.org/abs/1004.1711) [astro-ph.CO].
- [10] S. Hannestad, *Dark Energy and Dark Matter from Cosmological Observations*, *International Journal of Modern Physics A* **21** (2006) 1938–1949, [arXiv:astro-ph/0509320](https://arxiv.org/abs/astro-ph/0509320).
- [11] G. Bertone, ed., *Particle Dark Matter: Observations, Models and Searches*. Cambridge University Press, 2010.
- [12] F. Zwicky, *On the Masses of Nebulae and of Clusters of Nebulae*, *Astrophys. J.* **86** (Oct., 1937) 217.
- [13] V. C. Rubin, W. K. J. Ford, and N. . Thonnard, *Rotational properties of 21 SC galaxies with a large range of luminosities and radii, from NGC 4605 / $R = 4$ kpc/ to UGC 2885 / $R = 122$ kpc/*, *Astrophys. J.* **238** (June, 1980) 471–487.
- [14] D. Clowe *et al.*, *A direct empirical proof of the existence of dark matter*, *Astrophys. J.* **648** (2006) L109–L113, [arXiv:astro-ph/0608407](https://arxiv.org/abs/astro-ph/0608407).
- [15] S. P. Martin, *A Supersymmetry Primer*, [arXiv:hep-ph/9709356](https://arxiv.org/abs/hep-ph/9709356).
- [16] J. Edsjö and P. Gondolo, *Neutralino relic density including coannihilations*, *Phys.Rev.* **D56** (1997) 1879–1894, [arXiv:hep-ph/9704361](https://arxiv.org/abs/hep-ph/9704361) [hep-ph].
- [17] M. McCaskey, *MadDM*, in *MadGraph Spring 2011*. <http://indico.cern.ch/conferenceDisplay.py?confId=125190>.

A Computer specification

In this appendix we account for what hardware and software we used for our work. The software we have used are DarkSUSY 5.0.5, Nvidia CUDA 3.2, gcc 4.3.4 and gfortran 4.4.3. The computer we used ran Ubuntu 10.04.2 LTS (kernel: 2.6.32-30-server). As processor it had an Intel Core 2 Quad Processor (2.83 GHz) and the graphics cards was an GeForce GTX 260.

CUDA comes with a program called deviceQuery which probe the graphics card(s) and print out its capabilities. We include its output here.

```
./deviceQuery Starting...
```

```
[...]
```

```
Device 0: "GeForce GTX 260"
```

```
  CUDA Driver Version:            3.20
  CUDA Runtime Version:          3.20
  CUDA Capability Major/Minor version number:  1.3
  Total amount of global memory:  938803200 bytes
  Multiprocessors x Cores/MP = Cores:  27 (MP) x 8 (Cores/MP)
                                         = 216 (Cores)

  Total amount of constant memory:  65536 bytes
  Total amount of shared memory per block:  16384 bytes
  Total number of registers available per block:  16384
  Warp size:  32
  Maximum number of threads per block:  512
  Maximum sizes of each dimension of a block:  512 x 512 x 64
  Maximum sizes of each dimension of a grid:  65535 x 65535 x 1
  Maximum memory pitch:  2147483647 bytes
  Texture alignment:  256 bytes
  Clock rate:  1.24 GHz
  Concurrent copy and execution:  Yes
  Run time limit on kernels:  No
  Integrated:  No
  Support host page-locked memory mapping:  Yes
  Compute mode:  Default (multiple host
                  threads can use
                  this device
                  simultaneously)

  Concurrent kernel execution:  No
  Device has ECC support enabled:  No
  Device is using TCC driver mode:  No
```

```
[...]
```