

# AUTOTUNING NVCC PARAMETERS IN THE JULIA LANGUAGE

---

Pedro Buel and Alfredo Goldman

*{phrb, gold}@ime.usp.br*

August 24, 2017



Instituto de Matemática e Estatística  
Universidade de São Paulo



The slides and all source code are hosted at [GitHub](#):

- `github.com/phrb/nvidia-workshop-autotuning`

1. Introduction to Autotuning
2. Examples & Results on Different Domains
3. An Autotuning Library in the Julia Language
4. NVCC Flag Autotuner

Casting program optimization as a [search problem](#):

## [Search Spaces](#):

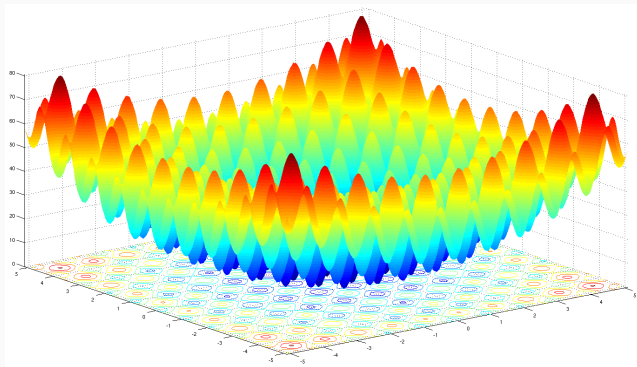
- Algorithm Selections
- Program Configurations
- ...

## [Search Objectives](#):

- Minimize [execution time](#)
- Maximize [usage of resources](#)
- ...

# SEARCH SPACES & TECHNIQUES

The **search spaces** created by program optimization problems can be **difficult to explore**



Rastrigin function, with **global minimum**  $f(0, 0) = 0$

## SEARCH SPACES & TECHNIQUES

| System         | Domain               | Technique              |
|----------------|----------------------|------------------------|
| ATLAS          | Dense Linear Algebra | Exhaustive             |
| Insieme        | Compiler             | Genetic Algorithm      |
| SPIRAL         | DSP Algorithms       | Pareto Active Learning |
| Active Harmony | Runtime              | Nelder-Mead            |
| Periscope      | HPC Applications     | Various                |
| OpenTuner      | Domain-Agnostic      | Ensemble               |

**Table 1:** Some autotuning systems, their domains and techniques

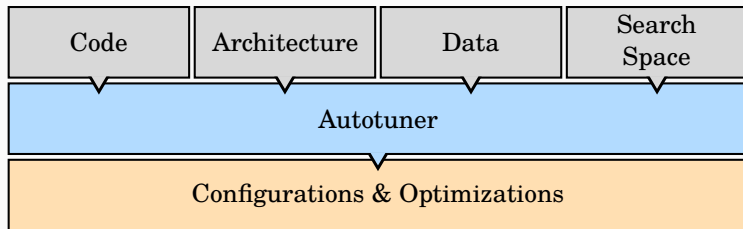
## SEARCH SPACES & TECHNIQUES

| System         | Domain               | Technique              |
|----------------|----------------------|------------------------|
| ATLAS          | Dense Linear Algebra | Exhaustive             |
| Insieme        | Compiler             | Genetic Algorithm      |
| SPIRAL         | DSP Algorithms       | Pareto Active Learning |
| Active Harmony | Runtime              | Nelder-Mead            |
| Periscope      | HPC Applications     | Various                |
| OpenTuner      | Domain-Agnostic      | Ensemble               |

**Table 1:** Some autotuning systems, their domains and techniques

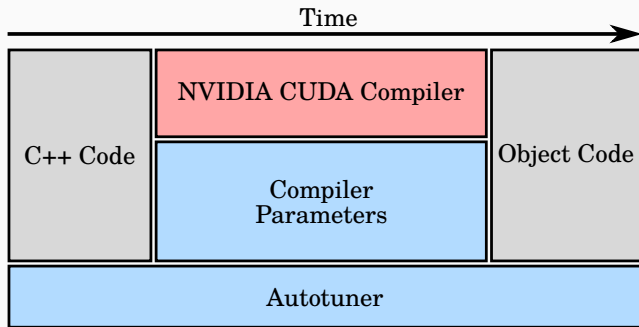
- Different **problem domains** generate different **search spaces**
- **No single solution** for all domains
- Search techniques can be composed: **OpenTuner**
- Independent searches can be **parallelized and distributed**

## AUTOTUNING: ABSTRACT MODEL





# NVIDIA CUDA COMPILER: FROM CUDA C++ TO OBJECT CODE



- We tuned applications from the [Rodinia Benchmark Suite](#)
- C++ → Object Code: takes [seconds](#)

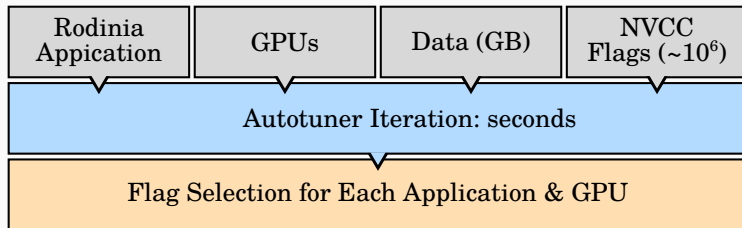
# RESULTS

Flags in the [search space](#):

| <i>Flag</i>             | <i>Step</i> |
|-------------------------|-------------|
| no-align-double         | NVCC        |
| use_fast_math           | NVCC        |
| gpu-architecture        | NVCC        |
| relocatable-device-code | NVCC        |
| ftz                     | NVCC        |
| prec-div                | NVCC        |
| prec-sqrt               | NVCC        |

| <i>Flag</i>                   | <i>Step</i> |
|-------------------------------|-------------|
| def-load-cache                | PTX         |
| opt-level                     | PTX         |
| fmad                          | PTX         |
| allow-expensive-optimizations | PTX         |
| maxrregcount                  | PTX         |
| preserve-relocs               | NVLINK      |

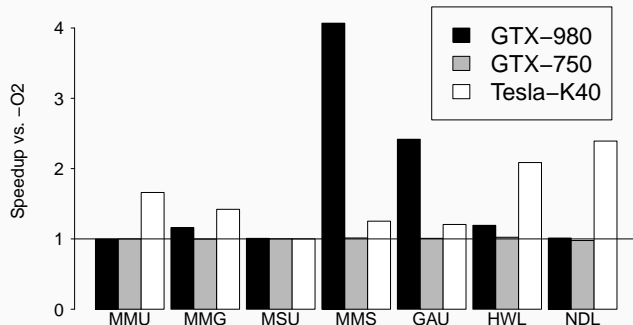
## AUTOTUNING: GPUS



1h of tuning  $\rightarrow \approx 10^3$  iterations

## RESULTS

Most significant speedups for Rodinia applications and matrix multiplication optimizations, after 1.5h of tuning:



We found no globally good parameter selections for specific GPUs or applications



## StochasticSearch

We are developing an autotuning library:

- In the [Julia](#) language
- [Domain-Agnostic](#)
- [Parallel and distributed autotuning](#)
- [github.com/phrb/StochasticSearch.jl](https://github.com/phrb/StochasticSearch.jl)

Multiple [simultaneous search techniques](#) share a search space, but not results:

[Implemented](#) techniques:

- Simulated Annealing
- Iterated Local Search
- Randomized First Improvement
- Iterative First Improvement
- Iterative Probabilistic Improvement
- Iterative Greedy Construction

Multiple [simultaneous search techniques](#) share a search space, but not results:

[Implemented](#) techniques:

- Simulated Annealing
- Iterated Local Search
- Randomized First Improvement
- Iterative First Improvement
- Iterative Probabilistic Improvement
- Iterative Greedy Construction

[Future](#) techniques:

- Iterative Best Improvement
- Randomized Best Improvement
- Dynamic Local Search
- Tabu Search
- Ant Colony Optimization
- ...

# WHY USE THE JULIA LANGUAGE?



Why the [Julia Language](#)?

- [High-Level abstractions](#)
- [Simple interface](#) for parallel and distributed programming
- [Better performance](#) than Python, Matlab, R, . . .



Parallel and Distributed programming:

- [Process-based](#) parallelism
- [Communication](#) between processes uses [channels](#)
- [Serialize/Deserialize](#) complex data types

Parallel and Distributed programming:

- [Process-based](#) parallelism
- [Communication](#) between processes uses [channels](#)
- [Serialize/Deserialize](#) complex data types
- Works on [many-core and clouds](#) using an MPI-like [machinefile](#)
- [No shared memory](#) yet, with [experimental native threading](#)

# PARALLEL AND DISTRIBUTED PROGRAMMING IN JULIA

Parallel and Distributed programming:

- [Process-based](#) parallelism
- [Communication](#) between processes uses [channels](#)
- [Serialize/Deserialize](#) complex data types
- Works on [many-core and clouds](#) using an MPI-like [machinefile](#)
- [No shared memory](#) yet, with [experimental native threading](#)

Simple interface:

- [Remote execution](#) on a [new process](#): `remotecall(·)` and `@spawn`
- [Channels](#): `put!(·)` and `take!(·)`

# STOCHASTICSEARCH EXAMPLE: THE ROSEN BROCK FUNCTION

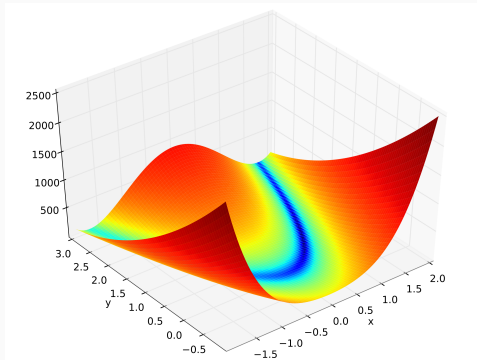
Let's create a [Tuning Run](#) using [StochasticSearch](#):

The Rosenbrock function:

- [Global minimum](#)  $f(1, 1) = 0$
- Many [local minima](#)

Components:

- [Parameters](#)
- [Configurations](#)
- [Cost Function](#)



## Parameters:

- Represent **individual optimizations**
- Determine **ranges** and **initial values**
- Integers, Booleans, Permutations, Strings, Enumerations, . . .

## Parameters:

- Represent individual optimizations
- Determine ranges and initial values
- Integers, Booleans, Permutations, Strings, Enumerations, . . .

## Example:

```
FloatParameter(-2.0, 2.0, 0.0, "i0")
```

## Configurations:

- Contain parameters
- Describe the search space
- Associate optimizations with metrics

## Configurations:

- Contain **parameters**
- Describe the **search space**
- Associate **optimizations** with **metrics**

## Example:

```
configuration = Configuration([FloatParameter(-2.0, 2.0, 0.0, "i0"),  
                             FloatParameter(-2.0, 2.0, 0.0, "i1")],  
                             "rosenbrock_config")
```



## Cost Function:

- Computes **costs** for each **configuration**
- **Costs** are usually expressed in **floating point values**

## Cost Function:

- Computes **costs** for each **configuration**
- **Costs** are usually expressed in **floating point values**

## Example:

```
function rosenbrock(x::Configuration, parameters::Dict{Symbol, Any})  
    return (1.0 - x["i0"].value)^2 + 100.0 * (x["i1"].value - x["i0"].value^2)^2  
end
```

## Tuning Run:

- Selection of search techniques
- Duration & communication

# STOCHASTICSEARCH: DEFINING SEARCH SPACES & TUNING RUNS

## Tuning Run:

- Selection of [search techniques](#)
- [Duration](#) & [communication](#)

## Example:

```
tuning_run = Run(cost          = rosenbrock,  
                 starting_point = configuration,  
                 stopping_criterion = elapsed_time_criterion,  
                 report_after    = 10,  
                 reporting_criterion = elapsed_time_reporting_criterion,  
                 duration        = 60,  
                 methods         = [[:simulated_annealing 1];  
                                   [[:iterated_local_search 1];  
                                   [[:randomized_first_improvement 1];]])
```

Short [practical examples](#) at `github.com/phrb/StochasticSearch.jl`:

- Rosenbrock
- Sorting Algorithm Cutoff
- Travelling Salesperson Problem
- [NVCC Flags](#)

# AUTOTUNING NVCC PARAMETERS IN THE JULIA LANGUAGE

---

Pedro Bruel and Alfredo Goldman

*{phrb, gold}@ime.usp.br*

August 24, 2017



Instituto de Matemática e Estatística  
Universidade de São Paulo