Cuketa
MiniRocket implementation in CUDA

Antonín Kříž
NI-GPU @ FIT ČVUT

May 2024

1 Introduction

MiniRocket [1] is one of many algorithms in the Rocket [2] family. Rocket, unlike most other SOTA algorithms for Time Series Classification uses a new approach compared to other models like TS-CHIEF [3], based on forests, Inception Time [4], based on CNNs, or HIVE-COTE (2) [5, 6], which ensemble multiple models. The main goal of Rocket algorithms isn't to beat other models in the classification performance, but rather to (at least) match it while offering substantial speed-ups. This is because training and testing time combined on the UCR Time Series Archive [7] set of datasets is 6 days for Inception Time, almost 2 weeks for TS-CHIEF and possibly one order of magnitude more for HIVE-COTE on the setup used by A. Dempster et al. in [2]. Rocket manages to take this time down to 1 hour 50 minutes while MiniRocket to 8 minutes [1] in case of the original Python + Numba[1] implementation to less than a minute in case of the Julia implementation present in my Bachelor's thesis[2] [8].

1.1 Rocket

The idea behind Rocket is already hidden in it's name, or rather abbreviation: ***R**and**O**m **C**onvolutional **KE**rnel **T**ransform.* As the name already suggests, Rocket utilises convolution using random kernels to transform the time series data. It does not do the classification and is purely a data pre-processing step, before feeding the data into traditional linear classifiers like Ridge or Logistic regression (the later is preferred for larger datasets) [2]. Using convolution isn't a new technique and Rocket's the approach can be roughly compared to CNNs. The main difference is that, unlike CNNs, which use small set of convolutional filter with carefully trained weights and other parameters like their length[3], dilation, bias and padding, Rocket generates thousands (10000 by default) of different kernels with randomly selected parameters as in Figure 1, utilising the

[1]`https://numba.pydata.org/`
[2]In Czech language only.
[3]Time series are (usually) 1-dimensional data.

strength in numbers. Since this removes the need to carefully train these kernels, only the fast and simple convolution step remains. Each kernel is applied to the time series and from these values the maximum value and the *PPV* - Prevalence of Positive Values - is extracted, similarly as in pooling layers in CNNs. These extracted values are used as features for the final linear classifier model.

I'VE TAKEN A CONTROVERSIAL NEW PILL THAT ACCELERATES MY BRAIN.
SO YOU'RE SMART NOW?
I'M STUPID FASTER.
I'M DOING 1000 CALCULATIONS PER SECOND AND THEY'RE ALL WRONG
√5=5
5+6=9
2×11=27
5²=52
SHENCOMIX.COM

Figure 1: Stupid Faster [9, 10]

1.2 MiniRocket

MiniRocket, fully ***MINI**mally* ***R**and**O**m* ***C**onvolutional* ***KE**rnel* ***T**ransform*, focuses on removing randomness from the Rocket algorithm, while reducing the processing time even further. Instead of generating thousands kernels, MiniRocket uses a fixed set of 84 kernel weights (as proven in Theorem 1.1) of length 9, built purely from six values of -1 and three values of 2.

Theorem 1.1. *There are 84 kernel weights combinations used in MiniRocket.*

Proof. Let 9 be the length of the kernel and 6 be the number of values set to -1 and 3 be the number of values set to 2. Then, by choosing 3 spots in the kernel, we get $\binom{9}{3} = 84$ unique kernel weights combinations. □

Simply said, the dilation and it's intensity is based on the input length and the number of different dilations per each kernel (combination of -1 and 2) is based on the number of features requested. Padding is enabled (or disabled) for every other feature and bias is based on quantiles of the values calculated in the convolution par of the training step. These quantiles are not random but based on *golden-ratio-modulo-one* series, providing low discrepancy series for the whole $[0, 1]$ interval.

	Rocket	MiniRocket
length	$\{7, 9, 11\}$	9
weight	$\mathbb{N}(0, 1)$	$\{-1, 2\}$
bias	$\mathbb{U}(-1, 1)$	based on convolution
dilatation	random	fixed
padding	{yes, no}	deterministic
features	PPV, MAX	PPV

Table 1: Rocket a MiniRocket comparison

Thanks to this approach, MiniRocket does not even really need to calculate every single of these convolutions, but it can just add -1 and 2-multiples of the input values, shifted by dilation and padding. This means that these multiples can be computed only once for each time series and then reused for every weights-dilation-padding-bias combination. Let α be the -1-multiple and β the 2-multiple of the input time series of length n. For kernel with dilation 1 and the 2-values in position 2 (and two other), we can construct a matrix C'

$$C' = \begin{bmatrix} 0 & 0 & 0 & 0 & \alpha_1 & \cdots & \alpha_{n-4} \\ 0 & 0 & 0 & \beta_1 & \beta_2 & \cdots & \beta_{n-3} \\ 0 & 0 & \alpha_1 & \alpha_2 & \alpha_3 & \cdots & \alpha_{n-2} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \alpha_4 & \alpha_5 & \alpha_6 & \alpha_7 & \alpha_8 & \cdots & 0 \\ \alpha_5 & \alpha_6 & \alpha_7 & \alpha_8 & \alpha_9 & \cdots & 0 \end{bmatrix},$$

which, when columns are summed, is equal to the convolution output. These calculations can be simplified even further. Let C_α be a vector of C' column sums for kernel which is made purely of -1-multiples. We can, based on the fact that -1 + 3 = 2, make C_γ, a vector made out of sum of 2-multiple rows from C'. We can keep C_α for each dilation and re-calculate C_γ only based on the location of the thee 2-multiple rows in C'. This way we can get rid of all multiplications in the algorithm (we just need to flip the sign for α and do two additions for β in the beginning) and do only $\frac{3}{9}$ sums out of the 9 otherwise required for each weights-dilation combination.

Finally, MiniRocket uses only *PPV* as it's pooling function since, according to benchmarks [1] mentioned in the original paper, the classification accuracy decrease is statistically insignificant, while reducing the memory requirements by half and further increasing the performance.

2 Cuketa 🥒

Introducing Cuketa 🥒 [4] /sʊketa/, ***CU**DA accelerated minimally random convolutional **KE**rnel **T**r**A**nsform.* The implementation presented in this project, would not be feasible considering the original implementation of MiniRocket. Thanks to inefficient parallelisation and repeated memory allocations (although understandable considering limitations of Python + Numba), the process of conversion to efficient CUDA code would be more challenging and possibly way less efficient thanks to frequent cache misses. Thus, this thesis is based on the implementation that can be found in TimeSeriesClassification.jl[5] [8].

The CUDA implementation is nearly identical, considering the algorithm itself, to the Julia implementation. The main difference is that there are almost no CPU computations. Almost all operations happen on the GPU through multiple CUDA kernels on pure CUDA allocated arrays - no other data structures were required. The exception is the `fit_dilations` function, which prepares the dilations and calculates the number of final features per dilation.

2.1 Kernels and their optimisation

Most CUDA kernels are written in a way to reduce the number of CUDA kernel calls from the host. For example this Julia code:

```julia
for idx in 1:(9÷2)
    d = e + dilation * (idx - 1)
    @turbo (@views C[end-d+1:end]) .+= (@views A[begin:d])
end

for idx in (9÷2)+2:9
    d = s + dilation * (idx - ((9 ÷ 2) + 2))
    @turbo (@views C[begin:end-d+1]) .+= (@views A[d:end])
end

for idx in @views INDICES[:, kernel_index]
    if idx < 5
        d = e + dilation * (idx - 1)
        @turbo (@views C[end-d+1:end]) .+= (@views G[begin:d])
    elseif idx > 5
        d = s + dilation * (idx - ((9 ÷ 2) + 2))
        @turbo (@views C[begin:end-d+1]) .+= (@views G[d:end])
    else
```

[4]This emoji is based on Twemoji [11], an open-source emoji set created by Twitter.
[5]`https://github.com/antoninkriz/TimeSeriesClassification.jl`

```
        @turbo @views C .+= G
    end
end
```

was replaced with just two CUDA kernel calls:

```
C_A_add<<<BLOCK_SIZE(input_length, BLOCK_SIZE_C_A)>>>(
    C.get(), A.get(),  input_length, dilation, s, e
);

const auto &[ind0, ind1, ind2] = INDICES[kernel_index];
C_G<<<BLOCK_SIZE(input_length, BLOCK_SIZE_C_G)>>>(
    C.get(), G.get(),  input_length, dilation, s, e,
    ind0, ind1, ind2
);
```

and all arithmetic operations and the loops are run on the GPU. The host code is purely managing what operations are being done on the GPU and the GPU in return does not need to do any decision making. *PPV* reductions are done using optimised parallel sum from [12]. The reduction leaves some space for more advanced (or rather optimised) reductions, since instead of a recursive reduction, as highlighted in [13], it uses just `atomicSum(...)`, which should still be quite fast since we're not using the result of the call. This implementation is therefore balancing performance, readability and my free time.

What could be really optimised is the block size (thread count) for each CUDA kernel. Kernels which are similar (in terms what they do) are using the same block size. This lead to four different block sizes used in the code:

- `BLOCK_SIZE_INIT`
  Used for initialising array values from the input time series and generating quantiles from the *golden-ratio-modulo-one* series.

- `BLOCK_SIZE_C_XXX`
  Used for operations over the arrays based on the C' matrix, i.e. the `C_A_add` and `C_G` CUDA kernels.

- `SIZE_QUANTILE_BIASES`
  Used purely for extracting biases based on quantiles in the training phase.

- `BLOCK_SIZE_PPV_PX`
  Used for calculating PPV in the transformation phase.

To efficiently optimise the block size I used an input of 10 time series with length of 100 000 with randomly generated input data. This is OK, since the algorithm is not dependent on the input values themselves, just the sizes of the input. I compiled the program with each block size being a power of two, ranging from 128 to 1024. This gave me $4^4 = 256$ different versions of the same program. Each version was then ran with its run time averaged over 3 runs. The best results were with following sizes

- `BLOCK_SIZE_INIT` $= 512$
- `BLOCK_SIZE_C_XXX` $= 256$
- `SIZE_QUANTILE_BIASES` $= 512$
- `BLOCK_SIZE_PPV_PX` $= 256$

as can be seen in Attachment 4.1, considering only the run time of the algorithm and ignoring the time of the initial data transfer from the host to the device. This is OK for estimating block sizes since here we care only about the algorithm itself and not the fact that it's running on a GPU.

2.2 Benchmarks

Finally we're getting to the part where the CUDA implementation is compared to Python and Julia implementations. The inputs are randomly generated arrays of dimension $N \times M$ (or rather $M \times N$ in case of column-major arrays in Julia), where N is the number of examples and M is the length of a time series, with following values:

- $N = 100, M = 10000$
- $N = 100, M = 100000$
- $N = 100, M = 1000000$

and the test was run 3 times for all values together and the total time (in case of CUDA implementation including the host-to-device transfer time). CUDA version used block sizes from the Section 2.1.

The testing was done on a computer with following hardware and software in the list bellow:

- `CPU`: Intel i7 11700
- `GPU`: NVIDIA RTX 3070
- `RAM`: 80 GB DDR4 3333MHz@CL17
- `Linux`: Manjaro with Kernel 6.6.26
- `CUDA`: cuda_12.3.r12.3/compiler.33567101_0
- `Julia`: 1.10.0
- `TimeSeriesClassification.jl`: 1.0.0
- `Python`: 3.11.8
- `sktime`: 0.29.0
- `numpy`: 1.26.4
- `numba`: 0.59.1

2.3 Results

As can be seen on the Table 2 and the Figure 2, the CUDA implementation is much slower for time series of short lengths. This is expected due to the need of repeated CUDA kernel calls, which happen to be much slower than the locally running code, still possibly present in the CPU cache. This difference is even more significant with larger amount of short time series, since non-CUDA implementations can process multiple examples (time series) using multiple threads thanks to multi-threading. On the other hand the CUDA implementation is at least 2 times faster for data of large length since the performance of the GPU relies on parallelisation per the number of observation of each time series.

implementation / time series length	10 000	100 000	**1 000 000**
Cuketa 🥒	1 851	3 274	**27 515**
TimeSeriesClassification.jl - 8 threads	502	4 940	**57 710**
TimeSeriesClassification.jl - 1 thread	918	9 100	**92 917**
sktime - 8 threads	857	13 082	**314 905**

Table 2: Run time in milliseconds per implementation and time series length (per 100 examples)

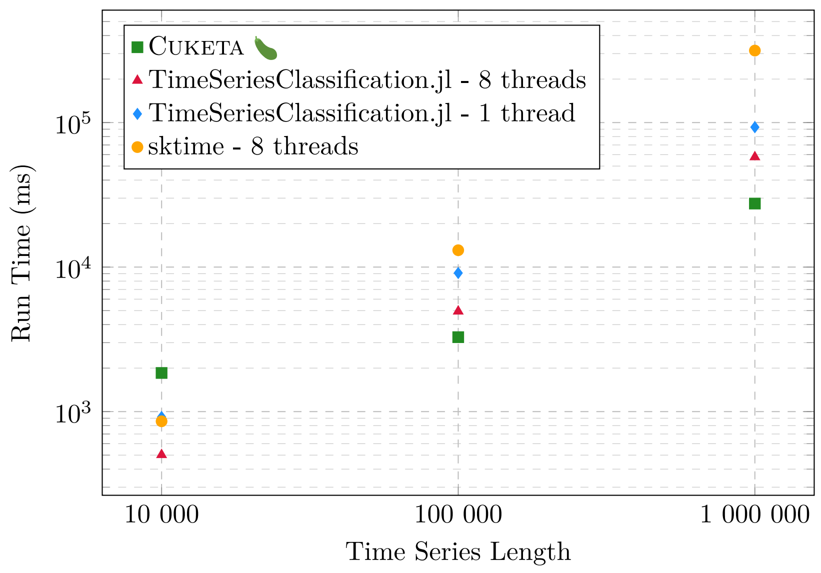

Figure 2: Benchmarked run times per implementation and time series length (per 100 examples)

3 Conclusion

Cuketa, the CUDA accelerated implementation of MiniRocket, performs much better on time series of large lengths, while current implementations benefit from large number of shorter time series. This was expected due to the way each algorithm utilises parallelisation. When time series length is of at least 100 000 observations, Cuketa tends to perform 1.5× faster than the fastest CPU implementation of the MiniRocket algorithm and 4× faster than the official implementation. For a time series with 1 000 000 observation Cuketa is at least 2× and 11.4× faster respectively.

References

[1] Angus Dempster, Daniel F. Schmidt, and Geoffrey I. Webb. "MiniRocket: A Very Fast (Almost) Deterministic Transform for Time Series Classification." In: *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. KDD '21. ACM, Aug. 2021. DOI: 10.1145/3447548.3467231. URL: http://dx.doi.org/10.1145/3447548.3467231.

[2] Angus Dempster, François Petitjean, and Geoffrey I. Webb. "ROCKET: exceptionally fast and accurate time series classification using random convolutional kernels." In: *Data Mining and Knowledge Discovery* 34.5 (July 2020), pp. 1454–1495. ISSN: 1573-756X. DOI: 10.1007/s10618-020-00701-z. URL: http://dx.doi.org/10.1007/s10618-020-00701-z.

[3] Ahmed Shifaz et al. "TS-CHIEF: a scalable and accurate forest algorithm for time series classification." In: *Data Mining and Knowledge Discovery* 34.3 (Mar. 2020), pp. 742–775. ISSN: 1573-756X. DOI: 10.1007/s10618-020-00679-8. URL: http://dx.doi.org/10.1007/s10618-020-00679-8.

[4] Hassan Ismail Fawaz et al. "InceptionTime: Finding AlexNet for time series classification." In: *Data Mining and Knowledge Discovery* 34.6 (Sept. 2020), pp. 1936–1962. ISSN: 1573-756X. DOI: 10.1007/s10618-020-00710-y. URL: http://dx.doi.org/10.1007/s10618-020-00710-y.

[5] Jason Lines, Sarah Taylor, and Anthony Bagnall. "Time Series Classification with HIVE-COTE: The Hierarchical Vote Collective of Transformation-Based Ensembles." In: *ACM Trans. Knowl. Discov. Data* 12.5 (July 2018). ISSN: 1556-4681. DOI: 10.1145/3182382. URL: https://doi.org/10.1145/3182382.

[6] Matthew Middlehurst et al. *HIVE-COTE 2.0: a new meta ensemble for time series classification*. 2021. arXiv: 2104.07551 [cs.LG].

[7] Hoang Anh Dau et al. *The UCR Time Series Archive*. 2019. DOI: 10.48550/arXiv.1810.07758. arXiv: 1810.07758 [cs.LG]. URL: https://arxiv.org/abs/1810.07758 (visited on 04/10/2023).

[8] Antonín Kříž. "Time Series Classification in Julia." Bachelor's Thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023. URL: http://hdl.handle.net/10467/109353.

[9] Shen Comix. *Stupid Faster*. Sept. 2019. URL: https://www.instagram.com/p/B2MjCGlAaOz.

[10] Shen Comix. *Stupid Faster - bonus*. Sept. 2019. URL: https://www.instagram.com/p/B2NNOGkAAgK.

[11] Twitter. *twemoji: Emoji for everyone. https://twemoji.twitter.com/.* URL: https://github.com/twitter/twemoji.

[12] Mark Harris. *Optimizing Parallel Reduction in CUDA*. 2008. URL: https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf.

[13] Mark Harris, Shubhabrata Sengupta, and John D. Owens. *Chapter 39. Parallel Prefix Sum (Scan) with CUDA — developer.nvidia.com*. 2007. URL: `https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda`.

4 Attachments

4.1 Run time per block sizes

- `B1` = `BLOCK_SIZE_INIT`
- `B2` = `BLOCK_SIZE_C_XXX`
- `B3` = `SIZE_QUANTILE_BIASES`
- `B4` = `BLOCK_SIZE_PPV_PX`
- `duration` = run time of the whole algorithm excluding copying data from the host to the device in milliseconds

B1	B2	B3	B4	duration
512	256	512	256	489.333333
512	256	1024	128	490.333333
512	128	1024	256	490.666667
256	256	512	128	490.666667
512	256	256	256	490.666667
512	128	512	128	491.000000
256	256	1024	128	491.000000
256	256	128	256	491.000000
256	256	1024	256	491.333333
256	256	256	128	491.333333
256	256	256	256	491.333333
256	256	512	256	491.333333
512	256	1024	256	491.666667
512	256	128	128	491.666667
512	128	256	128	491.666667
512	256	512	128	491.666667
256	256	128	128	491.666667
512	128	256	256	492.000000
256	512	512	256	492.333333
512	256	128	256	492.333333
512	128	512	256	492.333333
512	256	256	128	492.666667
512	128	128	256	492.666667
512	128	1024	128	492.666667
512	128	128	128	493.000000
512	512	256	256	493.000000
256	512	128	256	493.000000
256	512	1024	256	493.333333
512	512	128	128	493.333333
512	512	128	256	493.333333
256	512	512	128	493.666667
512	512	512	128	493.666667
256	512	256	256	493.666667
256	512	256	128	494.000000
512	512	256	128	494.333333
128	256	1024	256	494.333333
256	512	128	128	494.333333
128	256	256	256	494.666667
128	256	1024	128	494.666667

B1	B2	B3	B4	duration
128	256	512	256	495.000000
512	512	512	256	495.000000
128	128	1024	256	495.000000
512	512	1024	256	495.333333
128	256	128	256	495.333333
256	512	1024	128	495.333333
128	128	512	256	495.666667
512	512	1024	128	495.666667
128	128	128	128	495.666667
256	128	128	128	495.666667
128	256	256	128	496.000000
128	256	128	128	496.000000
128	128	512	128	496.333333
128	512	128	256	496.333333
256	128	128	256	496.333333
128	128	128	256	496.333333
256	128	256	128	496.666667
128	128	256	256	496.666667
256	128	512	128	496.666667
128	128	256	128	497.000000
128	512	1024	256	497.000000
256	128	256	256	497.000000
256	128	1024	128	497.000000
256	128	1024	256	497.000000
128	128	1024	128	497.333333
1024	128	1024	256	497.333333
128	512	256	256	497.666667
1024	512	512	256	497.666667
1024	512	256	256	498.000000
128	512	128	128	498.000000
256	128	512	256	498.000000
128	256	512	128	498.000000
128	512	512	128	498.333333
1024	256	128	256	498.333333
128	512	1024	128	498.333333
128	512	256	128	498.333333
1024	256	256	256	498.666667
128	512	512	256	498.666667
1024	128	256	256	498.666667
1024	512	512	128	499.000000
1024	128	128	128	499.000000
1024	128	1024	128	499.333333
1024	256	1024	256	499.666667
1024	256	512	256	499.666667
1024	128	128	256	499.666667
1024	512	256	128	500.000000
1024	256	256	128	500.000000
1024	256	1024	128	500.333333
1024	256	128	128	500.333333
1024	128	512	256	500.666667
1024	512	128	256	501.333333
1024	128	512	128	501.333333
1024	256	512	128	502.000000
1024	128	256	128	502.000000
1024	512	1024	256	502.333333
1024	512	1024	128	503.000000
1024	512	128	128	503.333333

B1	B2	B3	B4	duration
512	1024	1024	256	510.000000
512	1024	512	256	510.333333
1024	1024	1024	128	510.666667
512	1024	256	256	511.666667
512	1024	1024	128	511.666667
512	1024	128	256	511.666667
1024	1024	1024	256	511.666667
512	1024	256	128	512.333333
512	1024	128	128	512.666667
1024	1024	128	128	513.000000
1024	1024	128	256	513.000000
512	1024	512	128	514.000000
256	1024	128	256	514.000000
256	1024	256	256	514.666667
128	1024	256	256	514.666667
256	1024	512	128	514.666667
128	1024	128	256	514.666667
128	1024	1024	256	514.666667
256	1024	128	128	515.000000
256	1024	1024	256	515.000000
256	1024	512	256	515.000000
128	1024	512	256	515.000000
256	1024	256	128	515.333333
256	1024	1024	128	515.666667
128	1024	128	128	515.666667
128	1024	256	128	515.666667
128	1024	512	128	516.000000
128	1024	1024	128	516.000000
1024	1024	256	128	516.333333
1024	1024	512	256	517.000000
1024	1024	256	256	517.333333
1024	1024	512	128	518.333333
256	256	512	512	533.333333
512	256	256	512	534.000000
256	256	1024	512	534.666667
512	256	128	512	534.666667
256	256	128	512	535.000000
512	128	1024	512	535.333333
512	256	512	512	535.333333
512	256	1024	512	535.666667
512	128	128	512	535.666667
512	512	128	512	536.000000
256	256	256	512	536.000000
256	512	512	512	536.000000
512	128	512	512	536.000000
512	128	256	512	536.333333
256	512	256	512	536.666667
256	128	512	512	536.666667
256	512	128	512	537.000000
512	512	1024	512	538.000000
128	256	128	512	538.000000
512	512	512	512	538.000000
256	512	1024	512	538.333333
256	128	1024	512	538.666667
128	256	256	512	539.000000
128	128	512	512	539.000000
128	128	256	512	539.333333

B1	B2	B3	B4	duration
512	512	256	512	539.333333
128	256	512	512	539.333333
128	128	1024	512	539.333333
256	128	256	512	539.666667
128	256	1024	512	539.666667
256	128	128	512	539.666667
128	128	128	512	540.000000
128	512	1024	512	540.666667
128	512	128	512	541.000000
1024	512	256	512	542.000000
128	512	512	512	542.333333
1024	512	512	512	542.333333
1024	128	1024	512	542.333333
128	512	256	512	542.666667
1024	256	128	512	542.666667
1024	256	512	512	542.666667
1024	256	1024	512	543.000000
1024	128	256	512	543.666667
1024	128	512	512	544.000000
1024	128	128	512	544.000000
1024	256	256	512	544.333333
1024	512	128	512	545.333333
1024	512	1024	512	546.333333
512	1024	1024	512	554.333333
512	1024	512	512	554.666667
512	1024	256	512	555.000000
512	1024	128	512	555.333333
128	1024	128	512	556.333333
1024	1024	1024	512	556.333333
128	1024	256	512	557.333333
128	1024	1024	512	558.000000
256	1024	128	512	558.666667
256	1024	1024	512	558.666667
128	1024	512	512	558.666667
256	1024	512	512	558.666667
256	1024	256	512	559.000000
1024	1024	128	512	559.666667
1024	1024	256	512	560.333333
1024	1024	512	512	560.666667
256	256	128	1024	678.000000
512	256	256	1024	678.666667
512	256	128	1024	678.666667
256	256	512	1024	679.333333
512	256	1024	1024	679.666667
512	256	512	1024	680.333333
256	256	256	1024	680.333333
512	128	512	1024	680.666667
256	256	1024	1024	680.666667
512	128	128	1024	680.666667
512	512	1024	1024	681.333333
256	512	128	1024	681.333333
512	512	128	1024	681.666667
512	128	1024	1024	681.666667
512	128	256	1024	682.000000
512	512	512	1024	682.000000
256	512	256	1024	682.333333
256	512	1024	1024	683.000000

B1	B2	B3	B4	duration
128	128	512	1024	683.666667
128	256	512	1024	683.666667
512	512	256	1024	683.666667
256	512	512	1024	683.666667
128	256	128	1024	684.000000
256	128	128	1024	684.000000
128	128	128	1024	684.666667
128	256	256	1024	684.666667
128	256	1024	1024	684.666667
128	128	256	1024	684.666667
128	128	1024	1024	685.000000
256	128	512	1024	685.000000
256	128	1024	1024	685.666667
128	512	256	1024	686.333333
256	128	256	1024	686.333333
128	512	512	1024	686.666667
128	512	1024	1024	686.666667
1024	128	1024	1024	687.000000
1024	128	128	1024	687.333333
1024	512	512	1024	687.666667
128	512	128	1024	687.666667
1024	256	1024	1024	689.666667
1024	256	256	1024	689.666667
1024	256	128	1024	689.666667
1024	256	512	1024	690.000000
1024	128	512	1024	690.333333
1024	128	256	1024	691.000000
1024	512	1024	1024	691.666667
1024	512	128	1024	692.000000
1024	512	256	1024	692.333333
512	1024	256	1024	698.333333
512	1024	1024	1024	698.666667
1024	1024	128	1024	699.666667
512	1024	512	1024	700.000000
512	1024	128	1024	700.666667
256	1024	1024	1024	701.333333
128	1024	256	1024	702.333333
128	1024	512	1024	703.333333
256	1024	512	1024	703.333333
128	1024	1024	1024	703.333333
256	1024	128	1024	704.000000
256	1024	256	1024	704.333333
128	1024	128	1024	704.666667
1024	1024	256	1024	705.000000
1024	1024	512	1024	706.000000
1024	1024	1024	1024	748.333333