NFGen: Automatic Non-linear Function Evaluation Code Generator for General-purpose MPC Platforms

这次介绍的论文是清华团队Xiaoyu Fan等人发表在CCS'22 NFGen。

本文针对MPC下的非线性函数计算，提出了NFGen工具包。NFGen利用离散分段多项式，自动化综合考虑后端MPC协议的开销、近似精度等指标，生成最优的近似多项式，并利用后端的MPC协议库进行计算（例如PrivPy，MP-SPDZ等）。相比于之前手动在MPC源码下直接编写近似多项式，NFGen在精度、性能等方面都有长足提升。

1. 问题与挑战

目前MPC在理论上虽然能够支持任意函数的计算，但是在计算复杂非线性函数，比如e^x，$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$等，还存在很多问题：

1. **正确性与精度**：为了效率考虑，目前大多数MPC协议还是需要将浮点数（FLP）转化为定点数（FXP）进行计算。该转化不仅会引起精度的损失，在一些极端的情况下会带来溢出等错误；
2. **性能**：更重要的，MPC协议的计算性能比对应的明文计算低好几个数量级，尤其是针对复杂的非线性函数；
3. **泛化性**：即便我们可以用MPC协议内置支持的一些基本操作实现很多非线性函数，但是有些复杂的函数比如$\gamma(x, z)$等还是很难去实现；
4. **可移植性**：MPC协议底层技术多样，针对不同的场景硬件环境等各种权衡复杂。如何设计面向多样化场景的最优方案是个难题。

针对上述难题，本文提出了NFGen。NFGen利用最基本的算子$+, \times, >$来实现多项式近似。由于这些基本算子被广泛支持，因此可以很容易对接不同的MPC协议库。

挑战：针对一个非线性函数，获得一个FXP下面向MPC协议库的良好(k, m)-分段近似多项式是本文面临的最大技术挑战，其中m表示分段，k表示分段多项式最大的阶数（degree）：

1. **FXP下近似多项式的挑战**：1）首先多项式需要满足FXP的值域和放缩限制，保证计算中间结果不溢出（上溢出、下溢出）；2）其次，FXP实际上是整数，寻找一个最优的多项式是一个NP-完全的整数规划问题。因此，本文需要寻找一个合理的近似。
2. **m和k的权衡**：m越大，则分段越多，因此需要的比较$>$也越多；k越大，则需要的乘法$\times$也越多。

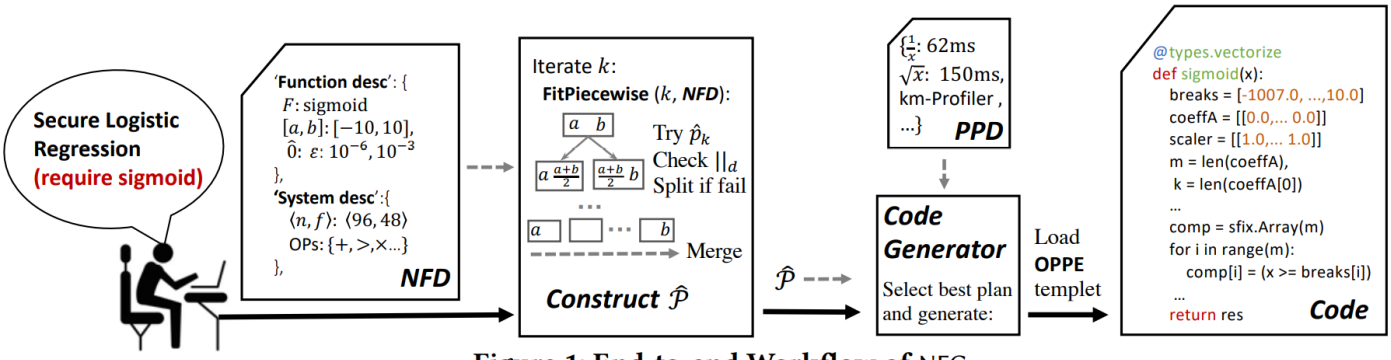

Figure 1: End-to-end Workflow of NFGen

如上图所示，NFGen工作流程如下：

1. 给定函数和系统配置文件**NFD**，NFGen首先在明文下计算函数得到候选定点数函数簇$\hat{\mathcal{P}}$；
2. 进一步，根据MPC性能文件**PPD**从$\hat{\mathcal{P}}$选取性能最优的函数；
3. 最后，利用生成代码模板**OPPE**，并生成后端MPC协议对应的代码。

2. 方案设计

符号和假设：令$F(x)$表示目标浮点数连续函数，$[a,b]$表示近似区间；候选近似定点数多项式簇$\hat{\mathcal{P}} = \{\hat{p}_k^m\}$，其中$\hat{p}_k^m$表示该多项式在$[a,b]$上分为$m$段$[w_0, w_1, \ldots, w_m](w_0 = a, w_m = b)$，每一段$[w_{j-1}, w_j]$有近似多项式$\hat{p}_k^j(\hat{x}) = \sum_{i=0}^{k} \hat{c}_i \hat{x}^i$。当$j$不太重要时，本文忽略$j$的描述。$\hat{}$表示$\langle n, f \rangle$-FXP定点数，其中$n$为全部比特位长，$f$为小数部分比特位长。

2.1 非线性近似

非线性近似模块分为FitPiecewise和FitOnePiece两个主要模块。

2.1.1 FitPiecewise

如算法1所示，该模块采用递归的方法对每一段调用FitOnePiece进行近似。如果当前分段无法达到近似精度，则进对当前分段进一步分为小段进行近似。最后的结果，为了减少分段数量，则尝试将相邻的分段合并。

2.1.2 FitOnePiece

该部分是NFGen近似算法的核心，目标是针对一段进行多项式近似。给定$F(x)$，近似定点数多项式$\hat{p}_k(x) = \sum_{i=0}^{k} \hat{c}_i \hat{x}^i$的目标是

$$\text{Minimize } \max_{\hat{x} \in [a,b]} |F(\hat{x}) - \hat{p}_k(x)|$$

Algorithm 2: FitOnePiece Algorithm

Input : Target function $F(x)$, domain $[a, b]$ and order k.
Return: Feasible discrete polynoimial $\hat{p}_k$ or `Null`.

$\bar{k} \leftarrow$ ConstrainK$([a, b], \langle n, f \rangle)$ /* **1) Constrain k** */

/* **2) Fit best polynomial in FLP space** */

Maximum representable points N $\leftarrow \frac{b-a}{2^{-f}}$;

if N $> \bar{k} + 1$ **then**

  $p_{\bar{k}} \leftarrow$ *Cheby-Interpolation*$(F, [a, b], \bar{k})$

else

  $\bar{k} = N - 1$;

  $p_{\bar{k}} \leftarrow$ *Lagrange-Interpolation*$(F,$ N feasible points and $\bar{k})$

end

/* **3) & 4) Convert to FXP space** */

$\hat{p}_{\bar{k}} \leftarrow$ ScalePoly$(p_{\bar{k}}, [a, b])$ (Algo 5) ;

$\hat{p}_{\bar{k}} \leftarrow$ ResidualBoosting$(\hat{p}_{\bar{k}}, F, [a, b])$ (Algo 6) ;

$\hat{p}_k$: Expand coefficients and scaling factors of $\hat{p}_{\bar{k}}$ to k, filling 0 ;

/* **5) Check accuracy, return valid $\hat{p}_k$ or Null** */

Sampled number $\mathrm{N_s} \leftarrow \min(MS, \mathrm{N})$;

$\hat{\mathrm{X}} \leftarrow$ FLPsimFXP(Linspace$([a, b], \mathrm{N_s})$) ;

if $\max_{\hat{x} \in \hat{\mathrm{X}}} |\hat{p}_k(\hat{x}) - F(\hat{x})|_d < \epsilon$ **then**

  Return: $\hat{p}_k$;

Return: `Null` ;

在算法2的近似计算中，首先需要解决确定合适的k，以尽可能避免溢出问题：在$|\hat{x}|$过大时，$\hat{x}^k$可能会上溢出，在$|\hat{x}| \to 0$过小时，$\hat{x}^k$可能会下溢出，尤其是k过大时。为了解决该问题，本文提出了ContrainK算法。该方法的核心思想如下：

1. 为了防止上溢出，需要在$|\hat{x}|_{max} > 1$时保证$(|\hat{x}|_{max})^{kO} \leq 2^{n-f-1} \Rightarrow k_O \leq \frac{n-f-1}{\log_2 |\hat{x}|_{max}}$;
2. 下溢出的情况则复杂一些：如果$0 \in [a, b]$，那么存在$|\hat{x}| \to 0$。此时，令$k_U = 3$；否则当$|\hat{x}|_{min} < 1$时，需要满足$|\hat{x}|_{min}^{k_U} \geq 2^{-f} \Rightarrow k_U \leq \frac{f}{-\log_2 |\hat{x}|_{min}}$。最后$\bar{k} == \min(k, k_O, k_U)$具体算法如下：

Algorithm 3: ConstrainK

Function ConstrainK(*domain* $[a, b]$, *FXP format* $\langle n, f \rangle$):

$|\hat{x}|_{max} \leftarrow \max(|a|, |b|)$ and $|\hat{x}|_{min} \leftarrow \min(|a|, |b|)$;

$k_O \leftarrow k$ if $(|\hat{x}|_{max} < 1)$ else $\frac{n-f-1}{\log_2(|\hat{x}|_{max})}$;

If $a \cdot b < 0$ **then** $k_U \leftarrow 3$;

Else $k_U \leftarrow k$ if $(|\hat{x}|_{min} > 1)$ else $\frac{f}{-\log_2(|\hat{x}|_{min})}$;

Return: Maximum feasible $\bar{k} \leftarrow \min(k, k_O, k_U)$

确定$\bar{k}$之后，则在$[a, b]$内进行采点，进行插值计算得到浮点数近似多项式，进而将浮点数近似多项式转化为定点数。转化时则会面临如下两个挑战：

系数$\hat{c}_i$下溢出：当系数$|\hat{c}_i|$过小时，需要在f比特的高位部分保留过多的0，损失了高位的比特，而且当$|\hat{c}_i| < 2^{-f}$会引起下溢出。更严重的当$i \to k$时，$|c_i|$可能越小；

近似精度损失：另一方面，将浮点转化为定点数，舍弃$f+1$位及其之后的小数部分，会损失大量精度。

为了尽可能解决上述问题，NFGen提出了如下策略：

利用放缩因子增大表示范围：为了缓解系数下溢出问题，本文提出将浮点数系数c_i转化为两个定点数$(\hat{c}_i, \hat{s}_i)$使得$c_i \approx \hat{c}_i \hat{s}_i$，其中$\hat{s}_i \leq 1$是一个放缩因子。从而使得$\hat{c}_i$保留尽可能多的高比特位。不过，本文也需要保证：

1. $\hat{c}_i$不能太大，以免$\hat{c}_i \hat{x}^k$上溢出，即$\hat{c}_i \hat{x}^k < 2^{n-f-1}$；
2. $\hat{s}_i$是一个合法的FXP且$0 < \hat{s}_i \leq 1$。

具体算法如下：

Algorithm 5: ScalePoly : $p_k \rightarrow \hat{p}_k$

Input : $p_k = \sum_{i=0}^{i=k} (c_i \cdot x^i)$ in continuous space, domain $[a, b]$.
Return: $\hat{p}_k = \sum_{i=0}^{i=k} (\hat{c}_i \cdot x^i \cdot \hat{s}_i)$ in discrete space.

Character $\hat{x} \leftarrow \max(|a|, |b|)$, most likely to overflow ;
for $i \leftarrow 0$ **to** k **do**
  $\hat{c}_i, \hat{s}_i \leftarrow \text{ScaleC}(c_i, \langle n, f \rangle, i, \hat{x})$;
end
Return $\hat{p}_k$.
Function ScaleC(c, $\langle n, f \rangle$, order k and $\hat{x}$):
  $\hat{s}_{CUF} \leftarrow 2^{-f}$;
  $\hat{s}_{COF} \leftarrow \text{FLPsimFXP}(\frac{c\hat{x}^k}{2^{n-f-1}}, n, f)$;
  $\hat{s} \leftarrow \min\{\max(\hat{s}_{CUF}, \hat{s}_{COF}), 1\}$;
  $\hat{c} \leftarrow \text{FLPsimFXP}(\frac{c}{\hat{s}}, n, f)$;
  Return $\hat{c}, \hat{s}$;

其中FLPsiimFXP如下:

Algorithm 4: FLPsimFXP : $x \rightarrow \hat{x}$

Input : $x \in \mathbb{R}$ and FXP format $\langle n, f \rangle$.
Return: $\hat{x}$

If $(|x| > 2^{n-f-1})$: **Return:** 2^{n-f-1} ;
If $(|x| < 2^{-f})$: **Return:** 0 ;
Return: $\text{round}_2(x, f)$

利用残差提升精度：对于目标函数和近似函数$\hat{p}_k(x)$，定义残差函数$R(x) = F(x) - \hat{p}_k(x)$。进一步，本文再利用多项式$\hat{r}_{k'}(x)$近似估计$R(x)$，并进一步更新$\hat{p}_k(x)$为$\hat{p}_k(x) + \hat{r}_{k'}(x)$，其中$k' < k$。具体算法构造如下:

Algorithm 6: ResidualBoosting

Input : $\hat{p}_k$ in discrete space, target $F(x)$ and domain $[a, b]$.
Return: $\hat{p}_k^*$ in discrete space.

$\hat{p}_k^* \leftarrow \hat{p}_k, R \leftarrow F - \hat{p}_k^*$;
Sample number $\mathrm{N_s} \leftarrow$ min (Max samples MS, All points $\frac{b-a}{2^{-f}}$) ;
$\hat{\mathrm{X}} \leftarrow \mathrm{FLPsimFXP}(\mathrm{Linspace}([a, b], \mathrm{N_s}))$;
for $k' \leftarrow k - 1$ **to** 0 **do**
    $r_{k'} \leftarrow$ *Cheby-Interpolation*$(R, [a, b], k')$;
    $\hat{p}_k^{tmp} \leftarrow \mathrm{Boost}(\hat{p}_k^*, r_{k'}, [a, b])$;
    /* Boost when benefit exist. */
    if $(\max_{\hat{x} \in \hat{\mathrm{X}}} |F(\hat{x}) - \hat{p}_k^{tmp}(\hat{x})|_d < \max_{\hat{x} \in \hat{\mathrm{X}}} |F(\hat{x}) - \hat{p}_k^*(\hat{x})|_d)$ **then**
        $\hat{p}_k^* = \hat{p}_k^{tmp}, R = F - \hat{p}_k^*$;
    end
end
Return: p_k^*

Function Boost$(p_k, r_{k'}, k \geq k'$ with domain $[a, b])$:
    $p_{k'}(x) = \sum_{i=0}^{k'} (\hat{c}_i^{(\hat{p}_k)} \cdot \hat{s}_i^{(\hat{p}_k)} + c_i^{(r_{k'})}) \cdot x^i$;
    $\hat{p}_{k'} \leftarrow \mathrm{ScalePoly}(p_{k'}, [a, b])$;
    $\hat{p}_k(x) = \sum_{i=0}^{k'} (\hat{c}_i^{(\hat{p}_{k'})} \cdot x^i \cdot \hat{s}_i^{(\hat{p}_{k'})}) + \sum_{i=k'+1}^{k} (\hat{c}_i^{(\hat{p}_k)} \cdot x^i \cdot \hat{s}_i^{(\hat{p}_k)})$;
    Return: $\hat{p}_k$

最后，算法2在定义域内采样一些随机点，测试近似多项式是否满足精度要求。

2.2 OPPE设计

Algorithm 7: *OPPE* Algorithm (OPPE)

Config : Three parts plaintext parameters of $\hat{p}_k^m$: $\hat{W}$(without endpoint $\hat{w}_m$), $C = \{\hat{c}_{j,i}\}$ and $S = \{\hat{s}_{j,i}\}$.
Input : Secret input $[\hat{x}]$.
Return: The secret evaluation result of $[\hat{p}_k^m(\hat{x})]$.

$[\text{comp}] \leftarrow \text{GT}([\hat{x}], \hat{W})$ # compare x with each break point. ;
$[\text{mask}] \leftarrow \text{ADD}([\text{comp}], -\text{leftshift}([\text{comp}], 1))$;
for $i \leftarrow 0$ **to** $k-1$ **do**
    $[\text{coeff}]_i \leftarrow \hat{\sum}_{j=0}^{j=m-1} \text{MUL}([\text{mask}]_j, \hat{c}_{j,i})$;
    $[\text{scaler}]_i \leftarrow \hat{\sum}_{j=0}^{j=m-1} \text{MUL}([\text{mask}]_j, \hat{s}_{j,i})$;
end
$[\text{xterm}] \leftarrow \text{CalculateKx}([\hat{x}], k)$;
Return $\hat{\sum}_{i=0}^{i=k-1} (\text{MUL}(\text{MUL}([\text{coeff}]_i, [\text{xterm}]_i), [\text{scaler}]_i)$;
Function CalculateKx($[\hat{x}], k$):
    /* Calculate $[1, [\hat{x}], [\hat{x}]^2, ..., [\hat{x}]^k]$ */
    shift $\leftarrow 1$, $[\text{res}] \leftarrow [1, \text{tail}([\hat{x}], k)]$ /* repeat $[\hat{x}]$ k times */
    while shift $< k$ **do**
        $[\text{res}]_{\text{shift:}} = \text{MUL}([\text{res}]_{\text{shift:}}, [\text{res}]_{:-\text{shift}})$;
        shift $\times = 2$;
    end
    Return [res] ;

如算法7所示，多项式的安全计算遵循底层协议的计算，并且本文也利用了SIMD等MPC库广泛支持的优化算法提升性效率。

3. 实验评估

本文做了大量的实验，本博客只选择其中两部分说明：

Table 2: Examples in Performance Evaluations

$F(x)$	$\mathcal{S}$	✓	(k, m)	T_{Fit}	Communication (MB)			Computation time (ms)		
					Base	NFGen	**Save**	Base	NFGen	**SpeedUp**
$sigmoid(x) = \frac{1}{1+e^{-x}}$ $x \in [-50, +50], F(x) \in [0.0, 1.0]$ Non-linear buildling-blocks: 2	A	×	(10, 8)	4.3	618	263	**60%**	147	23	**6.3×**
	B	✓	(7, 10)	3.5	1	1	-5%	137	124	**1.1×**
	C	✓	(5, 14)	3.5	4	4	-5%	1155	802	**1.4×**
	D	✓	(5, 14)	3.5	18	19	-8%	1863	1525	**1.2×**
	E	✓	(5, 14)	3.5	212	308	-45%	75949	106857	0.7×
	F	✓	(5, 14)	3.5	207	234	-13%	9732	11224	0.9×
$tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$ $x \in [-50, +50], F(x) \in [-1.0, 1.0]$ Non-linear buildling-blocks: 3	A	×	(9, 8)	4.5	1876	216	**90%**	335	21	**15.7×**
	B	×	(5, 9)	3.2	13	1	**92%**	800	80	**10.0×**
	C	×	(5, 9)	3.2	19	3	**83%**	5901	597	**9.9×**
	D	×	(5, 9)	3.2	64	14	**78%**	8882	1115	**8.0×**
	E	×	(5, 9)	3.2	996	197	**80%**	337530	68550	**4.9×**
	F	×	(5, 9)	3.2	966	150	**84%**	45486	7309	**6.2×**
$soft\_sign(x) = \frac{x}{1+\lvert x\rvert}$ $x \in [-50, 50], F(x) \in [-1.0, 1.0]$ Non-linear buildling-blocks: 2	A	×	(8, 8)	1.9	518	231	**60%**	131	21	**6.1×**
	B	✓	NA	1.3	1	1	0%	79	78	1.0×
	C	✓	NA	1.3	2	2	0%	451	437	1.0×
	D	✓	NA	1.3	8	8	0%	741	753	1.0×
	E	✓	NA	1.3	52	52	0%	15507	15520	1.0×
	F	✓	NA	1.3	49	49	0%	2315	2373	1.0×
$Normal\_dis(x) = \frac{e^{-\frac{x^2}{2}}}{\sqrt{2\pi}}$ $x \in [-10, +10], F(x) \in [0.0, 0.4]$ Non-linear buildling-blocks: 1	A	×	(8, 12)	5.2	420	295	**30%**	67	24	**2.8×**
	B	×	(8, 12)	3.6	3	2	**45%**	4906	156	**31.5×**
	C	×	(8, 12)	3.6	7	5	**27%**	5029	970	**5.2×**
	D	×	(8, 12)	3.6	24	23	**5%**	6588	1846	**3.6×**
	E	×	(5, 22)	3.6	257	481	-87%	89740	166328	0.5×
	F	×	(8, 12)	3.6	249	301	-21%	14908	14861	1.0×
$Bs\_dis(x)$ [5] $= \left(\frac{\sqrt{x}+\sqrt{\frac{1}{x}}}{2\gamma x}\right)\phi\left(\frac{\sqrt{x}-\sqrt{\frac{1}{x}}}{\gamma}\right)$ $\gamma = 0.5, x \in [10^{-6}, 30], F(x) \in [0.0, 0.2]$ Non-linear buildling-blocks: 3	A	×	(10, 8)	4.0	2815	263	**90%**	630	22	**29.1×**
	B	×	(7, 11)	3.2	13	1	**89%**	11463	133	**86.1×**
	C	×	(5, 16)	3.2	23	5	**79%**	14631	915	**16.0×**
	D	×	(5, 16)	3.2	65	22	**66%**	19167	1763	**10.9×**
	E	×	(5, 16)	3.2	741	352	**53%**	239549	122325	**2.0×**
	F	×	(5, 16)	3.2	718	268	**63%**	42157	13136	**3.2×**

* T_{Fit} is the time for $\hat{p}_k^m$ fitting in seconds. ✓indicates whether baseline achieves the accuracy requirements.

表2中A-F表示不同的开源协议库。对于一些表中的非线性函数计算,，NFGen大多数获得了在精度、通信和时间方面的提升。但是有些协议库都某些函数进行了高度优化，NFGen跟这些方案比还略优逊色。

另一个实验，则是针对LR回归。

Table 6: Logistic Regression Speedups

Dataset	Method	Train(sec)	Test(sec)
Adult [23] $(48,842 \times 65)$	PrivPy	413.1	1.8
	NFGen	43.6 / 9.5×	0.8 / 2.3×
Bank [34] $(41,188 \times 63)$	PrivPy	72.8	1.6
	NFGen	20.4 / 3.6×	0.8 / 2.0×
Branch [36] $(400,000 \times 480)$	PrivPy	703.8	12.2
	NFGen	199.9 / 3.5×	6.9 / 1.8×

和PrivPy相比，性能方面获得了很好的提升。
其他实验和细节内容见原文。